Pavol Bielik

Machine Learning and Synthesis of Robust Programs

Diss. ETH No. 27449

DISS. ETH NO. 27449

MACHINE LEARNING AND SYNTHESIS OF ROBUST PROGRAMS

A thesis submitted to attain the degree of

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

presented by

PAVOL BIELIK

M.Sc., ETH Zurich

born on 7 February 1990
citizen of Slovak Republic

accepted on the recommendation of

Dr. Marc Brockschmidt (Microsoft Research)
Prof. Dr. Charles Sutton (University of Edinburgh, Google AI)
Prof. Dr. Eran Yahav (Technion)
Prof. Dr. Martin Vechev (ETH Zurich)

2021

Pavol Bielik: *Machine Learning and Synthesis of Robust Programs,* © 2021

ABSTRACT

Improving developer productivity is an important, but very difficult task, that researchers from both academia and industry have been trying to solve for decades. This has become even more challenging given the enormous scale at which today's software is produced. There is, however, an upside to this scale: the increased availability of code creates an exciting opportunity to learn from these large datasets.

The goal of this work is to leverage these datasets and to create programming tools that accomplish tasks that were previously difficult or practically infeasible. We address this problem, both at the foundational level by developing new techniques that learn over existing code and synthesize new programs, as well as at the application level, by creating software tools based on these models.

First, we address the core task of learning probabilistic models of code that achieve state-of-the-art precision and are applicable across a variety of programming languages. For this, we developed a novel probabilistic model, we identified the right program representation to be compiled into that model, and we designed suitable learning and inference algorithms. The key novelty of our approach is that our probabilistic model is parametrized by a learned program, rather than a set of non-interpretable weights, as typically done in machine learning.

Next, we address the problem of learning models of code that are not only accurate, but also robust. This is a critical issue as existing models have shown to be highly non-robust – a small input modification (e.g., code refactoring) can cause the model to consistently produce the wrong result, thus hindering the tool's adoption in practice and pose a potential security risk. This is a highly non-trivial task with several key challenges: learning the parts of the program relevant for the prediction without conditioning on the entire program, allowing the model to over-approximate the result when uncertain and developing models that learn compositional rules. In our work, we solve this problem from two perspectives: first, from the programming languages angle, we learn interpretable rules of a static analyzer, and second, from the machine learning perspective, we learn a robust deep learning model that infers type annotations for dynamically typed languages.

Finally, we develop two tools, `InferUI` and `FastSMT`, that automate the tedious and inefficient task of writing programs for two different application domains: writing relational layouts for the Android platform and writing strategies that are fast at solving SMT formulas. Further, both our tools significantly improve upon the programs written manually by domain experts; they prevent common layouts errors and achieve two orders of magnitude speed-up over the `Z3` solver, respectively. To make these tools practical, we combine program synthesis and machine learning. This allows us to synthesize programs from a single input-output example, while the machine learning component enables the synthesis to scale and generalize to real-world programs and formulas.

ZUSAMMENFASSUNG

Die Verbesserung der Entwicklerproduktivität ist eine wichtige, aber sehr schwierige Aufgabe, die Forscher aus Wissenschaft und Industrie seit Jahrzehnten versuchen zu lösen. Dies ist angesichts des enormen Umfangs, in dem die heutige Software hergestellt wird, noch schwieriger geworden. Diese Größenordnung hat jedoch einen Vorteil: Die erhöhte Verfügbarkeit von Code bietet eine aufregende Gelegenheit, aus diesen großen Datenmengen zu lernen.

Ziel dieser Arbeit ist es, diese Datensätze zu nutzen und Programmierwerkzeuge zu erstellen um Aufgaben zu erfüllen die zuvor schwierig oder praktisch nicht realisierbar waren. Wir lösen dieses Problem sowohl auf der grundlegenden Ebene, indem wir neue Techniken entwickeln, die über vorhandenen Code lernen und neue Programme synthetisieren, als auch auf Anwendungsebene, indem wir auf diesen Modellen basierende Softwaretools erstellen.

Zunächst befassen wir uns mit der Kernaufgabe, probabilistische Codemodelle zu erlernen, die auf dem neuesten Stand der Technik sind und auf eine Vielzahl von Programmiersprachen anwendbar sind. Zu diesem Zweck haben wir ein neuartiges probabilistisches Modell entwickelt, die richtige Programmdarstellung für dieses Modell identifiziert und geeignete Lern- und Inferenzalgorithmen entwickelt. Die wichtigste Neuerung unseres Ansatzes besteht darin, dass unser Wahrscheinlichkeitsmodell durch ein erlerntes Programm und nicht durch eine Reihe nicht interpretierbarer Gewichte parametrisiert wird, wie dies normalerweise beim maschinellen Lernen der Fall ist.

Als nächstes befassen wir uns mit dem Problem des Lernens von Codemodellen, die nicht nur genau, sondern auch robust sind. Dies ist ein kritisches Problem, da sich vorhandene Modelle als äußerst nicht robust erwiesen haben. Kleine Eingabemodifikationen (z. B. Code-Refactoring) können dazu führen, dass das Modell durchweg das falsche Ergebnis liefert, wodurch die Übernahme des Tools in die Praxis behindert wird und ein potenzielles Sicherheitsrisiko darstellt. Dies ist eine höchst komplexe Aufgabe mit mehreren zentralen Herausforderungen: Erlernen der für die Vorhersage relevanten Teile des Programms ohne Konditionierung des gesamten Programms, sodass das Modell das Ergebnis bei Unsicherheiten übersteuern kann, und erlernen der Kompositionsregeln ermöglicht. In unserer Arbeit lösen wir dieses Problem aus zwei Perspektiven: Erstens lernen wir aus Sicht der Programmiersprachen interpretierbare Regeln eines statischen Analysators und zweitens lernen wir aus Sicht des maschinellen Lernens ein robustes Deep-Learning-Modell, für die Ableitung der Typanmerkungen für dynamische Programmiersprache.

Schließlich entwickeln wir zwei Tools, InferUI und FastSMT, die die mühsame und ineffiziente Aufgabe des Schreibens von Programmen für zwei verschiedene Anwendungsbereiche automatisieren: das Schreiben von Beziehungslayouts für

die Android-Plattform und das Schreiben von Strategien, mit denen SMT-Formeln schnell gelöst werden können. Darüber hinaus verbessern unsere beiden Tools die von Fachexperten manuell geschriebenen Programme erheblich. Sie verhindern häufige Layoutfehler und erreichen eine Beschleunigung um zwei Größenordnungen gegenüber dem Z3-Solver. Um diese Tools praktisch zu gestalten, kombinieren wir Programmsynthese und maschinelles Lernen. Dies ermöglicht es uns, Programme aus einem einzigen Eingabe-Ausgabe-Beispiel zu synthetisieren, während die maschinelle Lernkomponente es ermöglicht, die Synthese zu skalieren und auf reale Programme und Formeln zu verallgemeinern.

THESIS PUBLICATIONS

This thesis is based on the following publications:

Adversarial Robustness for Code [1]
Pavol Bielik, Martin Vechev ICML'20

Learning to Solve SMT Formulas [2]
Mislav Balunovic, Pavol Bielik, Martin Vechev NeurIPS'18 (Oral)

Robust Relational Layout Synthesis from Examples [3]
Pavol Bielik, Marc Fischer, Martin Vechev OOSPLA'18

Program Synthesis for Character Level Language Modelling [4]
Pavol Bielik, Veselin Raychev, Martin Vechev ICLR'17

Learning a Static Analyzer from Data [5]
Pavol Bielik, Veselin Raychev, Martin Vechev CAV'17

PHOG: Probabilistic Model for Code [6]
Pavol Bielik, Veselin Raychev, Martin Vechev ICML'16

The following publications were part of my PhD research and present results that are supplemental to this work or build upon results of this thesis:

Guiding Program Synthesis by Learning to Generate Examples [7]
Larissa Laich, Pavol Bielik, Martin Vechev ICLR'20

Probabilistic Model for Code with Decision Trees [8]
Veselin Raychev, Pavol Bielik, Martin Vechev OOPSLA'16

Learning Programs from Noisy Data [9]
Veselin Raychev, Pavol Bielik, Andreas Krause, Martin Vechev POPL'16

The remaining publications were part of my PhD research, but are not covered in this thesis. The topics of these publications are outside of the scope of the material covered here:

Automated Discovery of Adaptive Attacks on Adversarial Defenses [10]
Chengyuan Yao, Pavol Bielik, Petar Tsankov, Martin Vechev In Submission

Robustness Certification with Generative Models [11]
Matthew Mirman, Alexander Hägele, Timon Gehr, Pavol Bielik, Martin Vechev PLDI'21

Adversarial Attacks on Probabilistic Autoregressive Forecasting Models [12]
Raphaël Dang-Nhu, Gagandeep Singh, Pavol Bielik, Martin Vechev ICML'20

Learning to Infer User Interface Attributes from Images [13]
Philippe Schlattner, Pavol Bielik, Martin Vechev ArXiv'19

SDNRacer: Concurrency Analysis for Software-Defined Networks [14]
Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, Martin Vechev PLDI'16

Scalable Concurrency Analysis for Android Applications [15]
Pavol Bielik, Veselin Raychev, Martin Vechev OOPSLA'15

Detecting Concurrency Violations in Software-Defined Networks [16]
Jeremie Miserez, Pavol Bielik, Ahmed El-Hassany, Laurent Vanbever, Martin Vechev SOSR'15

Programming with Big Code: Lessons, Techniques and Applications [17]
Pavol Bielik, Veselin Raychev, Martin Vechev SNAPL'15

SOFTWARE AND DATASETS

The following software and datasets were made publicly available as part of this thesis:

DATASETS

Py150 `https://www.sri.inf.ethz.ch/py150`
Corpus of 150,000 Python programs, which we originally published in 2016. Since then, it has been used or compared against in more than 100 works, revised by researchers at Google Brain [18], and included in a number of initiatives for the domain of code and beyond, such as:

- *CodeXGLUE: A benchmark dataset and open challenge for code intelligence* [19] [Microsoft]
- *Wilds: A benchmark of in-the-Wild distribution shifts* [20] [Standford, Berkley, Cornell, et. al.]
- publicly available at Hugging Face and Kaggle

Js150 `https://www.sri.inf.ethz.ch/js150`
Javascript version of the Py150 dataset containing 150,00 JavaScript programs.

PlayStore & GitHub Top 500 `https://github.com/eth-sri/inferui`
Dataset containing Android layouts and their associated metadata obtained by executing PlayStore and top 500 Android applications on GitHub.

SOFTWARE

PHOG `https://github.com/eth-sri/ModelsPHOG`
Code to replicate results of our Probabilistic Higher-Order Grammar model [6].

Robust Code `https://github.com/eth-sri/robust-code`
Implementation of all the techniques and pre-trained models used to replicate the results of our work on adversarial robustness for code [1].

FastSMT `https://github.com/eth-sri/fastsmt`
Implementation of all the techniques and pre-trained models used to replicate the results of our work on FastSMT [2]. The project website is available at:

`http://fastsmt.ethz.ch`

InferUI `https://github.com/eth-sri/inferui`
Implementation of the Android layout synthesizer and code to reproduce the results [3, 7]. The project website is available at:

`https://inferui.com`

ACKNOWLEDGEMENTS

I was fortunate to be surrounded by many great people, who helped me throughout this journey. I am very grateful for all your guidance and support; this thesis and all the lessons that I have learned during my PhD would have not been possible without all of you.

First of all, I would like to thank Prof. Martin Vechev for giving me the opportunity to join his lab, for being my mentor, for teaching me how to become a researcher, and for helping me to improve. Thank you, Martin, for all the challenging projects, for all the insightful meetings, for all your support, and for all the people that I had the chance to meet through you.

I am also very grateful to the committee members - Prof. Eran Yahav, Prof. Charles Sutton and Dr. Marc Brockschmidt - for taking the time to review my thesis and to provide detailed feedback. Further, thank you for all the discussions we had at various conferences throughout my PhD, as well as for all your work that provided countless inspirations for my own research.

I would also like to thank Prof. Thomas Gross, Prof. Andreas Krause, Prof. Peter Müller, Prof. Michael Pradel, Prof. Markus Püschel, Prof. Zhendong Su, and Prof. Laurent Vanbever for their advice and all the inspiring discussions. I have definitely learned a lot from each of you.

Working on all the projects and papers would have been much harder without the help of all my collaborators. Special thanks to Veselin Raychev - my first officemate - who helped me a lot at the beginning of my PhD. Also many thanks to my colleagues that I had a chance to work on a project together - Mislav Balunović, Marc Fischer, Ahmed El-Hassany, Gagandeep Singh, Timon Gehr, Matthew Mirman, Petar Tsankov - and to all the students I had the opportunity to supervise - Korbinian Abstreiter, Christian Fehlmann, Christine Zeller, Svetoslav Karaivanov, Jeremie Miserez, Benjamin Mularczyk, Prabhakaran Santhanam, Ylli Muhadri, Pavle Djordjević, Cristina Cristescu, Philippe Schlattner, Larissa Laich, Matúš Žilinec, Emilien Pilloud, Wonryong Ryou, Chengyuan Yao, Meet Vora - you taught me how to become a better advisor.

I am also very grateful to all the other colleagues from the SRILab - Maximilian Baader, Benjamin Bichsel, Andrei Dan, the two Mitkos (Dimitar) Dimitrov, Dana Drachsler-Cohen, Timon Gehr, Inna Grijnevitch, Jingxuan He, Pesho Ivanov, Matthew Mirman, Sasa Misailovic, Momchil Peychev, Samuel Steffen, and Petar Tsankov - for all the discussions, for all the group retreats and the birthday cakes. I apologize for not joining you for lunch and dinner more often.

I would also like to thank Alex Summers and Malte Schwerhoff for providing different perspectives to my research problems. Alex and Malte, your opinions and your friendship are highly appreciated. Also many thanks for the relaxing board games evenings.

There are many more colleagues from various research groups with whom I was fortunate to interact throughout these years while teaching, supervising students, or attending talks: Mitko (Dimitar) Asenov, Vytautas Astrauskas, Stefan Bauer, Lucas Brutschy, Alexandra Bugariu, Victoria Caparrós Cabezas, Luca Della Toffola, Jérôme Dohrau, Marco Eilers, Michael Faes, Octavian Ganea, Uri Juhasz, Remi Meier, Georg Ofenbeck, Gaurav Parthasarathy, Federico Poli, Francois Serre, Daniele Spampinato, Alen Stojanov, Arshavir Ter-Gabrielyan, Caterina Urban, and Felix Wolf. Thank you all for your valuable input.

My life at ETH would have been harder and less organized without the help of the administrative assistants Mrs. Fiorella Meyer and Mrs. Marlies Weissert. Thank you Fiorella and Marlies for your kindness and for always finding solutions to my problems. Also, a big thank you to Mrs. Denise Spicher and to all the people from ISG, especially to Thomas Schmid.

Outside of ETH, dancing and football were the main sources of my relax. I would like to thank all my friends from Folklór 75, especially to Růženka Hraška (Fantová), Betka Hricová, Matthias Haag and Pavol Ondrisek, without whom I would have never joined and figured out that dancing can be so enjoyable. Thank you also for not getting upset when I was dancing or playing badly because I was thinking about my projects.

I would also like to thank all the people who made my life in Zurich enjoyable and more interesting, including numerous bike trips, barceques, nice evenings outside or just being generally nice people to be around. Special thanks goes to Lukáš Kurilla, Rasto Marko, Mojmír Mutný and Martin Štefánik.

Last but not least, I am deeply grateful to my parents and to my brother for their support. You encouraged me to follow this path and you taught me the value of kindness, respect, and hard work. I would have not been at ETH without you.

CONTENTS

1 INTRODUCTION 1
- 1.1 This Work 2
- 1.2 Contributions 3
- 1.3 Overview of the techniques and tools developed in our work 5
  - 1.3.1 Probabilistic Models of Code 5
  - 1.3.2 Robust and Accurate Models of Code 7
  - 1.3.3 Learning Strategies to Solve SMT Formulas 11
  - 1.3.4 From Images to Layouts 12
- 1.4 Summary 14

2 PHOG: PROBABILISTIC MODEL FOR CODE 15
- 2.1 Parametrizing language models with programs 20
  - 2.1.1 Simple Programs 21
  - 2.1.2 `CondGen`: A Domain Specific Language for Describing Conditioning Context 25
  - 2.1.3 Small-step Semantics of `CondGen` 28
- 2.2 From `CondGen` to a Probabilistic Model 33
  - 2.2.1 Extension: Predicting Out-of-Vocabulary Labels 35
- 2.3 Learning 37
  - 2.3.1 Learning `SimplePrograms` 38
  - 2.3.2 Learning `EqualityPrograms` 39
  - 2.3.3 Learning `BranchPrograms` 39
- 2.4 Evaluation: Probabilistic Models of Code 43
  - 2.4.1 Main Results 47
  - 2.4.2 Probabilistic Model for JavaScript 48
  - 2.4.3 Predicting Out-of-Vocabulary Labels 51
  - 2.4.4 Learned Programs 52
  - 2.4.5 Probabilistic Model for Python 54
- 2.5 Evaluation: Character Level Language Modelling 54
  - 2.5.1 Language Models 55
  - 2.5.2 Model Performance 57
- 2.6 Related Work 58
  - 2.6.1 Probabilistic Models of Code 58
  - 2.6.2 Decision Tree Learning 60
  - 2.6.3 Program Synthesis 61
- 2.7 Summary 61

3 LEARNING STATIC ANALYZERS 63
- 3.1 Overview 66
- 3.2 Checking Analyzer Correctness 70

3.3 Learning Analysis Rules 71
3.4 The Adversary: Testing an Analyzer 73
3.4.1 Choosing Relevant Samples to Test via Equivalence Classes 74
3.4.2 Defining Relevant Program Modifications 74
3.4.3 Choosing Modification Positions 76
3.5 Points-to Analysis 77
3.5.1 Instantiating our Learning Approach 78
3.5.2 Language for Points-To Inference Rules 79
3.6 Allocation Site Analysis 81
3.6.1 Instantiating our Learning Approach 82
3.7 Implementation 83
3.7.1 Obtaining Concrete Behaviors of a Program 84
3.7.2 Checking Analysis Correctness 84
3.7.3 JavaScript Restrictions 85
3.8 Evaluation 85
3.8.1 Learning Points-to Analysis Rules for JavaScript 86
3.8.2 Learning Allocation Site Analysis for JavaScript 88
3.8.3 Analysis Generalization 90
3.9 Related Work 91
3.10 Conclusion 93

4 ROBUSTNESS FOR MODELS OF CODE 95
4.1 Overview: Accurate and Robust Models of Code 98
4.2 Training Neural Models to Abstain 102
4.3 Adversarial Training for Code 104
4.3.1 Program Modifications 105
4.3.2 Finding Adversarial Examples 106
4.4 Learning to Refine Representations 109
4.5 Training Algorithm 113
4.6 Evaluation 115
4.6.1 Accurate and Adversarially Robust Models 119
4.7 Related Work 122
4.8 Conclusion and Discussion 124

5 FASTSMT: LEARNING STRATEGIES TO SOLVE FORMULAS 127
5.1 Overview 130
5.2 Language for Expressing SMT Solver Strategies 132
5.3 Learning a Neural Policy 133
5.3.1 Models 133
5.3.2 Training 135
5.4 Synthesizing a combined, interpretable strategy 139
5.5 Evaluation 142
5.5.1 Comparison to the State-of-the-art SMT Solver Z3 143
5.5.2 Effectivness of the Learned Models 145

5.6 Related work 147
5.7 Conclusion 148

6 INFERUI: FROM IMAGES TO LAYOUTS 151
6.1 Practical Benefits of our Approach 156
6.2 Capturing Relational Constraints 158
6.2.1 Layout Constraint Solving 162
6.3 Single Device Relational Layout Synthesis 164
6.4 Generalizing Layouts to Multiple Devices 166
6.4.1 Robustness Properties 167
6.4.2 Incorporating User Feedback 169
6.5 Scaling Synthesis with a Probabilistic Constraints Model 170
6.5.1 Guiding the Synthesis via Probabilistic Model of Constraints 170
6.5.2 Guiding the Synthesis via Probabilistic Model of Outputs 175
6.6 Evaluation 176
6.6.1 Scalability & Runtime 177
6.6.2 Precision & Robustness 179
6.6.3 Synthesising Natural Layouts 180
6.6.4 Incorporating User Feedback 181
6.7 Related Work 181
6.8 Conclusion 183

7 CONCLUSION AND FUTURE WORK 185
7.1 Future Work 187

BIBLIOGRAPHY 191

1 INTRODUCTION

The increasing availability of large datasets is a key trend that acts as a catalyst in many domains, such as natural language processing, computer vision or speech recognition. The domain of programming is no exception and has seen an enormous increase in large and high quality code bases written in different programming languages and targeting diverse types of applications. For example, only on GitHub, there are more than 100 million repositories out of which more than 33 million are public. To put these numbers into perspective, just twelve years ago the number of all GitHub repositories was only 33 thousand while now, there are 1.6 new repositories created every second [21].

Learning from "Big Code" [22, 23] takes advantage of the availability of these large codebases and aims to develop new techniques and tools that help programmers to be more productive in a wide range of tasks they perform daily – implementing new code, maintaining or refactoring existing code, debugging, testing, fixing bugs, reviewing code, understanding code and many more. In other words, tools based on learning from large codebases have a tremendous potential to improve all aspects of the software development as it is done today.

Naturally, this created a significant interest from both academia and industry and resulted in several tools already deployed in practice, including `JSNice` [22] – a statistical renaming and type inference tool for JavaScript used by more than 100 000 users yearly, `Getafix` [24] – an industrially deployed tool that automatically fixes selected classes of Java bugs (i.e., null dereferences or incorrect API calls) by learning from human-written fixes, or `TypeWriter` [25] – another industrially deployed tool that combines statistical type inference for Python with search-based validation used to assess the prediction correctness.

However, the number of tools that learn over programs and are deployed in practice are only few and far between. Learning from code is still an emerging research area with the number of publications doubling in the last year [23]. The main reasons for this are twofold – *(i)* given that such tools have the potential to improve almost any aspect of software development, much of the recent research effort is dedicated to exploring new tasks, and *(ii)* there has been an enormous interest in using deep learning to train models of code. In particular, the deep learning based approaches are currently so popular that the majority of the research papers published in the last two years employ some form of neural model. For example, deep learning models have been trained for a wide range of existing and new tasks, including code completion [26–29], type prediction [30–33], code summarization [34–37], code classification [38, 39], bug detection [40–43], learning loop invariants [44, 45], malware detection [46], program translation [47, 48], neural decompilation [49], or code search [50–52].

1.1 THIS WORK

In this work, we focus on learning from large codebases, both at the foundational level by developing new techniques that learn over existing code and synthesize new programs, and at the application level, by creating software tools based on these models. To achieve this, we address a number of hard problems at the intersection of programming languages, program synthesis and machine learning.

CHALLENGES The first challenging problem we consider is designing new probabilistic models of code that achieve state-of-the-art precision and are applicable across a range of programming languages. This includes addressing several key challenges including: developing a novel probabilistic model, learning the right program representation to be compiled into that model, designing the appropriate learning and inference algorithms. The key novelty of our approach is that our probabilistic model is *parametrized by a learned program*, rather than a set of non-interpretable weights, as done typically in machine learning.

The second challenge is learning models and synthesizing programs that are not only accurate, but also robust. When learning static analyzers (e.g., points-to analysis rules), the notion of robustness is a natural requirement since the program analyzers should produce sound results for all valid programs in a given programming language. However, the same robustness property is highly desirable also for other models of code, including those based on deep learning. In fact, the issue of robustness is especially critical for deep learning models, which have been shown to be very brittle in other domains including natural language processing [53–55].

The third challenge we address is to automate the tedious, error-prone, and inefficient task of writing programs for two different application domains: writing relational layouts for the Android platform and writing strategies that are fast at solving SMT formulas. For both domains, solving this task requires a careful combination of program synthesis and machine learning techniques. This combination is critical for developing tools that scale to real-world problems and that can be directly integrated into existing tools (e.g., Z3 solver).

COMBINING MACHINE LEARNING AND PROGRAMMING LANGUAGES In our work, we apply both machine learning and programming languages techniques to address the same problem but from different perspectives, as well as explore different ways of integrating them together. Machine learning enables learning *accurate* models that handle *noisy data* and has the potential to *automate* many design steps found in traditional programming language systems (e.g., in program synthesis)[1]. Further, machine learning models can often be applied to new tasks relatively fast, which makes them very desirable (given that a suitable dataset is available). Addressing the problem from the programming languages perspective

---

1 Strengths of each domain are based on the current state-of-the-art but are not specific to the given domain. For example, there is a large body of work on interpreting deep learning models [56–58], neural network verification [59] or extending traditional program synthesis techniques to handle noise [9, 60].

FIGURE 1.1: High-level illustration of two perspectives considered in our work that learn to predict program properties from a dataset of programs. *Learning over programs* directly learns a machine learning model that predicts a given property. In contrast, *learning programs* learns a program from a domain-specific language which, when evaluated, computes the property.

allows building systems that are more *interpretable*, with *provable guarantees*, can be *easily integrated* into existing systems and can work well even with *small datasets*. The disadvantage, however, is that building such systems is typically more time consuming and they do not readily apply to other tasks.

The main difference between using machine learning and programming language techniques considered in our work is that the former *learns over programs* while the latter *learns programs*, as illustrated in Figure 1.1. In fact, for machine learning, the programs are often just a different type of input data over which the machine learning models are trained and the main goal is to correctly predict a given property (e.g., the type of a variable, whether a program has a bug). In contrast, when *learning programs*, the goal is to learn another program (e.g., a static type inference analysis) that computes the desired property. Learning such programs is useful beyond computing the property, as they can be inspected, understood and integrated into existing systems by a domain expert.

1.2 CONTRIBUTIONS

The main contributions of this dissertation are summarized below:

CHAPTER 2: Designs a probabilistic model of code called probabilistic higher order grammar (PHOG). This is a non-neural based model that learns to condition each prediction on a small set of relevant parts of the program. The key insight is that the conditioning is expressed as a learned program in a domain-specific language. That is, the probabilistic model is *parametrized by a program*. The model generalizes over a number of prior works [61–64] and we evaluated it by learning probabilistic models for two programming languages (JavaScript and Python), as well as character level language models for natural language text from Wikipedia. Even though our models are non-neural and are instantiated with a simple maximum likelihood model (based on counting), they are comparable and even better than a number of sophisticated deep learning models developed *after* our work.

CHAPTER 3: Presents a new, automated approach for learning static analyzers from data. Our approach builds on the results from Chapter 2 and extends

it to allow learning analyzers that are sound and precise (with respect to the training dataset). This is instead of considering a probabilistic setting in which the model is allowed to make mistakes, which is unsound. To achieve this, we designed a counter-example guided learning procedure that generates new programs beyond those in the initial dataset, critical for discovering corner cases and ensuring the learned analysis generalizes to unseen programs. In particular, we show that our approach can successfully learn allocation site and a restricted set of points-to analysis rules for JavaScript that are not supported by the state-of-the-art static analyzer Facebook Flow [65].

CHAPTER 4: Addresses the task of learning robust models of code. This is a similar problem to the one considered in Chapter 3, but with the focus on solving it from a machine learning perspective of *learning over programs* rather than *learning programs*. That is, instead of learning the static analysis explicitly as a program, we learn a robust neural network that produces the most likely analysis output. Even though the underlying models are different, we will show that the technical solutions need to address the same set of challenges when learning sound and precise models. Further, we will show the parallels between the techniques used in machine learning (adversarial robustness, learning to abstain) and programming languages (counter-example guided synthesis, over-approximation) which have very similar goals but are called differently by the different research communities.

CHAPTER 5: Shows how both *learning over programs* and *learning programs* concepts can be combined to solve a given task efficiently. Concretely, we present a new approach for learning strategies that solve SMT formulas consisting of two main steps: *(i)* train a neural policy that applies a sequence of equisatisfiable transformations until the STM formula is solved, and *(ii)* synthesize an interpretable representation of the policy decisions in the form of a loop-free program with branches. Extracting a program that represents the policy decisions allows us to integrate our approach directly into the underlying SMT solver and avoids the need for the expensive evaluation of the learned policy at inference time. We show that our approach is effective in practice – it solves 27% more formulas over a range of benchmarks and achieves up to 100× runtime improvement over the state-of-the-art SMT solver Z3 [66].

CHAPTER 6: Describes a new application domain that learns layouts and their attributes from images. Concretely, given an image of an application design, we synthesize a relational layout that when rendered, places all the components at that same location, such that it looks visually the same. In order to build an end-to-end system capable of generating such layouts, a number of challenging tasks need to be addressed, including: *(i)* identifying the type, size and location of various user interface components in the image, *(ii)* synthesizing a layout that positions the identified components at the correct location on the screen while generalizing to unseen devices with different

LEARNING OVER PROGRAMS LEARNING PROGRAMS

INSTANTIATION (CHAPTER 2): PROBABILISTIC MODELS OF CODE

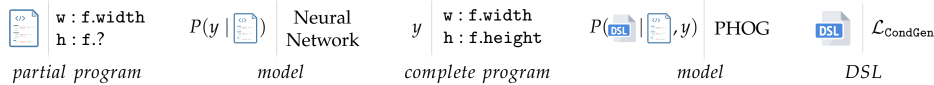


FIGURE 1.2: An overview of the instantiation of our work for the task of learning probabilistic models of code (i.e., predicting the next program token to write).

dimensions, and *(iii)* learning to correctly predict the attributes of each component (e.g., shadows, colors, borders, etc.), such that it looks visually the same as in the original image. We demonstrate how both machine learning and programming language techniques can be used to address these challenges by instantiating our approach for the Android platform.

1.3 OVERVIEW OF THE TECHNIQUES AND TOOLS DEVELOPED IN OUR WORK

In the rest of this section, we give an overview of the techniques and applications explored in this work. Concretely, we focus on highlighting how *learning over programs* and *learning programs* are combined and instantiated for different domains.

1.3.1 *Probabilistic Models of Code*

We start with learning a generative probabilistic model of code, that is, a probability distribution over programs. This is a natural first task since a probabilistic model of code is a core component for many tasks including code completion [6, 27], deminification [22], patch generation [67], translation between programming languages and others. This also applies to our work, where in Chapter 3 we build upon the probabilistic model of code defined in Chapter 2 to learn static analyzers, while in Chapter 6 we use it to guide program synthesizers.

More formally, a probabilistic model of code learns a probability distribution $P(y \mid \boldsymbol{x})$ given a training dataset $\mathbb{D} = \{(\boldsymbol{x}_j, y_j)\}_{j=1}^n$ of n samples where $\boldsymbol{x}_j$ are inputs (partial programs) and y_j are outputs (correct predictions for the partial programs). The instantiation for both *learning over programs* and *learning programs* approaches used in our work is shown in Figure 1.2. For the former, we use state-of-the-art neural models to represent the probability distribution $P(y \mid \boldsymbol{x})$. For the latter, one of the contributions of this work is developing a new model, called probabilistic higher order grammar (PHOG). While we leave the technical discussion of neural networks and PHOG to Chapter 2, we highlight the concepts used in both (though often using different names) in Table 1.1.

CONCEPT	Neural Networks	PHOG (our work)
OUT-OF-VOCABULARY	pointer networks	equality programs (§ 2.2.1)
	subword information	-
ATTENTION	soft (and hard)	hard (§ 2.1)
INTERPRETABILITY	attention & gradients	programs & executions (§ 2.1)
COVERAGE	activation similarity	path coverage (§ 3.4.1)
STATE	LSTM cell state	state programs (§ 2.1.2.2)
EMBEDDINGS	continuous	discrete (§ 2.2)

TABLE 1.1: Comparison of various concepts used when training neural models and the probabilistic higher order grammar (PHOG) developed in our work.

OUT-OF-VOCABULARY WORDS A common challenge inherent to language modelling is handling the cases where words appearing at inference time are different than those seen during training. This can be especially problematic for code, where much of the variety comes from user defined identifiers, strings and constants [8, 68]. To address this issue, two general techniques exist – using subword information and copying words from other parts of the input. For neural networks, the former technique is used to *enrich word vectors with subword information* [69–72] while the latter is known as *pointer networks* [27, 73]. For PHOG, we address this issue using equality programs – our technique that has been developed concurrently to *pointer networks* but for non-neural models.

ATTENTION Attention is a powerful technique used in deep learning [74–77], which allows the model to use the information from any part of the input instead of forcing the model to summarize the input into a fixed-length vector. The same concept, referred to as the *conditioning set*, is also used in PHOG and allows the model to selectively condition each prediction on the relevant parts of the input. The main difference is that in PHOG, the conditioning set is small and discrete (similar to hard attention in neural networks [78]). This is in contrast to soft attention that contains the probability distribution over all the inputs.

INTERPRETABILITY In terms of interpretability, there has been a large body of works developed for neural networks including methods that create proxy models of neural networks [79–82], identify relevant parts of the input via gradients [83–86] or optimization [86], find interpretations of high level features [83, 85, 87, 88], black-box methods [56, 89, 90] or using attention mechanisms (e.g., [91–93] for graphs). In comparison, *learning programs* allows interpretability by construction since the output is a program that can be inspected and executed. Naturally, the interpretability of inspecting programs by hand is typically useful only if the resulting programs are small and concise.

COVERAGE A side effect of the fact that the learned programs can be inspected and executed is that we can take advantage of traditional program analysis techniques to analyze them. One example where this is beneficial is when testing the correctness of the learned model. For programs, we show how coverage guided fuzzing can be used to efficiently find inputs that explore all possible loop-free paths of the analyzed program (in Chapter 3). For neural networks, similar ideas have been recently explored by grouping inputs together based on the similarity of their activations [94].

STATE Another concept used both in neural networks and PHOG is the notion of *state*. For neural networks, the most widely used implementation of a state is that of an LSTM cell [95]. For example, this allows the model to learn cells suitable to specific contexts, such as inside quotes, inside comments or sensitive to expression depth [96]. For PHOG, the notion of state is supported using *state programs* that extend the model with a stack capable of adding and removing different states. While less expressive than LSTM, *state programs* do allow expressing many of the same contexts including tracking quotes or comments.

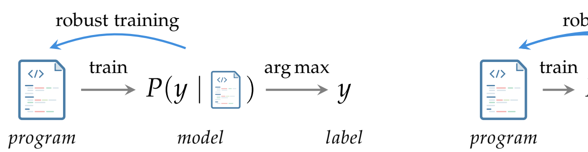
EMBEDDINGS Finally, a crucial property of neural networks is that they represent input words as high dimensional real vectors (i.e., word embeddings). In contrast, PHOG keeps the original discrete representation, effectively treating all words as independent of each other (i.e., the distance between all words in the vocabulary is the same). However, we designed the PHOG model such that it can be instantiated with any probabilistic model, including neural networks, log-bilinears models, support vector machines and more. As a result, the PHOG model can also take advantage of the continuous word representations, if instantiated with a probabilistic model that supports them.

1.3.2 *Robust and Accurate Models of Code*

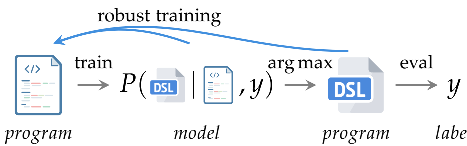
Despite substantial progress on training accurate models of code, the issue of robustness has been mostly overlooked. Yet, this is an important problem across various domains (e.g., computer vision [97–100], natural language processing [55, 101–105]) where neural models have been shown to be highly non-robust - a small input modification (e.g., pixel change) can cause the model to consistently predict the wrong label [53, 54].

For the domain of code, the input modifications correspond to common code refactoring a user might make, such as renaming variables, adding new code, or various label and semantic preserving changes. For a model to be robust, it should compute the correct prediction both on the original input program, as well as on all its possible modifications. However, as existing models of code optimize only for accuracy and do not consider the notion of robustness, it is not difficult to show that they are highly non-robust even for simple program transformations such as variable renaming or changing constants.

LEARNING OVER PROGRAMS



LEARNING PROGRAMS



INSTANTIATION (CHAPTER 3 & 4): ROBUST MODELS OF CODE

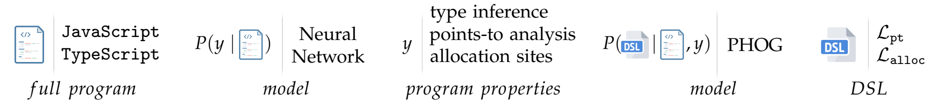


FIGURE 1.3: Instantiation of our work for the task of training robust models of code. Here, even though both approaches solve the same task independently of each other, they share many common concepts as shown in Table 1.2.

In our work, we address the challenge of training robust and accurate models of code from two independent perspectives – *learning over programs* (using neural networks in Chapter 4) and *learning programs* (building upon PHOG in Chapter 3). We illustrate their concrete instantiations in Figure 1.3 which considers three different tasks – type inference, points-to analysis and allocation site analysis. As can be seen, we selected the tasks that correspond to existing static analyses that domain experts write by hand. The reasons for this are threefold:

- The learned programs from Chapter 3 can be used by domain experts to design static analyzers faster, by discovering parts of the analyzer from data.
- It enables us to study the robustness for tasks without ambiguous labels. This is in contrast to the inherently probabilistic, and ambiguous, setting of some of the other tasks such as code completion[2].
- We can collect suitable datasets automatically, by taking advantage of the fact that the programs are executable and that the analyzer must approximate the concrete program behaviour. Thus, we can obtain a dataset by executing a large amount of programs in a given programming language with some inputs, and obtaining a subset of the concrete semantics for each program.

In what follows, we highlight common concepts used by both approaches to train robust models, as summarized in Table 1.2.

ADVERSARIAL TRAINING At a high-level, both approaches use a variant of adversarial training [53] which instead of minimizing the expected loss on the original distribution of programs $\mathbb{E}_{(x,y)\sim D}[\ell(f(x),y)]$ (as done in standard training),

2 The issue with standard robustness in ambiguous settings is that one needs to additionally differentiate between: *(i)* the case where the model predicts a different, yet valid label, due to ambiguity, and *(ii)* the case where the model predicts an incorrect label. This is a non-trivial task since the set of all valid labels is typically not known. For example, in code completion we observe a single variable name used by the developer, even though different users might prefer different variable names in the same context.

CONCEPT	Neural Networks	PHOG
TRAINING	adversarial (§ 4.3)	adversarial (§ 3.3)
FINDING COUNTER-EXAMPLES	gradients (§ 4.3.2)	path coverage (§ 3.4)
	guided (§ 4.3.2)	guided (§ 3.4)
REPRESENTATION REFINEMENT	remove edges (§ 4.4)	extract nodes (§ 3.5.2)
APPROXIMATION	abstain (§ 4.2)	lattice (§ 3.3)
COMPOSITIONALITY	sequence of models (§ 4.5)	fixpoint
PROGRAM TRANSFORMATIONS	semantic & label preserving (§ 3.4.2, § 4.6)	

■ Novel contribution, ■ Adapting existing concepts to a new domain, □ Other works

TABLE 1.2: Comparison of various concepts used in our work for training robust models of code using two different perspectives – *learning over programs* using neural networks and *learning programs* using PHOG.

minimizes the expected *adversarial loss* $\mathbb{E}_{(x,y)\sim D}[\max_{\delta\subseteq\Delta(x)} \ell(f(x+\delta),y)]$. Here, f is the model (e.g., neural network), ℓ is the loss (e.g., cross-entropy), δ is a program transformation and $\Delta(x)$ is a set of all valid program transformations defined over x. That is, we minimize the worst case loss obtained by applying a valid modification to the original program x, which is similar to finding a counter-example.

FINDING COUNTER-EXAMPLES The first challenge in applying adversarial training to code is solving the inner $\max_{\delta\subseteq\Delta(x)}$. For neural networks, this is achieved using gradient based techniques [106], by sampling at random or using other techniques to guide the search (e.g., for cases where gradient based techniques are not applicable). For PHOG, we take advantage of the fact that the model is expressed as a program. This allows us to: *(i)* sample inputs from the dataset such that they cover all the execution paths of the synthesized program, and *(ii)* use execution traces to select which transformations are likely to affect the model output.

REPRESENTATION REFINEMENT For training to be successful, it should be performed using a suitable program representation. This is especially important for adversarial training where an adversary is actively trying to break the model. For code, such an adversary is quite powerful since existing neural models of code typically process the entire program, which can contain hundreds of lines of code. This is problematic, as it means that any program change can affect all predictions and there can be infinitely many program changes. To address this issue we develop a novel technique that: *(i)* learns which parts of the input program are relevant for the given prediction, and *(ii)* refines the model representation such that only relevant program parts are used as input. Essentially, the technique automatically learns an abstraction which given a program, produces a relevant representation of that program. For *neural networks* we instantiate this concept by representing

programs as graphs that are sparsified as part of the training. When *learning programs* using PHOG, we also represent programs as graphs (trees), but learn to extract concrete nodes relevant for the prediction instead of only removing edges.

APPROXIMATION The next concept is the ability to over-approximate the correct solution when the learned model is uncertain. This notion is natural for program analysis which typically computes an abstract representation of the program's concrete behaviors using a lattice of abstract facts equipped with an ordering between them. This allows computing results with various degrees of precision, all of which are sound over-approximations of the correct result. In other words, the analysis computes very precise results when it is certain (i.e., knows enough program facts to support this decision) and will soundly approximate the results when fewer program facts can be deduced. In the worst case, the analysis returns the top element of the lattice, which over-approximates all concrete program behaviours.

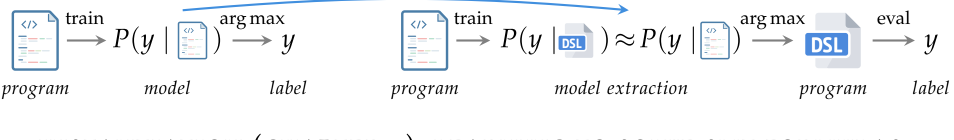
When *learning programs*, we follow this approach and incorporate the lattice of abstract states as part of the learned programs (although the lattice has to be provided manually and is not learned). When *learning over programs*, we augment the standard neural model with an option to abstain from making a prediction when uncertain. This corresponds to a flat lattice that either predicts a concrete label or returns the top element (i.e., abstains).

COMPOSITIONALITY Another high-level concept is compositionality. Currently, there is a conceptual gap between the design of static analysis tools that typically perform a computation until a fixpoint is reached, and neural networks that make predictions in constant time by passing the input program through a neural network (even though the network can be very large). Another consequence of the existing neural network architectures is that they make all the predictions at the same time. For example, in type prediction, the types of variable usages, as well as all the expressions are predicted in a single forward pass. As a result, the models tend to learn brittle statistical regularities around the predicted position rather than conditioning on the causal features (e.g., type of the previous usage).

To address this issue, we incorporate the fixpoint computation when *learning programs* using PHOG. When *learning over programs* using neural networks, we approximate the fixpoint computation by training a sequence of models that: *(i)* learns to predict labels of increasing complexity, and *(ii)* are compositional – we allow each model to explicitly condition on results computed by all prior models.

PROGRAM TRANSFORMATIONS Finally, the set of all valid program transformations $\Delta(x)$ needs to be defined. The set $\Delta(x)$ can be seen as the user provided specification that defines scenarios for which the model is expected to be robust. As such, it is therefore independent of the underlying model and only depends on the given task at hand. In our work, we use a wide range of semantic preserving (e.g., renaming variables, adding dead code, etc.), as well as label preserving transformations (e.g., changing constants, adding method parameters, etc.).

LEARNING OVER PROGRAMS LEARNING PROGRAMS



INSTANTIATION (CHAPTER 5): LEARNING TO SOLVE SMT FORMULAS

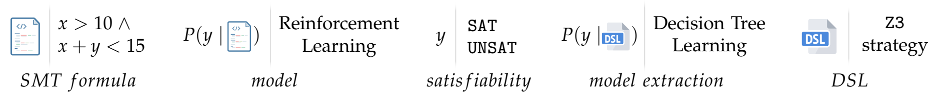


FIGURE 1.4: Illustration of combining *learning over programs* and *learning programs*. We first train a neural model to solve the task and then extract the model as an interpretable program. We instantiate the approach to learning to solve SMT formulas by *(i)* using reinforcement learning to select strategies that are efficient at solving SMT formulas, and *(ii)* synthesizing a Z3 strategy program that extracts the learned model and can be directly plugged into the Z3 solver.

1.3.3 *Learning Strategies to Solve SMT Formulas*

So far, we have considered a number of tasks over code and addressed them from two different perspectives – *learning over programs* and *learning programs*. Even though both approaches share many common concepts and techniques, we used them independently of each other. A natural next step is to take the advantages of both approaches and combine them to solve the same task.

One way how both approaches can be combined is by training a neural model to solve the task and then extracting its interpretable version, represented as a program in a domain-specific language. This is useful whenever neural networks are more efficient at solving the task at hand or easier to apply, yet the goal is to learn an interpretable program that can be integrated into an existing system.

Furthermore, using neural networks is especially useful in a challenging setting where the labelled dataset does not exist. That is, when the input is a set of programs without the corresponding ground-truth labels, but with an oracle that can check whether the predicted labels are consistent (or even correct). As an example, consider the task of fixing bugs where it is not known what the correct fix is, but it is possible to run a test suite to check whether the proposed fix is correct. Finding the correct fix is a search problem in a huge (or even infinite) search space, which is a setting where neural networks can be very effective.

In our work, we instantiate this setting to the task of learning strategies to solve SMT formulas and develop a tool called `FastSMT`. The instantiation is shown in Figure 1.4, where each program corresponds to a first-order logic formula (e.g., $\exists x, y \in \mathbb{N}\colon x > 10 \land x + y < 15$) and the goal of the SMT solver [66, 107] is to decide whether the formula is satisfiable or not (in this case the formula is satisfiable and one valid model is $x = 11$ and $y = 3$).

We phrase the challenge of solving SMT formulas as a tree search problem where at each step a satisfiability preserving transformation is applied to the input formula until the formula is solved. Our approach works in two phases: *(i)* given a dataset of unsolved formulas we use reinforcement learning to learn a policy that for each formula selects a suitable transformation to apply at each step in order to solve the formula, and *(ii)* we synthesize a strategy in the form of a loop-free program with branches. This strategy is an interpretable representation of the policy decisions and once learned, it is used to guide the SMT solver to decide formula satisfiability more efficiently, without requiring any modification to the solver itself and without needing to evaluate the learned policy at inference time.

1.3.4 *From Images to Layouts*

Our last tool, called `InferUI`, addresses the task of synthesizing robust relational layouts from examples. Concretely, the task of layout synthesis can be decomposed into the following three steps: *(i)* given an image of an application design, identify the set of components (e.g., buttons, text views, etc.) and their locations on the screen, *(ii)* synthesize a relational layout which when rendered, places the components at that same location, and *(iii)* infer the implementation of each component which when rendered, looks visually the same as the input image.

We illustrate the three steps in Figure 1.5. Here, the first step identifies that the input image contains three buttons and one background image. The second step synthesizes a relational layout for the Android platform – a program that can be compiled and rendered. The third step then infers the attributes of each component such as the background color, shadows, fonts, and more.

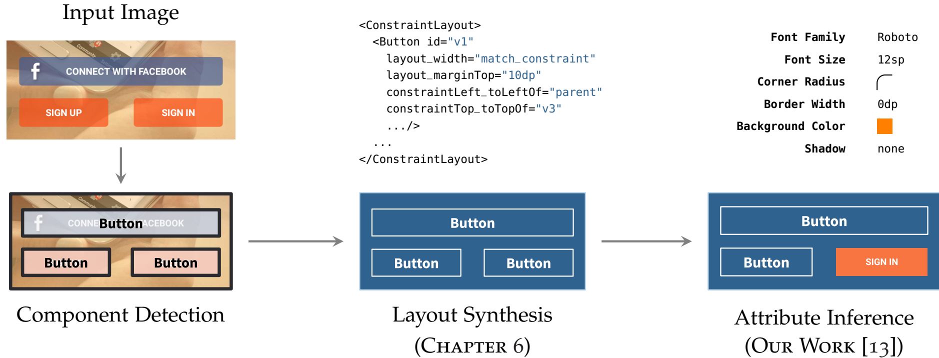


Figure 1.5: Illustration of the three steps used to synthesize relational layouts. First, all the components and their location are identified (either by the user or by training an object detection model [108–111]). Second, we synthesize a relational layout which, when rendered, places the components at their respective positions. Third, we learn to infer visual attributes for each components which, when rendered, look visually the same as in the input image.

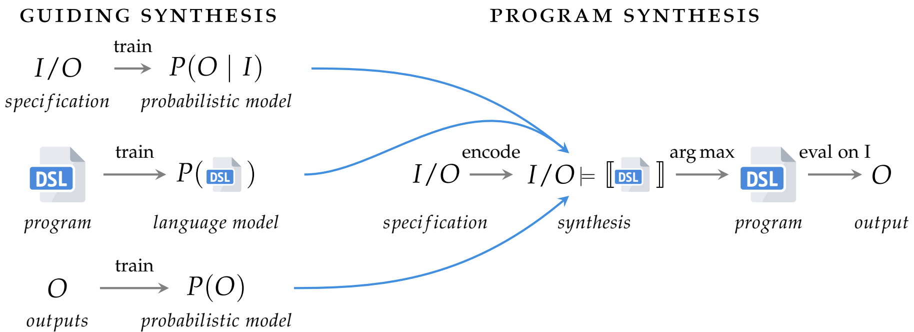


Figure 1.6: Illustration of combining machine learning and program synthesis for the task of synthesizing robust relational layouts presented in Chapter 6.

The most relevant step for this dissertation is the second step which synthesizes programs given an input specification. This is because it can be phrased as a program synthesis task, in line with the *learning programs* perspective. Similar to other synthesis tasks, this is a highly challenging problem since the input specification is severely underspecified – it contains a single input-output example containing view positions for one device, yet the synthesized program is expected to generalize across a range of devices with various resolutions and screen sizes. To make the synthesis practical and applicable to real-world layouts, we combine it with *learning over programs* in three key areas, as illustrated in Figure 1.6.

SCALABILITY We learn a probabilistic language model to guide the symbolic search used by program synthesis. This is critical for making the synthesis scalable to complex layouts, as it improves the runtime by up to 100×.

GENERALIZATION Because the input specification is underspecified, there is typically a large number of programs that satisfy it. However, the majority of those programs are incorrect and rejected by the users. To address this issue, we learn different probabilistic models and use them to guide the synthesis towards selecting better programs. These probabilistic models are learned not only over programs, but also over program outputs or the input-output specification.

ROBUSTNESS To further improve generalization, it is possible to extend the specification with hand-crafted robustness properties that capture properties of good layouts. In our work, we show how such robustness constraints can be formalized and encoded as part of the synthesis problem. Going one step further, in our subsequent work [7], we have shown that it is possible to learn suitable machine learning models that replace such robustness properties. This saves not only the time required to design and implement the robustness properties by hand, but also leads to synthesizing layouts with improved generalization.

1.4 SUMMARY

In this section, we have provided an overview of the topics presented in this thesis, some of the associated challenges, our contributions, as well as a brief introduction of the techniques and tools developed as part of our work. The main focus of our work is learning from large and small codebases using two different perspectives – a machine learning perspective that *learns over programs* and a programming languages (or synthesis) perspective that *learns programs*. Throughout this thesis, we will see how these perspectives can be used independently of each other to solve the same task and, more crucially, how they can be combined together in different ways.

2

PHOG: PROBABILISTIC MODEL FOR CODE

In this chapter, we introduce a new approach for learning probabilistic language models and show that our approach is a good fit for both, program source code and character level language modelling. We designed our model to be extremely fast at inference time and generic – it is readily applicable to any programming language (given a parser) and is trained to predict all program elements in the given language (e.g., variables, constants, method invocations, parameters, etc.). As such, our model can be used as a core building block by a variety of tools that learn from large codebases, including language translation [112, 113], patch generation [67], probabilistic type inference [22], or code completion [62, 64, 114].

probabilistic language model More formally, a probabilistic language model can be represented by the conditional probability of the next word given all the previous words:

$$P(\boldsymbol{w}) = \prod_{t=1}^{|\boldsymbol{w}|} P(w_t \mid \boldsymbol{w}_{<t}) \tag{2.1}$$

where w_t is t-th word and $\boldsymbol{w}_{<t} = (w_1, w_2, \ldots, w_{t-1})$ is a sequence of words preceding w_t (exclusive). Depending on the application, the words can correspond to a tokenized version of the program, nodes in the corresponding abstract syntax tree or other suitable program representation. For character level modelling, the words naturally correspond to characters in the input sentence.

conditioning context Finding the right program representation over which a probabilistic model is learned is key to the overall model precision. In particular, it is critical that the representation contains parts of the input relevant for the prediction while ignoring the rest. We can model the suitable representation by replacing $\boldsymbol{w}_{<t}$ in Equation 2.1 with a function $f(\boldsymbol{w}_{<t})$, which takes as input all the preceding words $\boldsymbol{w}_{<t}$:

$$P(\boldsymbol{w}) = \prod_{t=1}^{|\boldsymbol{w}|} P(w_t \mid f(\boldsymbol{w}_{<t})) \tag{2.2}$$

The generalized formulation in Equation 2.2 is useful as the function $f(\boldsymbol{w}_{<t})$ explicitly defines the *conditioning context* used as input to the probabilistic model. In our work, we consider the case where the code is written sequentially and the model is only allowed to look at the preceding words. However, the formulation in Equation 2.2 can be easily extended by including both preceding words as well as subsequent words, i.e., $f(\boldsymbol{w}_{<t}, \boldsymbol{w}_{>t})$.

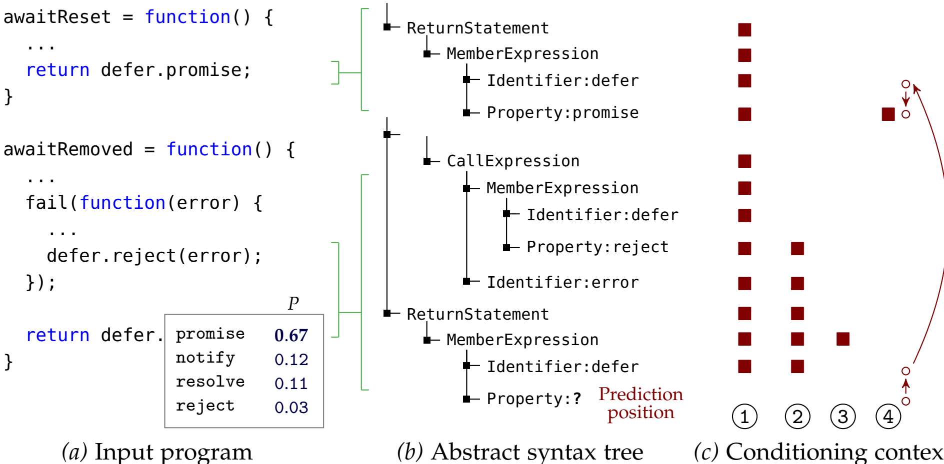


FIGURE 2.1: Example of a partial JavaScript code *(a)* and the corresponding abstract syntax tree *(b)*. To predict the property name, the probabilistic model for code can be conditioning on different contexts, illustrated as boxes ■ in *(c)*. The existing models condition either on the full input ① (e.g., neural networks) or on a fixed partial input ② (e.g., n-gram, log-bilinear models), ③ (e.g., PCFG). In contrast, we propose a model that conditions on context obtain dynamically ④ using a learned program that traverses the input and accumulates values.

To illustrate different *conditioning contexts*, consider the JavaScript code fragment shown in Figure 2.1 *(a)*. The code is parsed by a *context-free grammar* (CFG) in order to obtain an *abstract syntax tree* (AST), shown in Figure 2.1 *(b)*. Now, consider the last statement which returns a property of the `defer` object. The JavaScript CFG permits any valid identifier here, yet not every identifier would produce a desirable program. In a box akin to a code completion window, we show some reasonable values for which property should be returned along with their probabilities. To compute these probabilities, the probabilistic language model can be instantiated with different *conditioning context* $f(\boldsymbol{w}_{<t})$, as shown in Figure 2.1 *(c)*:

- *Conditioning on fixed partial input* $f(\boldsymbol{w}_{<t}) = \{\phi_i(\boldsymbol{w}_{<t})\}_{i=1}^{k}$. Traditionally, language models reduced the difficulty of the modelling problem by transforming the input into a fixed set of features $\phi_i(\boldsymbol{w}_{<t})$. The models that use this approach include n-gram models, log-bilinear models and probabilistic context free grammars.
  - *N-gram model* takes advantage of the observation that temporally closer words are statistically more dependent. Concretely, the n-gram model makes an independence assumption that each word depends only on the last $n-1$ words, thus $f(\boldsymbol{w}_{<t}) = (w_{t-n-1}, w_{t-n}, \cdots, w_{t-1})$.

  - *Log-bilinear model* defines the conditioning context as a linear combination of the features[1] $\phi_1(\boldsymbol{w}_{<t}), \cdots, \phi_k(\boldsymbol{w}_{<t})$, which often correspond to n-grams (i.e., $\phi_i(\boldsymbol{w}_{<t}) = w_i$ for unigram or $\phi_i(\boldsymbol{w}_{<t}) = (w_i, w_{i+1})$ for bigram), but can also capture user-defined domain knowledge such as cyclomatic complexity.
  - *Probabilistic context free grammar (PCFG)* is an extreme case of conditioning on a fixed partial input, where each prediction is conditioned only on the parent non-terminal node $f(\boldsymbol{w}_{<t}) = parent(w_t)$. While conditioning only on the parent is often sufficient for parsing, it is very limiting when it comes to language modelling.

  The main limitation of these models is that the context is *fixed* and identical for all prediction types. As shown in Figure 2.1 *(c)*, PCFG is an extreme case where the prediction is made only based on the fact that it is an object `Property`, without taking into account any information about the object itself. This naturally leads to poor probability estimates and suggestions that generate undesirable programs.

- *Conditioning on full input $f(\boldsymbol{w}_{<t}) = \boldsymbol{w}_{<t}$.* As the probabilistic language models became more powerful, they started conditioning on the full input instead of manually defining a fixed set of features upfront. To achieve this, the state-of-the-art and most widely used models are based on deep learning.

  - *Neural networks* convert each input word into a real-valued vector and then learn a smooth function that maps the vectors into the desired output value – in this case a summary of the context relevant to make a prediction. Over the years, many different architectures were proposed to learn the mapping efficiently, including those that treat the input as a sequence [95], tree [115], graphs [43] or even as a bag of words [116].

  Because the model conditions on the full input, which for programs can be hundreds of lines of code, it has to be powerful enough to: *(i)* discover and condition on features that are strongly correlated with the prediction, rather than *(ii)* conditioning on the much more common but weakly correlated features. This is especially important for training robust models, a topic we cover in Chapter 4.

- *[Our Work] Conditioning on dynamic sparse input $f(\boldsymbol{w}_{<t}) = p(\boldsymbol{w}_{<t})$.* In our work, we develop a new probabilistic language model which learns to dynamically select a sparse subset of the input relevant to the prediction. The model is *dynamic* because the conditioning context is computed based on the current prediction and not fixed a priori as done for n-gram or log-bilinear models. Further, it is *sparse*, meaning that only a few input words are selected, rather than conditioning of the full input as done in neural networks.

1 The main difference between n-gram and log-linear models is that while n-grams use discrete word representation, log-linear models use continuous representation by converting each word (or a feature) into a real-valued vector. As a result, log-linear models are in fact a simple neural language model.

– *Probabilistic higher order grammar (PHOG)* proposes a novel approach to building probabilistic models – by parametrizing the model with a learned program from a domain-specific language. That is, the conditioning context is obtained by *executing* the learned program p. Upon termination, p returns a set of input words that are relevant for the given prediction. In Figure 2.1 *(c)*, we show a conditioning that predicts an object property by finding a place in the code where it was used in the same context (a return statement with the same variable), and then including the property used there as conditioning context (arrows ↗ show movement over the AST that leads to computing this context).

The optimization problem addressed during the training is to search for the program in a domain-specific language, which given an input (in our case an AST), returns a conditioning context that maximizes the probability of the dataset. As we will show in Section 2.1, we designed the domain-specific language to be independent of the concrete programming language, making our approach widely applicable.

PHOG is a novel approach to building probabilistic models – by parametrizing the model with a learned program from a domain-specific language.

The design choice of using programs to parametrize the probabilistic model has a number of advantages. First, programs are interpretable and can be inspected by a developer. Second, programs are a natural way to incorporate inductive bias, especially for domains that operate over structured data. Finally, the domain-specific language that defines the set of valid programs is easily extensible and we designed it to include prior works that use fixed conditioning context.

THE NEED FOR DYNAMIC CONDITIONING In Figure 2.1 *(c)*, we illustrated how to compute conditioning context dynamically for a given type of prediction, in this case, an object property. However, to learn accurate probabilistic models, it is crucial that the conditioning depends not only on the prediction type, but also on the context in which the prediction is made.

As a concrete example, consider the four JavaScript code examples shown in Figure 2.2. The goal for each of these examples is to predict the most likely HTTP header property that should be set when performing a HTTP request (marked as ?). An important observation is that even though all four examples predict a property name, the best features needed to make the correct prediction are very different in each example (marked as green boxes). In Figure 2.2 *(a)* and *(b)*, the only available information is the variable name to which the dictionary object is assigned, and the fact that it is used in the `http.request` API, respectively. In Figure 2.2 *(c)*, a prediction can be based on the previous two properties that were already set, as well as on the fact that the dictionary is used as a second argument in the `http.request` API. In Figure 2.2 *(d)*, another similar call to `http.request` is present in the same program and a probabilistic model may leverage that information.

relevant positions used to condition the prediction

(a)
```
var http_options = {
  ? ← prediction
};      position
```

(b)
```
http.request(..., {
  ?
});
```

(c)
```
http.request(..., {
  host: 'localhost',
  accept: '*/*',
  ?
});
```

(d)
```
http.request(..., {
  host: 'www.google.com',
  accept: '*/*',
  user-agent: 'curl/7.2'
});
...
http.request(..., {
  host: 'www.bing.com',
  accept: '*/*',
  ?
});
```

FIGURE 2.2: JavaScript code snippets used to motivate the need for a probabilistic model with *dynamically* computed context based on the input. Although each example predicts a property name used for an HTTP request, the best context used to condition the prediction (shown as rectangles) is different for each input.

Intuitively, the best features depend on the *context* in which the completion is performed. Thus, ideally, we would like our probabilistic model to automatically discover the relevant features/context for each case.

To achieve this, recall than the key idea behind our approach is to parametrize a probabilistic model by a learned program p. With this in mind, a natural way to allow the program p to compute different conditioning contexts is to extend p with branch statements. Concretely, our approach can learn a program such as the one shown in Figure 2.3 directly from the training data. Here, the program contains several tests that check properties of the input code snippet (e.g., whether some properties have been set) in order to select the suitable conditioning context used for making the prediction. In this example, each of the four inputs from Figure 2.2 will end up in a different branch, allowing the model to learn a suitable context specialized for making each of the predictions.

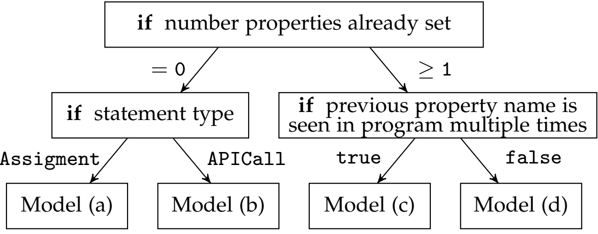


FIGURE 2.3: An example of a program, visualized as a decision tree, learned by our approach. When processed by this program, each of the inputs in Figure 2.2 will end up in a different path (tree leaf) and thus a different model will be used to answer the queries of each example. As a result, each of the models can learn conditioning context specialized for the types of predictions it makes.

The main challenge we need to address is how to learn such complex programs with branches efficiently. The key technical insight is to (recursively) split the training data in a fashion similar to decision trees [117] and to then learn smaller specialized programs for each branch of the tree. Importantly, our formulation allows us to cleanly represent and experiment with classic decision tree learning algorithms such as ID3 [118] but also permits new, previously unexplored extensions which lead to better precision than ID3.

OUTLINE We organize this chapter as follows. In Section 2.1 we introduce our domain-specific language, called `CondGen`, for obtaining the conditioning context. In Section 2.2 we define how we can use a program $p \in$ `CondGen` to parametrize a probabilistic language model. Next, in Section 2.3 we present our approach for learning programs in `CondGen`. We then provide a thorough experimental evaluation of our approach, instantiated for a number of datasets and two domains: *(i)* learning probabilistic model of code in Section 2.4 and character level language modelling in Section 2.5. Finally, we describe the related work in Section 2.6 and provide a brief summary and discussion in Section 2.7.

2.1 PARAMETRIZING LANGUAGE MODELS WITH PROGRAMS

We start by defining a domain specific language, called `CondGen`, for expressing programs that describe the conditioning context. At a high level, executing a program $p \in$ `CondGen` on an input $\boldsymbol{w}_{<t}$ returns a conditioning context $ctx \in Context$ based on which, the probabilistic model makes a prediction $P(w_t \mid p(\boldsymbol{w}_{<t}))$. For example, the conditioning context computed by program p for the input shown in Figure 2.2 *(a)* would consist of a single value `http_options`. However, the same program p executed on a different input in Figure 2.2 *(c)* would lead to a larger conditioning context that contains the object `http`, the method call `request` as well as two object properties `host` and `accept`.

To keep our approach widely applicable, the `CondGen` language presented next is not only independent of any programming language but also applicable beyond programs to tasks over text. In what follows we will:

- Describe the basic building blocks of `CondGen`, which consists of programs that traverse input and accumulate context with values from the input.
- Show how to instantiate these programs to language modelling over programs as well as character level language modelling.
- Describe the syntax and intuition behind the full `CondGen` language, which extends the basic programs with branches and state.
- Formally define the small-step semantics of the `CondGen` language.

2.1.1 *Simple Programs*

The SimpleProgram is a basic building block of the CondGen language and it has the following syntax:

$$\texttt{SimpleProgram} ::= \epsilon \mid \texttt{Move; SimpleProgram} \mid \texttt{Write; SimpleProgram}$$

The SimpleProgram describes a loop-free and branch-free program that accumulates context with values from the input by means of traversing within the input (using Move instructions) and writing the observed values (using Write instructions). The result of executing a SimpleProgram is accumulated context, which is used either to condition the prediction, to determine which program to execute next (described in Section 2.1.2.1) or to update the program state (described in Section 2.1.2.2). Note that the syntax allows empty programs (denoted at ϵ), in which case the accumulated context is empty.

One of the advantages of our approach is that the Move and Write instructions can be specialized depending on the target application. For example, it is straightforward to add instructions that implement sophisticated static analysis or are specific to a concrete programming language. We will demonstrate how this is useful in Chapter 3, which extends the language with instructions specifically designed for learning static analyzers. However, in this chapter, we keep the language generic and instantiate it for two domains – for programs and for natural language.

2.1.1.1 *Instantiation for Probabilistic Models of Code*

When learning probabilistic models of code, we represent the input programs using their corresponding abstract syntax trees (AST). Each AST node contains two attributes – the type of the node and an optional value. As an example, consider the AST node Property : promise from Figure 2.1, where Property denotes the type and promise is the value. The number of unique types is relatively small (e.g., 44 for our JavaScript corpus) and is determined by the non-terminal symbols in the underlying context-free grammar that defines the AST. The number of values is very large (10^9 in our JavaScript corpus) and is a mixture of identifiers, literals, constants and language specified operators (e.g., i, load, 7, "get", $+$, $*$).

MOVE INSTRUCTIONS For programs, we instantiate Move instructions as shown in Figure 2.4. Here, the Move instructions facilitate tree traversal by moving the current position in the tree to the parent node (UP), left and right siblings (LEFT, RIGHT), first and last child (DOWN_FIRST, DOWN_LAST), previous and next node in depth first search traversal order (PREV_DFS, NEXT_DFS), previous and next leaf[2] node (PREV_LEAF, NEXT_LEAF), previous node that has the same type or value (PREV_NODE_TYPE, PREV_NODE_VALUE) and finally to the previous node where the parent and grandparent have the same type and values (PREV_NODE_CONTEXT). We

2 Leaf nodes are those nodes in the AST that contain a non-empty value. For example, Property : promise is a leaf node but CallExpression and ReturnStatement are not.

▷ Instructions to traverse over abstract syntax trees

Move ::= LEFT | RIGHT | UP | DOWN_FIRST | DOWN_LAST | PREV_DFS | NEXT_DFS |
PREV_LEAF | NEXT_LEAF | PREV_TYPE | PREV_VALUE | PREV_CONTEXT

Write ::= WRITE_VALUE | WRITE_TYPE | WRITE_POS

FIGURE 2.4: Instantiation of the Move and Write instructions for learning probabilistic language models of code.

note that not all the instructions are defined symmetrically. For example, we defined PRED_NODE_VALUE but not NEXT_NODE_VALUE. This is because the corresponding instructions have limited usefulness since in our work, the predicted position w_t is the last word in the input (i.e., the code is written sequentially).

WRITE INSTRUCTIONS The write instructions WRITE_TYPE, WRITE_VALUE and WRITE_POS append facts about the currently visited node to the accumulated context by writing the type, value and position of the node in the parent respectively.

Example Consider the following example of a SimpleProgram:

LEFT, WRITE_VALUE, UP, WRITE_POS, UP, DOWN_FIRST, DOWN_LAST, WRITE_VALUE
(Figure 2.5)

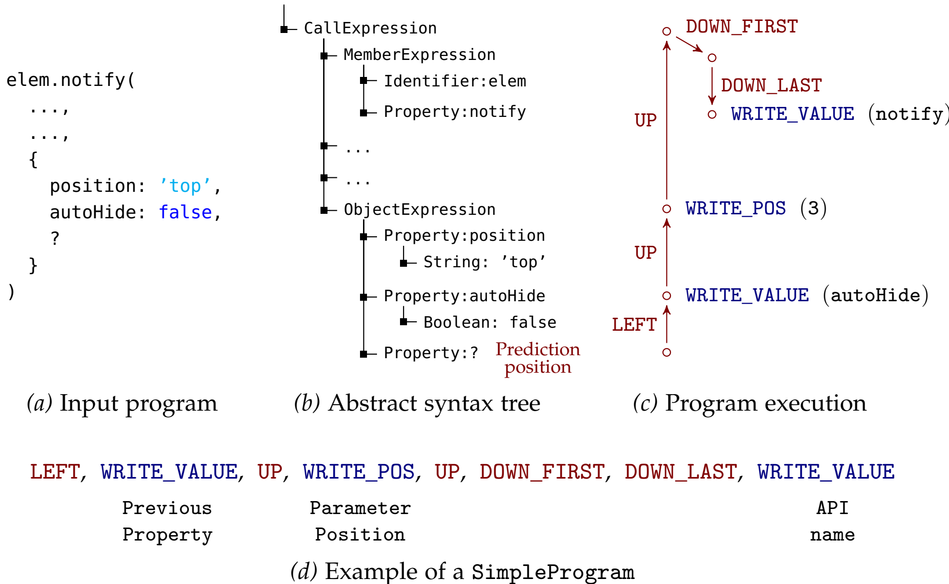
Initially, the program execution (shown in Figure 2.5) starts at the position where the prediction is made, in this case at the tree node Property :?. The first program instruction LEFT moves the current position to the left sibling, which corresponds to the node Property : autoHide. As the previous property of the same object is relevant to the prediction, the program executes WRITE_VALUE to store the value of the property (autoHide). Next, the program uses UP to traverse to the parent node and WRITE_POS to store the position of the node in the parent. In this case, this corresponds to conditioning on the fact that the object expression is a third argument. Next, the program executes a sequence of Move instructions that traverse to a node corresponding to the method name being executed and stores its value using WRITE_VALUE. The result of executing this program is a conditioning context containing tree values – autoHide, 3 and notify.

The key advantage of our definition of the Move and Write instructions is that the language is small, yet expressive enough to describe interesting programs. For example, we can express the program illustrated in Figure 2.1 as follows:

LEFT, PREV_NODE_CONTEXT, RIGHT, WRITE_VALUE
(Figure 2.1)

Similarly, we can directly encode a number of prior works that define fixed condition context. For example, we can express the 3-gram model as follows:

PREV_DFS, WRITE_VALUE, PREV_DFS, WRITE_VALUE
(3-gram model)



(a) Input program *(b)* Abstract syntax tree *(c)* Program execution

(d) Example of a `SimpleProgram`

FIGURE 2.5: Example of a `SimpleProgram` *(d)* and its execution *(c)*. The program uses `Move` instructions to traverse the AST *(b)* representation of the JavaScript code snippet *(a)*. The result of executing the program is the conditioning context, which in this case corresponds to the previous property (`autoHide`), the parameter position (third parameter) and the name of the method called (`notify`).

2.1.1.2 *Instantiation for Character Level Language Modelling*

We show our instantiation of the `Move` and `Write` instructions for character level language modelling in Figure 2.6. The intuition behind both is the same as for code, except that they operate over sequence of characters instead of trees.

MOVE INSTRUCTIONS We define four types of `Move` instructions – `LEFT` and `RIGHT` that move to the previous and next character, respectively, `PREV_CHAR` that moves to the most recent position in the input with the same value as the current character w_i and `PREV_POS` which works as `PREV_CHAR` but only considers positions in the input that are partitioned into the same language model[3]. Further, for each character c in the input vocabulary we generate instruction `PREV_CHAR(c)` that traverses to the most recent position of character c.

WRITE INSTRUCTIONS We define three `Write` instructions – `WRITE_CHAR` that writes the value of the character at the current position, `WRITE_HASH` that writes a hash of all the values seen between the current position and the position of last

3 If there is only a single language model, then `PREV_CHAR` and `PREV_POS` are equivalent. We describe how we partition the input and learn multiple language models in Section 2.2

```
▷ Instructions to traverse over sequence of characters
Move ::= LEFT | RIGHT | PREV_CHAR | PREV_CHAR(c) | PREV_POS
Write ::= WRITE_CHAR | WORD_HASH | WORD_DIST
```

FIGURE 2.6: Instantiation of the `Move` and `Write` instructions from Figure 2.8 for character level language modeling.

`Write` instruction, and `WRITE_DIST` that writes a distance (i.e., number of characters) between the current position and the position of the last `Write`. In our implementation we truncate `WRITE_HASH` and `WRITE_DIST` to a maximum size of 16.

Example With `Write` and `Move` instructions, we can express various programs that extract useful context for a given position in the input text, as illustrated in Figure 2.7. For example, a program that computes the length of the current word can be written as:

`PREV_CHAR(␣), WRITE_DIST`
(current word length)

Another useful program is:

`LEFT, PREV_CHAR, RIGHT, WRITE_CHAR`
(previous occurrence of the character on the left)

Here, `LEFT` moves to the prior character, `PREV_CHAR` to the previous occurrence of the prior character and `RIGHT` moves to the character right after. This sequence of instructions is very similar to instructions `LEFT`, `PREV_NODE_CONTEXT`, `RIGHT` used earlier for modelling code.

(a) Input text

(b) Program execution *(c)* Program

FIGURE 2.7: Examples of `SimplePrograms` used for character level language modelling. The first program computes the current word length by jumping to the preceding space via `PREV_CHAR(␣)` instruction. Executing the second program results in a conditioning context containing the prior character (`M`) and the most recent character that follows it (g).

2.1.2 `CondGen`: *A Domain Specific Language for Describing Conditioning Context*

Having described the core building block of `CondGen`, we now introduce the syntax of the full `CondGen`, shown in Figure 2.8. At a high level, `CondGen` consists of four types of programs that can be split into two categories:

- *Programs that accumulate context*, `SimpleProgram` and `EqualityProgram`, which traverse the input with the goal of accumulating a set of relevant values.
- *Programs with branches and state*, `BranchProgram` and `StateProgram`, which are used to dynamically select which programs to execute next.

```
CondGen ::= SimpleProgram | EqualityProgram |
            BranchProgram | StateProgram

▷ Program that traverses over the input and acculumates conditioning context
SimpleProgram ::= ε |
                  Move; SimpleProgram |
                  Write; SimpleProgram

▷ Program that enables out-of-vocabulary words prediction
EqualityProgram ::= (SimpleProgram, SimpleProgram)

▷ Conditionally selects next program to execute by maching the context
accumulated by a SimpleProgram against a set of constant values
BranchProgram ::=
  switch SimpleProgram
   case Constant or ··· or Constant then CondGen
   ···
   case Constant or ··· or Constant then CondGen
   default CondGen

▷ Updates the current state and selects next program to execute based on the state
StateProgram ::= StateUpdate; StateSwitch
StateUpdate ::=                        StateSwitch ::=
  switch SimpleProgram                   switch state
   case Constant then INC                 case Constant then CondGen
   case Constant then DEC                 ···
   default SKIP                           case Constant then CondGen
                                          default CondGen
```

FIGURE 2.8: Syntax of the `CondGen` language for character level language modelling.

2.1.2.1 *Branch Programs*

As we have seen so far, `SimplePrograms` are expressive enough to describe relevant conditioning context specialized for different types of predictions and their context. Our next goal is to provide a mechanism to compose and select which program to execute, such that the resulting program works well for the entire dataset.

To achieve this, we introduce `BranchProgram` that conditionally selects appropriate subprograms based on a learned condition. The condition is represented as another `SimpleProgram` that accumulates values and executes the program associated with the matching case clause. If no clause matches, then a default program is executed. Note that the `BranchProgram` is similar to the switch statement commonly found in imperative programming languages. The main difference is that in our work, only one matching case clause is executed[4] and we support disjunction of values for each case condition. Further, we allow executing a subprogram only if the conjunction of the conditions holds. This is possible because the programs in each case clause can be any `CondGen`, meaning that `BranchPrograms` can be nested.

During the learning (described in Section 2.3), the goal is to synthesize all the `BranchProgram` components – the `SimpleProgram` used as condition, the number of case clauses, the constant values $c_1, \ldots, c_n$ used to match each case clause and the `CondGen` programs inside of each case clause (including the default program).

Example We illustrate `BranchPrograms` on a simple example shown in Figure 2.9. Here, the `BranchProgram` condition is `LEFT WRITE_CHAR`, which corresponds to returning the previous character. For example, when the partial input is `a1a`, executing `LEFT WRITE_CHAR` will return `a`. Because this matches the case clause `'a'` **or** `'b'`, the execution will continue with the subprogram f_1. For the input `a1a2`, the condition returns `2`, which does not match any case clause. In this case, the default program f_2 is executed.

(a) Input text

`a1a2b1`

(b) Program

```
switch LEFT WRITE_CHAR
    case 'a' or 'b' then f1
    default f2
```

t	$w_{<t}$	w_t	`LEFT WRITE_CHAR`	subprogram
1		a	ϵ	f_2
2	a	1	a	f_1
3	a1	a	1	f_2
4	a1a	2	a	f_1
5	a1a2	b	2	f_2
6	a1a2b	1	b	f_1

FIGURE 2.9: Example of a `BranchProgram` executed on the input `a1a2b1`. The table shows the current character that has to be predicted (w_t), the partial input ($w_{<t}$), accumulated context by the condition `LEFT WRITE_CHAR`, and subprogram selected based on matching the accumulated context against the case clauses.

4 If multiple case clauses match the condition, then the first one is executed.

2.1.2.2 *State Programs*

A common difficulty in building statistical language models is capturing long range dependencies in the given dataset. Our CondGen language partially addresses this issue by using Move instructions that can jump to various positions in the input (e.g., using instructions such as PREV_CHAR, PREV_NODE_CONTEXT, PREV_NODE_VALUE, etc.). However, we can further improve this by explicitly introducing a state to our programs using StateProgram.

The StateProgram consists of two sequential operations – updating the current state and determining which program to execute next based on the value of the current state. For both, we reuse the switch construct defined previously for BranchPrograms. In our work, we consider integer valued state that can be either incremented (INC), decremented (DEC) or left unmodified (SKIP) after processing each input word. We note that other definitions of the state, such as stack based state, are possible.

Example As an example of a StateProgram, consider the question of detecting whether the current character is inside a comment or is part of the source code. These denote very different types of data and to achieve high model precision, the probabilistic model should produce predictions tailored for both of them. This can be achieved by using a simple state program shown in Figure 2.10. The StateUpdate condition is LEFT WRITE_CHAR LEFT WRITE_CHAR, which accumulates the last two characters which were written. The corresponding case clauses increment the state on '/*', decrement the state on '*/' and leave the state unchanged otherwise. As a result, the StateSwitch selects subprogram f_1 when predicting code (when $state = 0$) and subprogram f_2 when predicting comments.

```
StateSwitch ::=
  switch state
    case 0 then f1
    default f2

StateUpdate ::=
  switch LEFT WRITE_CHAR
         LEFT WRITE_CHAR
    case '/*' then INC
    case '*/' then DEC
    default SKIP
```

(a) StateProgram

```
state → 0          ...
state → 1          /* If these interrupts use reservation mode,
program f2          * clear the activated bit so request_irq()
                    * will assign the final vector.
                    */
                   if (can_reserve) {
state → 0            for_each_msi_entry(desc, dev) {
program f1                  irq_data = irq_domain_get_irq_data(
                                           domain, desc->irq);
                            irqd_clr_activated(irq_data);
                     }
                   }
```

(b) Code snippet

FIGURE 2.10: Illustration of a SwitchProgram executed on a code snippet from *Linux Kernel*. The SwitchProgram is designed to keep track of comments, by incrementing the state on '/*' and decrementing on '*/'.

2.1.2.3 *Equality Programs*

A common limitation of probabilistic models is that they cannot predict values (i.e., names of method calls, variable names, etc.) not seen in the training data. We offer simple mitigation for this limitation by introducing `EqualityPrograms`, which can express the equality of the output label to a value already present in the input. Technically, this is achieved by learning a pair of `SimplePrograms`. The semantics of the first program are as before – to accumulate context used to condition the prediction. The second program also accumulates context, but with a different purpose – the accumulated context contains concrete values from the input that might be used as the prediction. We will describe a simple way to extend any probabilistic model with the ability to predict such values in Section 2.2.1.

2.1.3 *Small-step Semantics of* `CondGen`

We formally define `CondGen` programs to operate on a state:

$$\sigma = \langle x, i, ctx, counts \rangle \in States$$

The domain *States* is defined as $States = \mathbb{X} \times \mathbb{N} \times Context \times Counts$, where $x \in \mathbb{X}$ is the input, $i \in \mathbb{N}$ is a position in the input, $Context = \Sigma^*$ is a list of values accumulated by executing `Write` instructions and $Counts\colon$ `StateSwitch` $\rightarrow \mathbb{N}$ is a mapping that contains a value denoting the current count for each `StateSwitch` program. For code, the input $\mathbb{X}$ corresponds to abstract syntax trees, where each node has an associated type and value[5]. For natural language, $\mathbb{X} = char^*$ corresponds to a sequence of characters. In both cases, we use Σ to denote the range of values produced by executing any `Write` instruction. For code, $\Sigma = \mathbb{N} \cup T \cup V$ can be a natural number, node type (T) or node value (V). For natural language, $\Sigma = \mathbb{N} \cup char$ can be a natural number or an input character.

Initially, the execution of a program $p \in$ `CondGen` starts with an empty context $[] \in Context$ and counts initialized to zero (denoted as $\mathbf{0}$) for every `StateSwitch` program. For a program $p \in$ `CondGen`, an input $x \in \mathbb{X}$, and a position i in x, we say that p computes the context $ctx = p(x,i)$ iff there exists a sequence of transitions, according to the small-step semantics, from $\langle p, x, i, [], \mathbf{0} \rangle$ to $\langle \epsilon, x, _, ctx, _ \rangle$. That is, ctx is the accumulated context obtained by executing all the instructions of the program p on the input x starting at position i. To remove clutter, we will use $p(x)$ to denote executing program p from the last position in the input x.

2.1.3.1 *Semantics of* `SimpleProgram` *for Probabilistic Models of Code*

We formalize the small-step semantics of `Write` and `Move` instructions in Figure 2.11, using the rules [Write] and [Move], respectively.

5 We provide a description and example of abstract syntax trees in Section 2.1.1.1.

$$\frac{w \in \texttt{Write} \qquad v = wr(w, x, i)}{\langle w :: s, x, i, ctx, counts \rangle \to \langle s, x, i, ctx \cdot v, counts \rangle} \text{[WRITE]}$$

$$\frac{m \in \texttt{Move} \qquad i' = mv(m, x, i)}{\langle m :: s, x, i, ctx, counts \rangle \to \langle s, x, i', ctx, counts \rangle} \text{[MOVE]}$$

FIGURE 2.11: Small-step semantics of Move and Write instructions. Each rule is of the type: CondGen $\times$ *States* $\to$ CondGen $\times$ *States*.

WRITE INSTRUCTIONS The semantics of each Write instruction is that it accumulates a value v to the conditioning set ctx as defined by the function $v = wr(op, x, i)$. Here, $wr\colon \texttt{Write} \times \mathbb{X} \times \mathbb{N} \to \Sigma$ is a function that takes as input Write instruction, an input $x \in \mathbb{X}$ and position in the input $i \in \mathbb{N}$, and returns a value v appended to the context ctx. We define wr as follows:

- $wr(\texttt{WRITE_TYPE}, x, i)$ returns the type of the node at position i in the input x.
- $wr(\texttt{WRITE_VALUE}, x, i)$ returns the value of the node at position i in the input x. If the node does not contain a value, a special symbol denoting the empty value is returned.
- $wr(\texttt{WRITE_POS}, x, i)$, which for node at position i, returns 0 if it the first child of its parent, 1 if it is the second child, etc.

MOVE INSTRUCTIONS The semantics of each Move instruction is to traverse the input by changing the current position i to $i' = mv(op, x, i)$. Here, $mv\colon \texttt{Move} \times \mathbb{X} \times \mathbb{N} \to \mathbb{N}$ is a function that computes the new position, defined as follows:

- $mv(\texttt{UP}, x, i) = i'$, where i' is the parent node of the node at position i in x.
- $mv(\texttt{LEFT}, x, i) = i'$, where i' is the position of left sibling of the node at position i in x. Similarly, $mv(\texttt{RIGHT}, x, i)$ returns the right sibling's position.
- $mv(\texttt{DOWN_FIRST}, x, i) = i'$, where i' is the first child of the node at position i in x. Similarly, $mv(\texttt{DOWN_LAST}, x, i)$ returns the position of the last child.
- $mv(\texttt{PREV_DFS}, x, i) = i'$, where i' is the predecessor of the node at position i in x in the left-to-right depth-first search traversal order. Similarly, $mv(\texttt{NEXT_DFS}, x, i)$ returns the position of successor node in the left-to-right depth-first search traversal order.
- $mv(\texttt{PREV_LEAF}, x, i) = i'$, where i' is the first leaf on the left of the node at position i in x. Similarly, $mv(\texttt{NEXT_LEAF}, x, i)$ returns the first leaf on the right.
- $mv(\texttt{PREV_NODE_CONTEXT}, x, i) = i'$, where i' is the largest position for which it holds that: $i' < i$, the nodes at positions i and i' have the same value and type, and, the parents of both nodes have the same type.

- mv(PREV_NODE_VALUE, $x, i) = i'$, where i' is the largest position for which it holds that: $i' < i$, and the nodes at positions i and i' have the same value. The function mv(PREV_NODE_TYPE, x, i) is defined analogously but ensures that the nodes have the same type.

Further, for all the Move instructions defined above, it is possible that the condition for the new position i' cannot be satisfied, i.e., the execution of the instruction *fails*. For example, when executing the UP instruction on a node that has no parent or executing LEFT instruction on a node that has no left sibling. In cases where the Move instruction *fails*, the semantics are that the original position i is returned.

2.1.3.2 *Semantics of* SimpleProgram *for Character Level Language Modelling*

The small-step semantics of Write and Move instructions for character level language modelling are also defined in Figure 2.11, using rules [Write] and [Move], respectively. The only difference is how the functions mv and wr are defined.

WRITE INSTRUCTIONS We define the wr function as follows:

- wr(WRITE_CHAR, x, i) returns the character at position i in the input string x. If i is not within the bounds of x (i.e., $i < 1$ or $i \geq len(x)$) then ϵ is returned.
- wr(WRITE_HASH, x, i) returns the hash of all characters seen between the current position i and the position of the last Write instruction that was executed. More formally, let i_{prev} be the position of the previous write. Then $wr(\texttt{WRITE_HASH}, x, i) = H(x, i, min(i + 16, i_{prev}))$, where H: $char^* \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is a hashing function that hashes characters in the string from the given range of positions. The hashing function used in our implementation is:

$$H(x, i, j) = \begin{cases} x_i & \text{if } i = j \\ H(x, i, j-1) * 137 + x_j & \text{if } i < j \\ \epsilon & \text{otherwise} \end{cases}$$

- wr(WRITE_DIST, x, i) returns a distance, measured as the number of characters, between the current position and the position of latest Write instruction. In our implementation we limit the return value to be at most 16, i. e., $wr(\texttt{WRITE_DIST}, x, i) = min(16, |i - i_{prev}|)$.

MOVE INSTRUCTIONS We define the mv function as follows:

- $mv(\texttt{LEFT}, x, i) = max(1, i - 1)$, moves to the position of the previous character in the input. Similarly, $mv(\texttt{RIGHT}, x, i) = min(len(x) - 1, i + 1)$ moves to the position of the next character. Here, $len(x)$ denotes the length of x.
- $mv(\texttt{PREV_CHAR}, x, i) = i'$, where i' is the position of the most recent character with the same value as the character at position i, i. e., the maximal i' such that $x_{i'} = x_i$ and $i' < i$. If no such character is found, the value -1 is returned.

- $mv(\texttt{PREV_CHAR(c)}, x, i) = i'$, where i' is the position of the most recent character with the same value as character c, i. e., the maximal i' such that $x_{i'} = c$ and $i' < i$. If no such character c is found in x the value -1 is returned.
- $mv(\texttt{PREV_POS}, x, i) = i'$, where i' is the same as for PREV_CHAR, except that only positions that fall into the same leaf program are considered.

2.1.3.3 *Semantics of* BranchProgram

The semantics of BranchProgram are described by the [Switch] and [Switch-Default] rules shown in Figure 2.12. In both cases, the guard of the BranchProgram, denoted as $op.guard$, is executed to obtain the context ctx_{guard}. The context is then matched against all the case clauses in the syntactic order they were defined. If an exact match is found, denoted as $ctx_{guard} \in \boldsymbol{c}_j$, then the corresponding program is selected for execution (rule [Switch]). Note that the condition $\bigwedge_{1 \leq n < j} ctx_{guard} \notin \boldsymbol{c}_n$ ensures that if multiple case clauses match, only the first one is selected. If no match is found, the default program denoted as $op.default$ is selected for execution (rule [Switch-Default]). In both cases, the current context is updated by appending the index of the branch taken or a special symbol $\perp$ for the default case. This allows the model to distinguish which branch a program has taken, effectively splitting the datasets into multiple partitions, one for each branch.

```
BranchProgram ::=
  switch SimpleProgram
    case Constant or ··· or Constant then CondGen
    ...
    case Constant or ··· or Constant then CondGen
    default CondGen
```

(a) BranchProgram Syntax

```
BranchProgram ::=
  switch guard
    case c1 then f1
    ...
    case ck then fk
    default fdefault
```

(b) BranchProgram Notation

$$\frac{op \in \texttt{BranchProgram} \quad \langle op.guard, x, i, [], counts\rangle \rightarrow \langle \epsilon, x, _, ctx_{guard}, _\rangle \quad ctx_{guard} \in \boldsymbol{c}_j \quad \bigwedge_{1 \leq n < j} ctx_{guard} \notin \boldsymbol{c}_n}{\langle op, x, i, ctx, counts\rangle \rightarrow \langle op.f_j, x, i, ctx \cdot j, counts\rangle} \text{[Switch]}$$

$$\frac{op \in \texttt{BranchProgram} \quad \langle op.guard, x, i, [], counts\rangle \rightarrow \langle \epsilon, x, _, ctx_{guard}, _\rangle \quad \bigwedge_{1 \leq n \leq k} ctx_{guard} \notin \boldsymbol{c}_n}{\langle op, x, i, ctx, counts\rangle \rightarrow \langle op.f_{\texttt{default}}, x, i, ctx \cdot \perp, counts\rangle} \text{[Switch-Default]}$$

(c) BranchProgram Semantics

Figure 2.12: Small-step semantics of BranchProgram. Each rule is of the type: CondGen $\times$ *States* $\rightarrow$ CondGen $\times$ *States*. The notation in *(b)* shows how individual parts of the BranchProgram are referred to. For example, the SimpleProgram used as a condition is denoted as *guard*.

2.1.3.4 *Semantics of* `StateProgram`

The semantics of `StateProgram` are described by the rules [StateUpdate], [Update] and [StateSwitch] in Figure 2.13. In our work, the state is represented as a set of counters associated with each `SwitchProgram`. First, the rule [StateUpdate] is used to execute `StateUpdate` program which determines how the counters are updated. The execution of `StateUpdate` is similar to `BranchProgram` and results in selecting one of the update operations `INC`, `DEC` or `SKIP` to be executed next.

The goal of the *update* instructions is to update the state. Their semantics are described by the [Update] rule and each instruction computes value of the updated counter and is described by following function *update*:

$$update\colon \{\texttt{INC}, \texttt{DEC}, \texttt{SKIP}\} \times \mathbb{N} \to \mathbb{N}$$

defined as follows: *(i)* $update(\texttt{INC}, n) = n + 1$ increments the value by one, *(ii)* $update(\texttt{DEC}, n) = max(0, n-1)$ decrements the value by one, bounded from below by zero, and *(iii)* $update(\texttt{SKIP}, n) = n$ keeps the value unchanged.

Finally, after the state has been updated, the `StateSwitch` program is executed. The semantics of the `StateSwitch` are very similar to `BranchProgram` and are defined by the rules [StateSwitch] and [StateSwitch-Def]. The only difference is that the guard is not a program but simply the value of the state associated with the given `StateSwitch` statement.

$$\frac{op \in \texttt{StateUpdate} \qquad \langle op.guard, x, i, [], counts\rangle \to \langle \epsilon, x, _, ctx_{guard}, _\rangle \qquad ctx_{guard} \in \boldsymbol{c}_j \qquad \bigwedge_{1 \le n < j} ctx_{guard} \notin \boldsymbol{c}_n}{\langle op :: s, x, i, ctx, counts\rangle \to \langle op.f_j :: s, x, i, ctx, counts\rangle} \;[\text{StateUpdate}]$$

$$\frac{op \in \texttt{StateUpdate} \qquad \langle op.guard, x, i, [], counts\rangle \to \langle \epsilon, x, _, ctx_{guard}, _\rangle \qquad \bigwedge_{1 \le n \le k} ctx_{guard} \notin \boldsymbol{c}_n}{\langle op :: s, x, i, ctx, counts\rangle \to \langle op.f_{\texttt{default}} :: s, x, i, ctx, counts\rangle} \;[\text{StateUpdate-Def}]$$

$$\frac{op \in \{\texttt{INC}, \texttt{DEC}, \texttt{SKIP}\} \qquad n = update(op, counts[s]) \qquad counts' = counts[s \to n]}{\langle op :: s, x, i, ctx, counts\rangle \to \langle s, x, i, ctx, counts'\rangle} \;[\text{Update}]$$

$$\frac{op \in \texttt{StateSwitch} \qquad counts[op] \in \boldsymbol{c}_j \qquad \bigwedge_{1 \le n < k} counts[op] \notin \boldsymbol{c}_n}{\langle op, x, i, ctx, counts\rangle \to \langle op.f_j, x, i, ctx \cdot j, counts\rangle} \;[\text{StateSwitch}]$$

$$\frac{op \in \texttt{StateSwitch} \qquad \bigwedge_{1 \le n \le k} counts[op] \notin \boldsymbol{c}_n}{\langle op, x, i, ctx, counts\rangle \to \langle op.f_{\texttt{default}}, x, i, ctx \cdot \bot, counts\rangle} \;[\text{StateSwitch-Def}]$$

Figure 2.13: Small-step semantics of `StateProgram`. Each rule is of the type: `CondGen` $\times$ *States* $\to$ `CondGen` $\times$ *States*.

2.2 FROM CondGen TO A PROBABILISTIC MODEL

Having defined the syntax and semantics of the `CondGen` language, we now describe the probabilistic model used in our work. Concretely, given a program $p \in$ `CondGen`, our goal is to estimate the probability of next word w_t using the conditioning context computed by executing p on all the preceding words $\boldsymbol{w}_{<t}$:

$$P(w_t \mid p(\boldsymbol{w}_{<t})) \tag{2.3}$$

Given the context computed using $p(\boldsymbol{w}_{<t})$, we can use a wide range of probabilistic models to estimate P from the data, including neural networks, support vector machines, log-bilinear models and more. However, in our work we estimate p using a very simple model – maximum likelihood estimation via counting:

$$P(w_t \mid p(\boldsymbol{w}_{<t})) = \frac{Count(p(\boldsymbol{w}_{<t}) \cdot w_t)}{Count(p(\boldsymbol{w}_{<t}))} \tag{2.4}$$

where $Count(p(\boldsymbol{w}_{<t}) \cdot w_t)$ denotes the number of times the word w_t is seen in the training dataset together with the context produced by $p(\boldsymbol{w}_{<t})$. Similarly, $Count(p(\boldsymbol{w}_{<t}))$ denotes the number of times the context was seen regardless of the subsequent word. We have selected such a simple model for two reasons:

- It is fast to train and query. Once a program $p \in$ `CondGen` is found, training takes only a couple of seconds even on large datasets since it involves only counting in a single pass over the training data.
- It regularizes the learning towards finding better conditioning context $p(\boldsymbol{w}_{<t})$, rather than using the model capacity to make good predictions with larger but noisy contexts. This is especially useful when the goal is to find interpretable programs that can be inspected by a domain expert.

Example We illustrate how the probabilistic model is built on a simple example shown in Figure 2.14. Here, the model is parametrized using a `BranchProgram` and a training dataset with a single input sequence `a1a2b1b2`. Executing the guard program `LEFT WRITE_CHAR` splits the input into two paritions: *(i)* containing samples at positions $2, 4, 6, 8$ processed by program f_1, and *(ii)* containing samples $1, 3, 5, 7$ processed by program f_2. For each of these partitions, we estimate separate models from the data, as shown in the two tables in Figure 2.14 *(bottom)*. For position $t = 8$, the probability is estimated using $Count(f_1(\boldsymbol{w}_{<8}) \cdot w_8)/Count(f_1(\boldsymbol{w}_{<t}))$, where $f_1(\boldsymbol{w}_{<8}) = 1$. Here, $Count(1) = 2$ because $f_1(\boldsymbol{w}_{<t}) = 1$ for positions 4 and 8. Similarly, $Count(1 \cdot 2) = 2$ for the same positions. As a result, the estimated probability of word 1 at position $t = 8$ is 1 (i.e., the model is 100% certain). In this case, the model is certain since it found good conditioning context using program f_1, which recognizes that the sequence of numbers is repeating. On the other hand, the conditioning context for f_2 is empty and can be further improved. Note that

```
switch LEFT WRITE_CHAR
    case 'a' or 'b' then LEFT LEFT WRITE_CHAR    (f1)
    default ε                                    (f2)
```

t	$\boldsymbol{w}_{<t}$	w_t	LEFT WRITE_CHAR	subprogram
1		a	ϵ	f_2
2	a	1	a	f_1
3	a1	a	1	f_2
4	a1a	2	a	f_1
5	a1a2	b	2	f_2
6	a1a2b	1	b	f_1
7	a1a2b1	b	1	f_2
8	a1a2b1b	2	b	f_1

t	$f_1(\boldsymbol{w}_{<t})$	$P_{f_1}(w_t \mid f_1(\boldsymbol{w}_{<t}))$
2	a	$\frac{Count(\texttt{a}\cdot 1)}{Count(\texttt{a})} = \frac{1}{1} = 1.0$
4	1	$\frac{Count(1\cdot 2)}{Count(1)} = \frac{2}{2} = 1.0$
6	2	$\frac{Count(2\cdot 1)}{Count(2)} = \frac{1}{1} = 1.0$
8	1	$\frac{Count(1\cdot 2)}{Count(1)} = \frac{2}{2} = 1.0$

t	$f_2(\boldsymbol{w}_{<t})$	$P_{f_2}(w_t \mid f_2(\boldsymbol{w}_{<t}))$
1	ε	$\frac{Count(\varepsilon\cdot a)}{Count(\varepsilon)} = \frac{2}{4} = 0.5$
3	ε	$\frac{Count(\varepsilon\cdot a)}{Count(\varepsilon)} = \frac{2}{4} = 0.5$
5	ε	$\frac{Count(\varepsilon\cdot b)}{Count(\varepsilon)} = \frac{2}{4} = 0.5$
7	ε	$\frac{Count(\varepsilon\cdot b)}{Count(\varepsilon)} = \frac{2}{4} = 0.5$

Figure 2.14: Illustration of a model learned from the input sequence a1a2b1b2. The BranchProgram splits the input into two partitions for which separate models P_{f_1} and P_{f_2} are estimated from the data. Note that for position $t = 2$, evaluating $f_1(\boldsymbol{w}_{<2}) = \texttt{a}$ returns the character a. This is because the semantics of executing the second LEFT instructions is to keep the position unchanged if there is no preceding character in the input.

for an empty context, the model will always return unconditional maximum likelihood estimates. Because the frequency of characters a and b is the same, the unconditional estimate in our example is 0.5 for both of them.

smoothing A common issue inherent to learning probabilistic language models is adjusting the maximum likelihood estimation by taking into account data sparseness. This is critical in improving the overall precision of the system, as otherwise the model becomes overconfident in predictions based on rarely seen conditioning sets (by assigning them high probability) and conversely can completely reject unseen contexts (by assigning them zero probability). As a concrete example, the model from Figure 2.14 assigns all the probability mass to a single

word (i.e., $P(1 \mid p(\boldsymbol{w}_{<6})) = 1.0$ and $P(2 \mid p(\boldsymbol{w}_{<6})) = 0$). To deal with data sparseness, we use both modified Kneser-Ney smoothing [119, 120] as well as Witten-Bell interpolation smoothing [121]. In both cases, the idea is to fall back to lower order (i.e., shorter) conditioning sets when computing the maximum likelihood estimate in case the given conditioning set is rare. The backoff order used in our work is the order in which features were added to the conditioning context.

2.2.1 *Extension: Predicting Out-of-Vocabulary Labels*

A common limitation of probabilistic models is that they cannot predict values not seen in the training data. We offer a mitigation for this using `EqualityPrograms` that consist of a pair of `SimplePrograms` $\langle p_{ctx}, p_{eq} \rangle$. The goal of the first program p_{ctx} is as before – to accumulate context used to condition the prediction. The goal of the second program p_{eq} is to modify the probabilistic model so that it can express the equality of the output label to a value contained in the input.

We note that our technique can be applied to any existing probabilistic models, by adjusting the training and inference procedures as defined next. To keep our discussion general, we will replace the notation $(\boldsymbol{w}_{<t}, w_t)$ with the standard notation $(\boldsymbol{x}, y)$ denoting the input and the ground truth label, respectively.

TRAINING First, we discuss the modified training procedure. Let $ctx_{eq} = c_1 \cdots c_n = p_{eq}(\boldsymbol{x})$ be the context computed by the program p_{eq} on a training sample $(\boldsymbol{x}, y)$. Recall that y is the ground truth label to be predicted for the input $\boldsymbol{x}$. We define:

$$y' = \begin{cases} eq_i & \text{if } \exists i.c_i = y \quad \text{(select } i \text{ to be minimal)} \\ y & \text{otherwise} \end{cases}$$

In other words, if any of the accumulated values is equal to the label y we are predicting, we replace the label with a special symbol eq_i and train the probabilistic model as before, but with y' instead of y.

PREDICTION Because we train the model to predict the special equality symbols eq_i, the model assigns part of the probability distribution to these symbols also at the inference time. As a result, we adjust the standard inference computed using $\arg\max_{y \in \mathcal{V}} P(y \mid \boldsymbol{x})$ to take into account the equality labels as follows:

$$\underset{y \in \mathcal{V} \cup ctx_{eq}}{\arg\max}\, P(y \mid \boldsymbol{x}) + \sum_{\forall i.c_i = y} P(eq_i \mid \boldsymbol{x})$$

Here, $\mathcal{V}$ denotes the model vocabulary and $ctx_{eq} = c_1 \cdots c_n = p_{eq}(\boldsymbol{x})$ is the context accumulated by program p_{eq}. The formulation above modifies the standard inference in two key ways: *(i)* it extends the vocabulary with all the labels computed by the equality program ctx_{eq}, and *(ii)* the probability mass of each equality label is included using $\sum_{\forall i.c_i = y} P(eq_i \mid \boldsymbol{x})$. Note that we use sum since multiple values in ctx_{eq} can refer to the same concrete value.

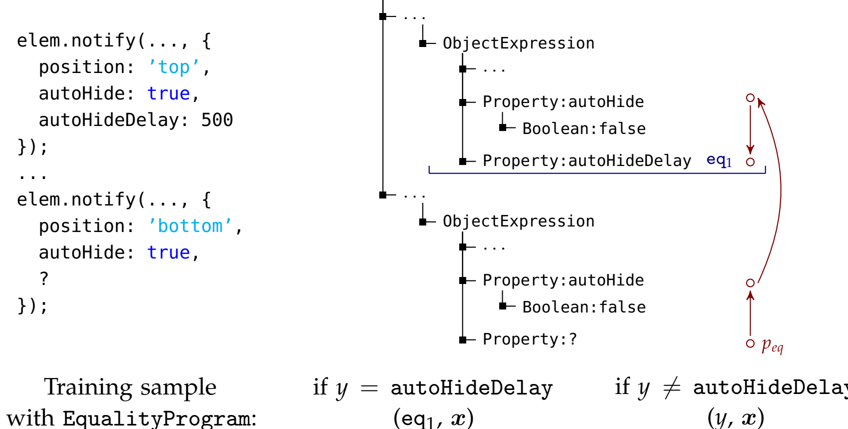


FIGURE 2.15: Illustration of relating the predicted label to a value already present in the program. If the ground truth label y matches the value found by the p_{eq} program, then the training sample is modified to predict a special equality symbol eq_1 instead of the standard label.

Example Consider the code snippet shown in Figure 2.15 that predicts the name of the next property that the developer should set. The ground truth value is a property name `autoHideDelay`, which is already present in the input program at a location specified by the following program:

$$p_{eq} ::= \texttt{LEFT PREV_NODE_CONTEXT RIGHT WRITE_VALUE}$$

The execution of p_{eq} is sketched with arrows in Figure 2.15 and will return the conditioning context $ctx_{eq} = \texttt{autoHideDelay}$.

If we train on Figure 2.15 with p_{ctx} set to the empty program and p_{eq} as described above, we will learn a probability distribution that $Pr(y = eq_1) = 1$. This means that the model is certain that the value to be predicted is equal to the first value determined by the program p_{eq}. Then, given a program at query time (could be another program, e.g., from Figure 2.2 *(d)*), this model will predict that y is equal to the value returned by p_{eq} (the predicted value will be `user-agent` for the program in Figure 2.2 *(d)*).

In our work, the semantics for p_{eq} relate a predicted label y to another value in the input x with an equality predicate. An interesting item for future work is to use other predicates to predict values that are a sum, a concatenation or another function on possibly several values in the input x. Further, in addition to equality constraints, it would also be possible to learn inequality constraints, such as that two parameters should not alias.

2.3 LEARNING

In this section, we present our approach for learning probabilistic language models parametrized by a program in CondGen.

DATASET Our training data is built by first taking the set of available inputs (e.g., programs or natural language text) and generating a set of pairs where each pair consists of a word in the input (to be predicted) and all the preceding words. More formally, our training dataset is $\mathbb{D} = \{(\boldsymbol{x}_j, y_j)\}_{j=1}^{n}$ of n samples, where $\boldsymbol{x}_j \in \mathbb{X}$ are inputs and $y_j \in \mathbb{Y}$ are outputs (correct predictions for the partial programs). Each possible output is called a label and $\mathbb{Y}$ is the set of possible labels. As an example, an input sentence $s = (w_1, w_2, \ldots, w_k)$ would be transformed into k training samples $\{(\boldsymbol{w}_{<t}, w_t)\}_{t=1}^{n}$, where w_t is the word to be predicted at position t and $\boldsymbol{w}_{<t}$ are all the preceding words seen in the input.

PROBLEM STATEMENT The goal of the learning is to minimize the following optimization objective:

$$\underset{p \in \texttt{CondGen}}{\arg\min}\ \mathbb{E}_{(\boldsymbol{x},y)\sim\mathbb{D}}\ell(\boldsymbol{x}, y; p) + \lambda \cdot \Omega(p) \tag{2.5}$$

where $\ell(\boldsymbol{x}, y; p)$ is a loss function that measures the performance of a model parametrized by program p, $\Omega\colon \texttt{CondGen} \rightarrow \mathbb{R}^{+}$ is a regularization term used to avoid over-fitting to the data by penalizing complex programs and $\lambda \in \mathbb{R}$ is a regularization constant. We instantiate $\Omega(p)$ to return the number of instructions in the program p. As the loss function, we use the cross-entropy loss. As we have no access to the underlying distribution but only to the dataset $\mathbb{D}$, we approximate the expected loss as:

$$\mathbb{E}_{(\boldsymbol{x},y)\sim\mathbb{D}}\ell(\boldsymbol{x}, y; p) = \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x},y)\in\mathbb{D}} -\log_2 P(y \mid p(\boldsymbol{x}_{<t})) \tag{2.6}$$

SCALING TO LARGE DATASETS In order for the learning procedure to explore a large number of candidate programs in a reasonable time, it is important that all the algorithms presented in this section scale to large datasets (in our experiments $|\mathbb{D}| = 10^8$) without the need to restrict how much data can be used for learning. To mitigate this problem, we use the representative dataset sampling technique [9]. The main idea is to select a small sample $|d| \ll |\mathbb{D}|$ such that $\left|\mathbb{E}_{(\boldsymbol{x},y)\sim\mathbb{D}}\ell(\boldsymbol{x}, y; p_i) - \mathbb{E}_{(\boldsymbol{x},y)\sim d}\ell(\boldsymbol{x}, y; p_i)\right| \leq \epsilon$ for all previously generated programs p_i. That is, evaluating programs on a small dataset d approximates evaluation on the full dataset $\mathbb{D}$ within error ϵ that is as small as possible.

2.3.1 *Learning* SimplePrograms

We next describe our approach to synthesizing programs $p \in$ SimpleProgram such that they minimize Equation 2.5. We use a combination of three techniques to solve this optimization problem and find $p^{\approx}_{best}$ – an *exact* enumeration, *approximate* genetic programming search, and Markov chain Monte Carlo (MCMC) search.

ENUMERATIVE SEARCH We start with the simplest approach that enumerates all possible programs up to some instruction length. As the number of programs is exponential in the number of instructions, we enumerate only short programs with up to 5 instructions (not considering PREV_CHAR(c)) that contain a single Write instruction. The resulting programs serve as a starting population for a follow-up genetic programming search.

The reason we omit the PREV_CHAR(c) instruction is that this instruction is parametrized by a character c that has a large range (of all possible characters in the training data). Considering all variations of this instruction would lead to a combinatorial explosion.

GENETIC PROGRAMMING SEARCH The genetic programming search proceeds in several epochs, where each epoch generates a new set of candidate programs by randomly mutating the programs in the current generation. Each candidate program is generated using the following procedure:

1. select a random program p from the current generation
2. select a random position i in p, and
3. apply a mutation that removes, inserts or replaces the instruction at position i with a randomly selected new instruction (not considering PREV_CHAR(c)).

These candidate programs are then scored with the objective function (Equation 2.5) and after each epoch, a subset of them is retained for the next generation while the rest is discarded. The policy we use to discard programs is to randomly select two programs and discard the one with the worse score. We keep discarding programs until the generation has less than 25 candidate programs. Finally, we do not apply a cross-over operation in the genetic search procedure.

MARKOV CHAIN MONTE CARLO SEARCH (MCMC) Once a set of candidate programs is generated using a combination of enumerative search and genetic programming, we apply MCMC search to further improve the programs. This procedure is the only one that considers the PREV_CHAR(c) instruction (which has 108 and 212 variations for the *Linux Kernel* and *Hutter Prize Wikipedia* datasets respectively)[6].

6 In our evaluation, we apply this step only to character level datasets where the number of Write instructions is large. For programs, there are fewer than 15 Write instructions and therefore this step is not required.

The synthesized program is a combination of several basic building blocks consisting of a few instructions. To discover a set of good building blocks, at the beginning of the synthesis we first build a probability distribution I that determines how likely a building block will be the one with the best cost metric, as follows:

- Consider all building blocks that consist of up to three `Move` and one `Write` instruction, $\mathcal{B} : \{\texttt{empty}, \texttt{Move}\}^3 \times \texttt{Write}$.
- Score each building block $b \in \mathcal{B}$ on the full dataset $\mathbb{D}$ by calculating the *bits-per-character* (BPC) b_{bpc} (as defined in Equation 2.6) and the error rate b_{error_rate} (i.e., the fraction of correctly predicted labels) on dataset $\mathbb{D}$.
- Accept the building block with probability $min(1.0, \texttt{empty}_{bpc}/b_{bpc})$ where $\texttt{empty}_{bpc}$ is the score for the unconditioned `empty` program. Note that for the BPC metric, lower is better. That is, if the program has better (lower) BPC than the `empty` program, it is always accepted. Otherwise, it is accepted with probability $\texttt{empty}_{bpc}/b_{bpc}$.
- For an accepted building block b, set the score as $I'(b) = 1.0 - b_{error_rate}$, that is, the score is proportional to the number of samples in the dataset $\mathbb{D}$ that are classified correctly using the building block b.
- Set the probability with which a building block $b \in \mathcal{B}$ will be sampled by normalizing the distribution I', that is, $I(b) = I'(b)/\sum_{b' \in \mathcal{B}} I'(b')$.

Given the probability distribution I, we now perform random modifications of a candidate program p by appending/removing such blocks according to the distribution I in a MCMC procedure that does a random walk over the set of possible candidate programs. That is, at each step we are given a candidate program, and we sample a random piece from the distribution I to either randomly add it to the candidate program or remove it (if present) from the candidate program. Then, we keep the updated program either if the score of the modified candidate program improves (according to Equation 2.5), or with a low probability even if the score did not improve.

2.3.2 *Learning* `EqualityPrograms`

The learning procedure for `EqualityPrograms` is the same as for `SimplePrograms`, except that we learn two programs – p_{ctx} and p_{eq}. To speed up the learning, in practice we first learn a set of good `SimplePrograms` which are then extended with the second equality program.

2.3.3 *Learning* `BranchPrograms`

The key idea of our approach is to phrase the problem of learning a probabilistic model of code parametrized with `BranchProgram` as learning a decision tree.

This is possible because a decision tree can in fact be seen as a restricted form of a `BranchProgram` with the following shape:

$$\textbf{if } (pred(\boldsymbol{x})) \textbf{ then } p_a \textbf{ else } p_b$$

where *pred* is a boolean predicate on the input $\boldsymbol{x}$ and p_a, p_b are recursive branches of the tree or unconditional probability distribution for leafs. The semantics of decision trees are standard: check the predicate $pred(\boldsymbol{x})$ and depending on the outcome execute either p_a or p_b.

While restrictive, the above decision tree formulation can be easily extended to our setting by allowing more than two branches and by replacing the unconditional probability at leafs with more powerful probabilistic models (in our case using `SimplePrograms` and `EqualityPrograms`). This enables us to cleanly instantiate existing decision tree learning algorithms and to obtain new variants. These variants have not been explored previously, yet turn out to be practically useful.

PRELIMINARIES: ENTROPY AND INFORMATION GAIN Our goal at learning time is to discover a model that "explains well" the training data $\mathbb{D}$. One possible and commonly used metric to measure this is the entropy of the resulting probability distribution, estimated as follows:

$$H(\mathbb{D}, p) = -\sum_{(\boldsymbol{x},y)\in\mathbb{D}} \frac{1}{|\mathbb{D}|} \log_2 P(y \mid p(\boldsymbol{x})) \tag{2.7}$$

We use a variant of the entropy metric called cross-entropy, which makes the entropy formulation above equivalent to the loss defined earlier in Equation 2.6.

The intuition behind decision tree learning is that we can recursively split the dataset into two smaller partitions (branches), each of which is allowed to specialize for the given subset of the dataset (e.g., by choosing suitable features). The goal of the optimization algorithm is then to select a predicate that leads to the best model performance. To measure the effect different predicates have on the model performance, a commonly used metric is called information gain defined as follows:

$$IG(D, pred) = H(\mathbb{D}, \epsilon) - \frac{\left|\mathbb{D}_{pred}\right|}{|\mathbb{D}|} H(\mathbb{D}_{pred}, \epsilon) - \frac{\left|\mathbb{D}_{\neg pred}\right|}{|\mathbb{D}|} H(\mathbb{D}_{\neg pred}, \epsilon) \tag{2.8}$$

Here, *pred* is the predicate used to split the dataset, $\mathbb{D}_{pred} = \{(\boldsymbol{x}, y) \in \mathbb{D} \mid pred(\boldsymbol{x})\}$ denotes the subset of the dataset for which the predicate evaluates to true and similarly, $\mathbb{D}_{\neg pred} = \{(\boldsymbol{x}, y) \in \mathbb{D} \mid \neg pred(\boldsymbol{x})\}$ denotes the subset of the dataset for which the predicate evaluates to false. For a given predicate *pred*, the information gain quantifies how many bits of information will be saved if we split the dataset with the predicate. For simplicity, we use an unconditional model, denoted using ϵ, for the entropy on the full dataset and both partitions. However, as we will see shortly, the above definition can be instantiated with other models which helps to find better predicates.

Algorithm 1: Decision tree algorithm for learning `BranchPrograms`.

```
def Learn(d_full, d_branch, syn)
    Input: Dataset d, local synthesis procedure syn
    Output: Program p ∈ BranchProgram
1   begin
2       if |d_branch| < 250 ∧ |d_branch| < 0.1 |d_full| then
3           return ε_i                          ▷ Stop if d_branch is too small
4       p ← syn(d_branch)
5       where p ≡ if (pred(x)) then p_a else p_b
6       p_a ← Learn(d_full, {(x, y) ∈ d_branch | pred(x)}, syn)
7       p_b ← Learn(d_full, {(x, y) ∈ d_branch | ¬pred(x)}, syn)
8       return p ≡ if (pred(x)) then p_a else p_b
```

ALGORITHM Our greedy learning algorithm for learning `BranchPrograms` is shown in Algorithm 1. A useful benefit of our approach is that we can instantiate different decision tree learning algorithms by simply varying the fragment of the language to which the learned program p belongs, along with the corresponding search procedure (denoted as *syn*). In this way, we instantiate existing algorithm ID3 [118] as well as our own variants ID3+ and E13 described next.

ID3 DECISION TREE LEARNING We instantiate ID3, one of the most commonly used and studied decision tree algorithms [117], as follows. Let $\texttt{BranchProgram}_0$ be a fragment of `BranchProgram` with programs in the following shape:

$$\textbf{if } (pred(\boldsymbol{x})) \textbf{ then } \epsilon_a \textbf{ else } \epsilon_b$$

where ϵ_a and ϵ_b are empty programs. Then, for a dataset $d \subseteq \mathbb{D}$ we define $syn_0(d) = \arg\max_{p \in \texttt{BranchProgram}_0} IG(d, p)$. That is, our goal is to find a program that maximizes the information gain[7]. We obtain the ID3 learning algorithm by invoking Algorithm 1 with $syn = syn_0$. Then, at each step of the algorithm, we synthesize a single branch using syn_0, split the data according to the branch and call the algorithm recursively. As a termination condition, we stop recursing once the dataset is smaller than a certain size (250 samples), and smaller than a certain fraction of the full dataset (10%). This limits the depth of the tree and prevents overfitting to the training data.

Nota that since the space of programs is very large we use approximations in our maximization procedure. Concretely, for a given candidate predicate, the possible branch targets are the 32 most common values obtained by executing the predicate on the training data. We then select the number of case clauses and their associated constants using a combination of enumerative search and genetic algorithm.

7 Note that here we overload the information gain definition from Equation 2.8 to take as input a program $p \in \texttt{BranchProgram}$, not a predicate. The semantics of such overloaded definition is that the information gain is computed over all the partitions specified by the program p.

Left program:

```
(p) switch LEFT WRITE_CHAR
    (f1) case 'a' or 'b' then ε
    (f2) default ε
```

$IG(\mathbb{D}, p) = 2 - 1/2 \cdot 1 - 1/2 \cdot 1 = 1$

t	$P_{f_1}(w_t \mid f_1(\boldsymbol{w}_{<t}))$
2	$\frac{Count(\varepsilon\cdot 1)}{Count(\varepsilon)} = \frac{2}{4} = 0.5$
4	$\frac{Count(\varepsilon\cdot 2)}{Count(\varepsilon 1)} = \frac{2}{4} = 0.5$
6	$\frac{Count(\varepsilon\cdot 1)}{Count(\varepsilon)} = \frac{2}{4} = 0.5$
8	$\frac{Count(\varepsilon\cdot 2)}{Count(\varepsilon)} = \frac{2}{4} = 0.5$
	$H(\mathbb{D}_{f_1}, f_1) = 1$

t	$P_{f_2}(w_t \mid f_2(\boldsymbol{w}_{<t}))$
1	$\frac{Count(\varepsilon\cdot a)}{Count(\varepsilon)} = \frac{2}{4} = 0.5$
3	$\frac{Count(\varepsilon\cdot a)}{Count(\varepsilon)} = \frac{2}{4} = 0.5$
5	$\frac{Count(\varepsilon\cdot b)}{Count(\varepsilon)} = \frac{2}{4} = 0.5$
7	$\frac{Count(\varepsilon\cdot b)}{Count(\varepsilon)} = \frac{2}{4} = 0.5$
	$H(\mathbb{D}_{f_2}, f_2) = 1$

Right program:

```
(p) switch LEFT WRITE_CHAR
    (f1) case 'a' then ε
    (f2) default ε
```

$IG(\mathbb{D}, p) = 2 - 1/4 \cdot 1 - 3/4 \cdot 1.92 = 0.31$

t	$P_{f_1}(w_t \mid f_1(\boldsymbol{w}_{<t}))$
2	$\frac{Count(\varepsilon\cdot 1)}{Count(\varepsilon)} = \frac{1}{2} = 0.5$
4	$\frac{Count(\varepsilon\cdot 2)}{Count(\varepsilon 1)} = \frac{1}{2} = 0.5$
	$H(\mathbb{D}_{f_1}, f_1) = 1$

t	$P_{f_2}(w_t \mid f_2(\boldsymbol{w}_{<t}))$
1	$\frac{Count(\varepsilon\cdot a)}{Count(\varepsilon)} = \frac{2}{6} = 0.33$
3	$\frac{Count(\varepsilon\cdot a)}{Count(\varepsilon)} = \frac{2}{6} = 0.33$
5	$\frac{Count(\varepsilon\cdot b)}{Count(\varepsilon)} = \frac{2}{6} = 0.33$
6	$\frac{Count(\varepsilon\cdot 1)}{Count(\varepsilon)} = \frac{1}{6} = 0.16$
7	$\frac{Count(\varepsilon\cdot b)}{Count(\varepsilon)} = \frac{2}{6} = 0.33$
8	$\frac{Count(\varepsilon\cdot 2)}{Count(\varepsilon)} = \frac{1}{6} = 0.16$
	$H(\mathbb{D}_{f_2}, f_2) = 1.92$

FIGURE 2.16: Comparing two different `BranchPrograms` trained on the input sequence `a1a2b1b2`. The entropy of the unconditional distribution (Equation 2.7) on the full input is $H(\mathbb{D}, \epsilon) = 2$. The information gain (Equation 2.8) of the left and right programs is 1 and 0.31, respectively. As a result, the left program will be selected over the right program when using ID3 algorithm.

Example As a concrete example, let us consider the two candidate `BranchPrograms` shown in Figure 2.16. Both programs can be explored during the ID3 learning as they belong to the $\texttt{BranchProgram}_0$ fragment (they both contain empty programs in the case clauses. To select which program is better, the ID3 algorithm selects the one with the higher information gain, computed using the Equation 2.8. In this case, the program with **case** `'a'` **or** `'b'` has higher information gain is it manages to split the dataset into two partitions with disjoint labels – one partition that contains labels `1`, `2` and a second partition with labels `a`, `b`.

ID3+ DECISION TREE LEARNING Our formulation of ID3 as learning a program in a `BranchProgram` fragment allows us to extend and improve the algorithm by making modifications of the `BranchProgram` fragment. We also provide exten-

sions that leverage the capability of the `BranchProgram` to express probabilistic models other than standard decision trees.

In particular, classic ID3 learning results in decision trees with the leafs of the tree being empty programs. Our models, however, allow combining branch instructions with other probabilistic models like in Figure 2.14. To handle such combinations, we extend the ID3 algorithm to also generate programs in the tree leafs.

Let $\texttt{BranchProgram}_S$ be a fragment of `BranchProgram` where tree leafs can be either empty programs or `SimplePrograms`. Programs in this fragment may describe probabilistic models without branches such as language models [68, 122] or other models (e.g., [6, 9]). Let $syn_S(d) = \arg\max_{p \in \texttt{BranchProgram}_S} IG(d, p)$ be a synthesis procedure which computes a program that best fits d. Then, the ID3+ learning algorithm extends ID3 by replacing the empty programs returned on line 3 of Algorithm 1 with $syn_S(d)$. The branch synthesis of the ID3+ algorithm uses the same syn_0 procedure as ID3 and as a result constructs a tree with the same structure. The only difference in the resulting tree is the programs in the leaves of the tree.

E13 ALGORITHM The ID3+ learning algorithm essentially first runs ID3 to generate branches until too few samples fall into a branch and then creates a probabilistic model for the final samples. Intuitively, there are two potential issues with this approach. First, ID3+ always trains the probabilistic models based on $\texttt{BranchProgram}_S$ only on small datasets. These small datasets may result in learning inaccurate models at the leaves. Second, syn_0 always minimizes information with respect to empty programs, but the leaves of the tree may contain non-empty programs (i.e., more powerful probabilistic models).

To address these limitations, we propose to instantiate the learning algorithm so that instead of using syn_0 as ID3 does, it uses the following procedure:

- First, let $syn_S(d) \in \texttt{BranchProgram}$ be the best program that we would synthesize for d if that program was a leaf node in the ID3+ algorithm.
- Let $\texttt{BranchProgram}_A$ be a fragment of `BranchProgram` with programs in the shape $p \equiv \textbf{if}\ (pred(\boldsymbol{x}))\ \textbf{then}\ p_a\ \textbf{else}\ p_b$ where $pred$ is a predicate and p_a and p_b are either the empty program or $syn_S(d)$.
- Finally, to obtain our final E13 algorithm, in Algorithm 1 we set syn to $syn_A = \arg\max_{p \in \texttt{BranchProgram}_A} IG(d, p)$.

2.4 EVALUATION: PROBABILISTIC MODELS OF CODE

In this section, we provide a thorough experimental evaluation of the learning approach presented so far, instantiated to the domain of code. For the purposes of evaluation, we chose the tasks of learning probabilistic models of code for two programming languages – JavaScript and Python. This is a challenging setting because the dynamic nature of these languages makes it difficult to extract precise semantic information via static analysis (e.g., type information). To evaluate our

probabilistic model, we built a code completion system, called `Deep3`, capable of predicting *any* JavaScript or Python program element.

KEY BENEFITS OF OUR APPROACH We demonstrate the key benefits of our approach by showing that:

- Our scalable learning allows for synthesizing large, complex `CondGen` programs. The probabilistic model parametrized by these programs significantly improves state-of-the-art accuracy for both JavaScript and Python dataset.
- Our model is precise enough to allow new types of predictions, not possible with prior approaches. For example, we can predict loops and try statements with 65% and 54% accuracy, respectively. In contrast, prior state-of-the-art models completely fail and achieve accuracy close to zero.
- The programs synthesized by our approach are interesting beyond the induced probabilistic model. Our synthesized `CondGen` programs enable us to highlight the program elements used to make each prediction, and can be used to explain and justify the prediction to the user.
- Even though our models are non-neural and instantiated with a simple maximum likelihood model (based on counting), they are comparable and even better than a number of sophisticated deep learning models developed 2+ years after our work. However, the latest work based on transformer models [29] does improve upon our work for a range of prediction types.

EXPERIMENTAL COMPARISON OF VARIOUS SYSTEMS In our experiments we compared the performance of several systems:

- PCFG and N-GRAM: we include two commonly used probabilistic models based on probabilistic context free grammars (PCFGs) and n-gram models (for $n = 3$). Despite their limited accuracy, these models are used by a number of existing programming tools [61, 68, 112, 123, 124].
- DeepSyn: we use a previous state-of-the-art system for JavaScript code completion. For a fair comparison, we implemented the model from [9] and trained it on the full Python and JavaScript languages (as opposed to only JavaScript APIs and fields as in [9]) by learning specialized models for each AST node type.
- SVM: instead of learning a generative model as in our work, a number of prior approaches (e.g., [22, 67, 123, 124]) learn discriminative models. In our evaluation, we compare to a discriminative model based on support vector machine (SVM) instantiated with syntactic feature functions that correspond to the types and values of the 10 nodes preceding the completion position in the AST (similar to the features used in [123]). To learn the feature weights, we used an online learning algorithm based on hinge loss (following [22])

and performed grid-search for the parameters that control the regularization (L_∞), the learning rate and the SVM margin.

- NEURAL MODELS: recently, applying deep learning has shown great success for the domain of code. In fact, the deep learning based approaches are currently so popular that they became de-facto standard models employed by the majority of research papers published in the last two years. We include comparison to a number of recent models including LSTM [27, 95, 125], Code2Seq [28] and Transformers [29, 116]. However, we do note that these works were developed *after* our models by 2 [27, 28] to 4 years [29].
- ID3+ and E13: these are the learning algorithms proposed in this work and implemented in `Deep3`.

JAVASCRIPT DATASET JS150 For our evaluation, we collected a corpus from GitHub repositories containing 150 000 non-obfuscated JavaScript files and made it publicly available at `www.sri.inf.ethz.ch/js150`. The first 100 000 files are used for training and the last 50 000 are used as a blind set for evaluation purposes only. From the 100 000 training data samples, we use the first 20 000 for learning the `CondGen` program. Further, in our experiments, we use only files that parse to ASTs with at most 30 000 nodes, because larger trees tend to contain JSON objects as opposed to code.

The files are stored in their corresponding ASTs formats as defined by the ESTree specification[8]. Each AST node contains two attributes – the type of the node and an optional value. As an example, consider the AST node `Property` : `autoHide` from Figure 2.15 where `Property` denotes the type and `autoHide` is the value. The number of unique types is relatively small (44 for JavaScript) and is determined by the non-terminal symbols in the grammar that describe the AST whereas the number of values (10^9 in our corpus) is very large and is a mixture of identifiers, literals and language specified operators (e. g., $+, -, *$).

PYTHON DATASET PY150 We also collected a corpus of Python programs from GitHub and made it available as ASTs at `www.sri.inf.ethz.ch/py150`. This dataset includes programs with up to 30 000 AST nodes and from projects with non-viral licenses such as MIT, Apache and BSD. To parse the dataset, we used the AST format for Python 2.7 from the parser included in the Python standard library (we also release the code that we used for parsing the input programs). Further, we would like to acknowledge the work of Kanade *et al.* [18] that provides a redistributable subset of our py150 corpus as of the year 2020.

The ASTs are stored similarly to the JavaScript ASTs such that every node includes the type and optionally a value. For example, the AST node `attr` : `path` is of type `attr` and has the value `path`. Semantically, this node corresponds to accessing the `path` attribute of a Python object.

8 https://github.com/estree/estree

	js150		py150	
Model	Types	Values	Types	Values
Prior Work				
PCFG	51.5%	50.1%	59.0%	10.2%
n-gram	69.2%	71.2%	63.2%	63.9%
`DeepSyn` [9]	74.1%	80.9%	72.5%	67.2%
SVM	67.5%	70.5%	-	-
Our Work				
PHOG	83.9%	**82.9%**	76.3%	69.2%
Follow-up Work				
LSTM [125]	84.8%	76.6%	-	-
LSTM \w Attention [27]	**88.6%**	80.6%	**80.6%**	69.8%
Pointer Mixture Network [27]	-	81.0%	-	**70.1%**

Table 2.1: Comparison of our work against a number of prior as well as follow-up works. The results show the accuracy of each model on the task of predicting arbitrary AST types and values for the js150 and py150 datasets. We can see that our model significantly outperforms prior works while being comparable, and in some cases even better, than follow-up work based on neural models.

methodology Given the structure of ASTs, we learn two different models for a programming language – one for predicting the node type and a second one for predicting the node value. Since both of these define probability distributions, they can be easily combined into a single model of the full programming language. To make sure our predictions also capture the structure of the tree, and not only the labels in the nodes of the tree, we also predict whether a given node should have any siblings or children. For this purpose, when predicting the type of an AST node, the label further encodes whether the given node has right siblings and children which allows us to expand the tree accordingly.

To train the model for predicting types, we generate one training sample for each AST node in the dataset by replacing it with an empty node (i. e., a hole) and removing all the nodes to the right. Following the same procedure, we generate training samples for predicting values, except that we do not remove the type of the node to be predicted, but only its value. In total, our JavaScript dataset consists of $10.7 \cdot 10^8$ samples used for training and $5.3 \cdot 10^7$ for evaluation. Our Python dataset consists of $6.2 \cdot 10^7$ training samples and $3 \cdot 10^7$ evaluation samples.

evaluation metric We measure the precision of our models by using the *accuracy* metric. Accuracy is the proportion of cases where the predicted label with the highest probability is the correct one in the evaluated program. Note that our model always provides a prediction and therefore the recall metric is 100%.

2.4.1 *Main Results*

In Table 2.1 we provide a summary of the main results. The top half contains the accuracy of prior works for predicting AST types and values for both of our JavaScript and Python datasets. Here, the best model is `DeepSyn` which consistently outperforms the second best n-model by 3% to 9%. Our tool outperforms SVM by even larger margin of 6% to 10%. In the bottom half of Table 2.1, we provide the results for a number of follow-up works based on neural networks. Similar to our work, these works perform predictions on the AST representation of the input program, but using much more sophisticated models. Even though the neural models have much higher capacity and can benefit from the distributional representation of the input words, their results are comparable to the results of `Deep3`. In particular, while neural models do improve the prediction of types, they can still be worse for the more difficult task of predicting values (e.g., by 2% for JavaScript).

To study the performance for predicting values further, we provide an additional comparison to follow-up works in Table 2.2. Here, all the models are based on neural networks and include the recent state-of-the-art `Transformer` [116] model widely used for natural language processing tasks, as well as `Code2Seq` [28] and `TravTrans` [29] models specifically designed to model program ASTs. The results in Table 2.2 show that both the LSTM and even Transformer models are significantly worse than our work (`Deep3`). Compared to `Code2Seq`, our model provides better predictions for many specialized predictions (e.g., +6% for attribute access or +4% for numeric constants), but the overall performance of both models is very similar. The only model that outperforms our work is the recent work `TravTrans` which combines transformers with the AST representation of programs. This shows that four years after our model was developed (as of the time of writing), while our model is no longer the state-of-the-art, it does outperform (or is comparable to) most neural models developed in the meantime and it remains the state-of-the-art non-neural model.

	Our Work	**Follow-up Work**			
Prediction Task	`Deep3`	`LSTM`	`Transformer`	`Code2Seq`	`TravTrans`
Value prediction MRR					
All leaf nodes	43.9%	23.8%	36.5%	43.6%	58.0%
Attribute access	45.3%	26.4%	41.0%	39.2%	60.5%
Numeric constant	53.2%	32.2%	51.7%	49.2%	63.5%
Function parameters	58.1%	45.5%	54.3%	56.5%	67.2%
		worse than `Deep3`		**similar**	**better**

Table 2.2: Mean reciprocal rank (MRR) of our work and a number of follow-up works for various value prediction tasks. All the results in this table are adapted from the follow-up work of Kim *et al.* [29].

JavaScript js150 Dataset					
	Prior Work			**Our Work:** `Deep3`	
Prediction Task	PCFG	3-gram	`DeepSyn` [9]	ID3+	E13
Node value prediction accuracy (Figure 2.18)					
Unrestricted prediction	50.1%	71.2%	80.9%	76.5%	**82.9%**
API prediction	0.04%	30.0%	59.4%	54.0%	**66.6%**
Field access prediction	3.2%	32.9%	61.8%	52.5%	**67.0%**
Node type prediction accuracy (Figure 2.17)					
Unrestricted prediction	51.5%	69.2%	74.1%	**83.9%**	80.0%
Predicting Loop statements	0%	37.5%	0.04%	**65.0%**	28.3%
Predicting Branch statements	0%	40.9%	17.3%	**65.7%**	40.4%

TABLE 2.3: Accuracy comparison for selected tasks between JavaScript models used in prior work and our technique.

In the remainder of this section, we will provide a thorough evaluation of our models, including the breakdown of predictions for types and values, inspecting and interpreting the learned programs, as well as an ablation study measuring the effect of different learning algorithms and the benefit of our technique to predict out-of-vocabulary words.

2.4.2 *Probabilistic Model for JavaScript*

In Table 2.3 we provide highlights of the accuracy for several interesting prediction tasks. Each row includes an application we evaluate on and each column includes the accuracy of the corresponding probabilistic model on this task. The tasks of "unrestricted predictions" for both values and types include a wide range of subtasks – some of them are easy and others are not. Example of an easy task is to predict that there will be nodes of type `Property` inside an `ObjectExpression` (i.e., properties inside a JSON object). As a result of these easy tasks, even the most trivial baselines such as PCFG succeed in predicting around half of the labels.

We include some of the more difficult tasks as separate rows in Table 2.3. When faced with these tasks of predicting APIs, field accesses or less frequent statements such as loops and branches, the accuracy of the PCFG model essentially goes down to 0%. This is expected as by construction, PCFG does not include the necessary context needed to make such predictions.

For every task in our experiments, a probabilistic model based on decision trees has higher accuracy than any of the previous models – PCFG, the 3-gram language model or the `DeepSyn` model. An interesting observation is that the model obtained

Prediction Task	Prior Work DeepSyn [9]	Our Work ID3+
ContinueStatement	17%	**58%**
ForStatement	0%	**60%**
WhileStatement	0%	**76%**
ReturnStatement	14%	**75%**
SwitchStatement	1%	**47%**
ThrowStatement	9%	**51%**
TryStatement	2%	**54%**
IfStatement	18%	**66%**

TABLE 2.4: List of new predictions enabled by our probabilistic model when predicting type of a statement.

from the ID3+ algorithm is more precise than the one obtained from E13 when predicting types and in contrast, E13 produces the most precise model for values. We hypothesize that the reason for this is the relatively smaller number of labels for types – there are 176 unique labels for types and around 10^9 labels for values. In our further evaluation, we take the CondGen program obtained from ID3+ for types and the CondGen program obtained from E13 for values.

2.4.2.1 *Predicting Node Types*

We first discuss the application of our model for predicting types of AST nodes, that is, learning the structure of the code. There are in total 44 different types of nodes in JavaScript that range over program statements, expressions, as well as constants and identifiers. Our predicted labels for types also include tree structure information whether the node has a right sibling and children. As a result, the total number of different labels that can be predicted is 176.

We provide a detailed list of difficult type predictions for previous models that are now enabled by our decision tree models in Table 2.4. An interesting insight of our evaluation is that there are entire classes of predictions where previous models (such as DeepSyn) fail to make correct predictions about the program structure. For example, previous models failed to predict a range of statements such as loops, switch statements, if statements and exception handling statements. One of the reasons why previous models predict such statements with very low accuracy is that these statements are significantly less frequent in code than other statements (e. g., assignments). In contrast, our model partitions the training data into multiple branches and builds precise models for each such case.

EXAMPLE COMPLETION To illustrate the difficulty of correctly predicting such queries consider the example shown in Figure 2.17. Here, the figure shows the

conditioning context →

```
for (j = 0; j < groups.length; j++) {
  idsInGroup = groups[j].filter(
    function(id) {
      return ids.indexOf(id) >= 0;
    }
  );
  if (idsInGroup.length === 0) {
    ?
  }
}
```

prediction position →

correct type →

	$Pr_{\text{ID3+}}$	Pr_{DeepSyn}	$Pr_{\text{n-gram}}$
ContinueStatement	**0.86**	0.03	0.00
ReturnStatement	0.04	0.11	0.11
ExpressionStatement	0.03	0.66	0.61
VariableDeclaration	0.02	0.06	0.10

Figure 2.17: Example of predicting type of a statement from a code snippet in our evaluation data. We show the top 4 predictions and their probabilities predicted by each system.

original code snippet with the developer querying the code completion system asking it to predict the statement at the position denoted with "?". As can be seen by inspecting the code, it is not immediately clear which statement should be filled in, as it depends on the intended semantics. However, by training on a large enough dataset and conditioning our prediction on appropriate parts of the code, we can hopefully discover some regularities that help us make good predictions.

Indeed, for this example, as well as for 58% of other queries, `Deep3` successfully predicted a `ContinueStatement`. Our model suggests `ContinueStatement` with a very high probability of 86%, whereas the second most likely prediction has only 4% probability. On the other hand, existing models (PCFG, n-gram and `DeepSyn`) are biased towards predicting `ExpressionStatement`, simply because it is three orders of magnitude more frequent than `ContinueStatement`. These models cannot discover proper conditioning to predict the correct statement with high confidence.

Interpreting the learned program To understand how `Deep3` obtained its high accuracy, we examined the program learned during the training and the branch this example fell into. The decision tree performed the following checks for this prediction:

1. check whether the query node is the first child of the current scope,
2. test whether the current scope is defined by an `IfStatement`,
3. retrieve what is the type of node that defined previous scope (where scopes are created by code blocks, functions or modules) and test if it defines a loop or other constructs such as function or an `IfStatement`.

Once all of these conditions were satisfied, the probabilistic model looked at the values in the last `IfStatement` where the query node is defined. In Figure 2.17, we highlighted with green boxes all positions in the code on which our probabilistic model conditions in order to make the correct prediction.

Prediction Task	Example	Prior Work DeepSyn	Our Work E13
API name	`this.getScrollBottom(inTop)`	59.4%	**66.6%**
API call target	`node.removeAttribute(attName)`	63.1%	**67.0%**
Array index identifier	`event[prop] = ...`	72.4%	**82.8%**
Assigment variable identifier	`result = ...`	66.8%	**70.3%**

Table 2.5: Accuracy of various applications for predicting values in JavaScript.

2.4.2.2 *Predicting Node Values*

We now turn our attention to evaluating the quality of the learned program trained for predicting values in JavaScript programs. While this task is similar to the task of predicting node types in JavaScript ASTs, some of the predictions are much more challenging because the label set for values is several orders of magnitude larger than the label set for types. In Table 2.5 we show the accuracy for several prediction tasks, as well as examples of predictions made for these tasks. For all these tasks, we improve the accuracy between 3% to 10% over the accuracy achieved by the DeepSyn model. In addition, for the API and field access prediction tasks, Deep3 outperforms DeepSyn by 6% and 5%, respectively as shown in Table 2.3. We expect that these tasks are useful in the context of IDE code completion and the improvement in accuracy should result in better user experience.

INTERPRETING THE LEARNED PROGRAM As an interesting example query for value prediction, consider the query shown in Figure 2.18 where the value should be completed with an API call. The goal of this query is to predict the API name at the position denoted by "?". For this case, the learned CondGen program investigates the context in which the API call is done (on a variable, on a field object, on this object, etc.), how the result of the call is used and since it is used as an argument in a function call, at what position that argument is. Note that while this program makes sense (it closely identifies the kind of API used), providing all this conditioning manually would require tremendous amount of effort.

2.4.3 *Predicting Out-of-Vocabulary Labels*

We next evaluate Deep3's capability to predict values not seen in the training data with the model described in Section 2.2.1. To check the effect of this extension (which affects 11% of the learned programs used as leaves in the decision tree), we performed an experiment where we compared the accuracy of the resulting probabilistic models with and without the extension. For a fair comparison, Table 2.3 summarizes the results for both DeepSyn and Deep3 with this extension enabled.

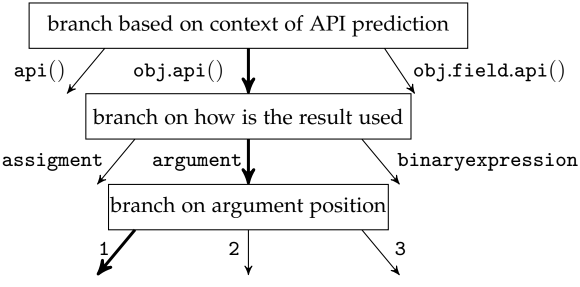


FIGURE 2.18: Example of an API completion query and visualization of the decomposition learned by our approach. As can be seen the our approach learns a specialized model that is learn on queries predicting API invoked directly on call target that are used as second argument in another method invocation.

If we disable the second program p_{eq} from Section 2.2.1, the overall accuracy for predicting values decreases by 2%, from 82.9% to 80.9%. This decrease is caused mostly by the lower accuracy of predicting identifiers and properties – these are the two prediction tasks that contain most of the user defined values. On the other hand, for predicting types, the second program that describes equality does not affect the accuracy. This is intuitive since all the possible labels for types are easily seen in the training data (there are only 176 unique labels).

2.4.4 *Learned Programs*

Using the learning approach proposed in our work, we discover a CondGen program with a large number of branches that is interesting in itself and provides several benefits beyond providing a state-of-the-art code completion system. The programs learned using the E13 algorithm contain 13,160 and 307 leaves together with 5,869 and 157 internal switch nodes in their decision trees for values and types, respectively. That number of cases is clearly infeasible to conceive or design manually. For example, only for predicting ContinueStatement, the learned CondGen program for types uses 30 different leaves in its decision tree. One of the advantages of our approach is that despite the relatively large size of the model, its learned CondGen program can be easily inspected, interpreted and even manually modified by an expert, if needed.

Python py150 Dataset					
	Prior Work			**Our Work:** `Deep3`	
Prediction Task	PCFG	3-gram	`DeepSyn` [9]	ID3+	E13
Value prediction accuracy					
Unrestricted prediction	10.2%	63.9%	67.2%	63.7%	**69.2%**
Attribute accesses	0%	25%	**42%**	27%	**42%**
Numeric constants	22%	44%	40%	39%	**46%**
Names (variable, module)	17%	38%	38%	39%	**51%**
Function parameter names	40%	50%	50%	50%	**57%**
Type prediction accuracy					
Unrestricted prediction	59.0%	63.2%	72.5%	76.1%	**76.3%**
Function calls	55%	65%	71%	**74%**	**74%**
Assignment statements	29%	39%	61%	**67%**	66%
Return statements	0%	19%	10%	**41%**	29%
Lists	23%	46%	52%	**58%**	52%
Dictionaries	30%	52%	59%	**61%**	**61%**
`raise` statements	0%	18%	1%	**27%**	13%
Function parameters	54%	47%	70%	75%	**76%**

TABLE 2.6: Accuracy comparison for selected tasks between Python models used in prior work and in our technique.

An interesting observation we made by looking at the learned `CondGen` program for values is that the model learns sequences of instructions that perform traversals to a previous method invocation depending on whether the call target is a simple identifier, another call expression or field access. That is, our synthesized `CondGen` performs a form of lightweight program analysis and leads to improvements in the prediction accuracy. Such specialized sequences are at the moment learned for performing certain kinds of predictions in some branches of a `CondGen` program. An interesting future work item is to build a library of such automatically learned program analyses and investigate their applicability for building either more precise probabilistic models for code or for other problems in this space.

PREDICTION SPEED Even though the learned programs contain thousands of instructions, executing them is fast. This is due to the fact that for any prediction, only a small part of the instructions in a program needs to be executed (since execution traverses only a single branch of the decision tree). As a result, `Deep3` is capable of inferring around 15 000 queries per second on a single CPU core.

2.4.5 *Probabilistic Model for Python*

To build our probabilistic model for Python, we took `Deep3` and fed it with another training dataset of Python ASTs. This means that the only effort necessary for our models to handle that programming language was to provide a parser for Python and to download a large training dataset. We summarize the accuracy of `Deep3` in Table 2.6 and compare it to baseline models such as PCFG, n-gram language model and a model synthesized by the algorithm of `DeepSyn` [9]. Overall, the results for Python mimic the ones for JavaScript, but with some nuances:

- Similar to JavaScript, the best Python models learned by `Deep3` outperform the models from previous works. The ID3+ algorithm performs well for predicting the node types, but not as well for predicting node values. The E13 algorithm is the best algorithm for predicting node values.
- The precision of all Python models is lower than the precision of the corresponding model for JavaScript. One possible explanation is the different structure of the Python ASTs, which include less information that is redundant and easily predictable. This interpretation of the results is also supported by the lower precision of the PCFG model.

2.5 EVALUATION: CHARACTER LEVEL LANGUAGE MODELLING

In this section, we evaluate our proposed probabilistic model by instantiating it to the task of character level language modelling. The only change required to our whole approach is adjusting the domain-specific language with `Move` and `Write` instructions that operate over sequence of characters, as described in Section 2.1.3.2.

DATASETS We use two diverse datasets: a natural language one and a structured text (source code) one. Both were previously used to evaluate character-level language models – the *Linux Kernel* dataset [126] and *Hutter Prize Wikipedia* dataset [127]. The *Linux Kernel* dataset contains header and source files in the C language shuffled randomly, and consists of 6 206 996 characters in total with vocabulary size 101. The *Hutter Prize Wikipedia* dataset contains the contents of Wikipedia articles annotated with meta-data using special mark-up (e.g., XML or hyperlinks) and consists of 100 000 000 characters and vocabulary size 205. From both datasets, we use the first 80% for training, the next 10% for validation and the final 10% as a test set.

EVALUATION METRICS To evaluate the performance of various probabilistic language models we use two metrics. Firstly, we use the *bits-per-character* (BPC) metric which corresponds to the negative log likelihood of a given prediction $\mathbb{E}[-\log_2 p(y \mid \boldsymbol{x})]$, where y is character being predicted and $\boldsymbol{x}$ denotes all the characters preceding y in the input. Further, we use *error rate*, which corresponds to the ratio of mistakes the model makes. This is a practical metric that directly

quantifies how useful the model is in a concrete task (e.g., auto-completion). As we will see, having two different evaluation metrics is beneficial, as better (lower) BPC does not always correspond to a better (lower) error rate.

2.5.1 *Language Models*

We compare the performance of our trained model, instantiated with the `CondGen` language, against two widely used language models – n-gram model and recurrent neural networks. For all models we consider character level modelling of the dataset at hand. That is, all models are trained by feeding the input data character by character, without any knowledge of higher level word or sentence boundaries.

N-GRAM We use the n-gram model as a baseline, as it has been traditionally the most widely used language model due to its simplicity, efficient training and fast sampling. We note that n-gram can be trivially expressed in the `CondGen` language as a program containing a sequence of `LEFT` and `WRITE` instructions. To deal with data sparseness we have explored various smoothing techniques including Witten-Bell interpolation smoothing [121] and modified Kneser-Ney smoothing [119, 120].

RECURRENT NEURAL NETWORKS We also compare to recurrent network language models shown to produce state-of-the-art performance in various natural language processing tasks. In particular, for the *Linux Kernel* dataset, we compare against a variant of recurrent neural networks with Long Short-Term Memory (LSTM) [95]. To train our models, we follow the experimental set-up and use the implementation of [126]. We initialize all parameters uniformly in range [-0.08, 0.08], use mini-batch stochastic gradient descent with batch size 50 and RMSProp [128] per-parameter adaptive update with base learning rate $2 \cdot 10^{-3}$ and decay 0.95. Further, the network is unrolled 100 time steps and we do not use dropout. Finally, the network is trained for 50 epochs (with early stopping based on a validation set) and the learning rate is decayed after 10 epochs by multiplying it with a factor of 0.95 each additional epoch. For the *Hutter Prize Wikipedia* dataset we compared to various other, more sophisticated models as reported by [129].

OUR MODEL For the purposes of this evaluation, we enhance our model described so far with two simple extensions:

- *Backoff*: where instead of learning a single program, we learn multiple programs and use backoff whenever the current program is not confident enough. Concretely, we extend the language with the following rule:

$$\texttt{EqualityProgram} \mid \texttt{EqualityProgram}\ \textbf{backoff}\ t;\ \texttt{EqualityProgram}$$

Linux Kernel Dataset [126]			
Model	Bits per Character	Error Rate	Model Size
LSTM (Layers×Hidden Size)			
2×128	2.31	40.1%	5 MB
2×256	2.15	37.9%	15 MB
2×512	2.05	38.1%	53 MB
***n*-gram**			
4-gram	2.49	47.4%	2 MB
7-gram	2.23	37.7%	24 MB
10-gram	2.32	36.2%	89 MB
15-gram	2.42	35.9%	283 MB
Our Work			
$\texttt{CondGen}_{\texttt{w/o cache\&backoff}}$	1.92	33.3%	17 MB
$\texttt{CondGen}_{\texttt{w/o backoff}}$	1.84	31.4%	19 MB
$\texttt{CondGen}_{\texttt{w/o cache}}$	1.75	28.0%	43 MB
`CondGen`	**1.53**	**23.5%**	45 MB

TABLE 2.7: Detailed comparison of LSTM, n-gram and our models on *Linux Kernel* dataset.

Then, for a program $f = f_1$ **`backoff`** $t;\ f_2$ the semantics are defined as:

$$P_f(y \mid f(\boldsymbol{x})) = \begin{cases} P_{f_1}(y \mid f_1(\boldsymbol{x})) & \text{if } \arg\max_{y' \in \mathbb{Y}} P_{f_1}(y' \mid f_1(\boldsymbol{x})) \geq t \\ P_{f_2}(y \mid f_2(\boldsymbol{x})) & \text{otherwise} \end{cases}$$

That is, we backoff to the next model if the probability of the most likely character according to the current model is less than a constant t.

- *Cache*: further, we also consider backoff to a cache model [130]. Concretely, the cache model is a model that is trained on the last k characters seen in the current input (and updated after each new character is read), rather than pre-trained on the training dataset.

In our experiments, we backoff the learned program to a 7-gram and 3-gram model and we use a cache size of 800 characters. The backoff thresholds t are selected by evaluating the model performance on the validation set. Finally, for the *Linux Kernel* dataset we manually include a `StateProgram` as a root that distinguishes between comments and code (illustrated in Section 2.1.2.2). The program learned for the *Linux Kernel* dataset contains ≈700 `BranchPrograms` and ≈2 200 `SimplePrograms` and has over 8 600 `Move` and `Write` instructions in total.

Hutter Prize Wikipedia Dataset [127]						
Metric	*n*-gram	DSL model	Stacked LSTM	MRNN	MI-LSTM	HM-LSTM†
	$n = 7$	Our Work	[131]	[132]	[133]	[129]
BPC	1.94	1.62	1.67	1.60	1.44	1.34

TABLE 2.8: Bits-per-character metric for various neural language models (as reported by [129]) achieved on *Hutter Prize Wikipedia* dataset where the `CondGen` model achieves competitive results. †Combines character and word level models.

2.5.2 *Model Performance*

We compare the performance of our model, n-gram and neural networks for the tasks of learning character level language models by discussing a number of relevant metrics shown in Table 2.7 and Table 2.8.

PRECISION We can see that as expected, the n-gram model performs worse in both BPC and error rate metrics. However, even though the best BPC is achieved for a 7-gram model, the error rate decreases up to 15-gram. This suggests that none of the smoothing techniques we tried can properly adjust to the data sparsity inherent in the higher order n-gram models. It is however possible that more advanced smoothing techniques such as one based on Pitman-Yor Processes [134] might address this issue. As our model uses the same smoothing techniques as n-grams, any improvement to smoothing is directly applicable to it.

As reported by [126], the LSTM model trained on the *Linux Kernel* dataset improves BPC over the n-gram. However, in our experiments this improvement did not translate to lower error rate. In contrast, our model is superior to n-gram and LSTM in all configurations, improving over the best other model in both evaluation metrics – decreasing BPC by over 0.5 and improving error rate by more than 12%.

For the *Hutter Prize Wikipedia* dataset, even though the dataset consists of natural language text and is much less structured than the *Linux Kernel*, our model is competitive with several neural network models. Similar to the results achieved on *Linux Kernel*, we expect the error rate of our model for *Hutter Prize Wikipedia* dataset, which is 30.5%, to be comparable to the error rate achieved by other models. However, this experiment shows that our model is less suitable for unstructured text such as the one found on Wikipedia.

TRAINING TIME Our model takes ≈ 8 hours to train. The majority of training time is spent in the synthesis of `SwitchPrograms` where one needs to consider a massive search space of possible programs from which the synthesis algorithm aims to find one that is approximately the best (e.g., for *Linux Dataset* the number of basic instructions is 108 which means that naive enumeration of programs up to

size 3 already leads to 108^3 different candidate programs). All of our experiments were performed on a machine with Intel(R) Xeon(R) CPU E5-2690 with 14 cores.

MODEL SIZE In Table 2.7 we include the size of all the trained models measured by the size in MB of the model parameters. The models have roughly the same size except for the n-gram models with high order, for which the size increases significantly. The reason why both the n-gram and our models are relatively small is that we use hash-based implementation for storing the prediction context. That is, in a 7-gram model the previous 6 characters are hashed into a single number. This significantly decreases the model size at the expense of some hash collisions.

INTERPRETING THE LEARNED PROGRAM By inspecting the synthesized program we identified interesting `SimplePrograms` building blocks such as the sequence `PREV_CHAR(␣) RIGHT WRITE_CHAR` that conditions on the first character of the current word, `PREV_CHAR(\n) WRITE_DIST` that conditions on the distance from the beginning of the line or `PREV_CHAR(_) LEFT WRITE_CHAR` that checks the preceding character of a previous underscore (useful for predicting variable names). These are examples of more specialized programs that are typically found in the branches of nested switches of a large `CondGen` program. The top level switch of the synthesized program uses the character before the predicted position (i.e. **`switch`** `LEFT WRITE_CHAR`) and handles separately cases such as newline, tabs, special characters (e.g., !#@.∗), upper-case characters and the rest.

2.6 RELATED WORK

We next survey some of the work that is most closely related to ours.

2.6.1 *Probabilistic Models of Code*

Recently, there has been an increased interest in building probabilistic models of code and using these probabilistic models for various prediction tasks. The existing techniques for building probabilistic models of code can be roughly split into four categories – n-gram models, model based on probabilistic grammars, log-bilinear models and neural models.

N-GRAM MODELS The most popular and widely used probabilistic model of code is the n-gram model, due to its simplicity and efficient learning (first explored by Hindle et. al., [61] for modelling source code). Although the model used by Hindle is based on a syntactic representation of the code (e.g., including tokens such as (,) or ;), it was a promising first step in finding regularities in natural code. Shortly after, various improvements of the representation over which the model is learned were proposed, including structural and data dependencies [63] and defining task specific abstractions [64, 135]. Additionally, to address some of the

n-gram limitations, several extensions were developed, such as modeling local and global context [136] or including topic models [63].

In comparison, our work defines a new probabilistic model that is more precise and more expressive than the n-gram model. In fact, the n-gram model is a special case with which our model can be instantiated, as it can be easily expressed by a `SimpleProgram`. At the same time, our model keeps many of the advantages of n-gram model by being fast to train and query. Further, we can easily change the parameterization of our model by varying the choice of the domain-specific language. Additionally, we note that we can incorporate any of the above n-gram extensions, as these do not require the underlying model to be n-gram. We illustrated this in our evaluation where we showed how our model can be easily extended with a cache and backoff.

PROBABILISTIC GRAMMARS Another line of work considers probabilistic grammars, typically a PCFG, with various extensions built on top. A close work from the domain of natural language processing are lexicalized grammars, such as those produced by annotating the non-terminals with summaries of decision sequences used to generate the tree so far [137]. Instead, in our work, we parametrize the production rules of the grammar and phrase the task of finding the best parametrization as an optimization problem.

Additionally, several approaches have been developed to improve the precision, specifically in the domain of modelling source code. To capture code idioms, [138] uses probabilistic tree substitution grammars which extend a PCFG with production rules that allow tree fragments on the right-hand side. Gvero et. al., [112] augment grammar production rules with additional semantic checks that allow picking local variables in the current scope. Similarly, to model code locality and code reuse, various extensions were proposed that incorporate the context of the already generated AST by extending a PCFG with traversal variables [139] and using adaptor grammars with a cache [140].

These extensions are applicable to PHOG and are mainly orthogonal to our work. Instead, we focus on a probabilistic model which can be used as a more precise basic building block that can replace the PCFG used in the above works.

LOG-BILINEAR MODELS An alternative to n-grams and probabilistic grammars is a log-bilinear model. This model is especially suited when we have a large set of features and the goal is to learn the optimal weights per feature that reveal which features are relevant for the prediction. Such features are either simply generated (e.g, previous 10 non-terminals and terminals [123]) or manually designed for a given task (e.g., 17 features relevant for name prediction [124]). Another related work proposes a language model of C# programs that uses a log-linear model that operates over abstract syntax tree [139].

In terms of the features (conditioning context), our work is as expressive as log-bilinear models, as we can easily encode any features via additional instructions in the `CondGen` language. Further, instead of supplying the features manually, we

learn these automatically, which is especially useful when training generic models of a full scale programming language. Concretely, instead of learning a linear combination over a small set of predefined features, our work enables learning from an exponentially larger set of features (specified using programs in the `CondGen` language). Finally, because the model of our work is described as a program, it is more interpretable, can also be understood, edited by a human and further tuned towards a specific application.

NEURAL MODELS Recently, applying deep learning has shown great success for many tasks including building probabilistic models of code. In our evaluation, we compare to a number of models developed *after* our work, including LSTM [27, 95, 125], Code2Seq [28] and Transformers [29, 116]. Even though these models are newer and more sophisticated, we have shown that our model still outperforms (or is comparable) to most neural models developed in the meantime and it remains the state-of-the-art non-neural model. However, we do note that the recent work `TravTrans` [29] (developed four years after our model), which combines transformers with the AST representation of programs, does outperform our model.

DISCRIMINATIVE LEARNING In addition to generative approaches for code modelling, there have also been several works that employ discriminative models including [22, 67]. Such approaches, however, do not provide valid probability distributions and require user specified feature functions whose weights are then learned. More importantly, these models also lead to less precise models as shown in our evaluation, where our model achieved lower error rate (by 5% to 10%) compared to models used by [22] trained on shallow features from [123]. Further, the feature functions can be difficult to discover manually, are designed only for a specific task the tool addresses, and cannot serve as a basis for a general purpose model of complex, rich programming languages such as JavaScript and Python. Further, the combination of weights and feature functions leads to models that are difficult to understand, debug and explain.

2.6.2 *Decision Tree Learning*

Decision trees are a well studied and widely used approach for learning classifiers. Among a large number of decision tree algorithms, notable ones include ID3 [118] and its successor C4.5 [141], along with various general purpose techniques such as bagging and random forests. Although decision trees are mostly used as a black box classification technique, it is also possible to extend them to reflect the domain knowledge available for the given task at hand. For example, in the context of learning program invariants [142], it is necessary that the decision tree classifies all the examples perfectly and includes domain knowledge in form of implication counter-examples.

In our work, we first show that decision tree algorithms are a good fit for learning programs in `CondGen` language and then we describe how to extend and adapt

the classic ID3 learning algorithm so that it takes advantage of the fact that the leafs do not necessary have to correspond to unconditioned maximum likelihood estimate but can be represented using probabilistic models conditioned on complex features (in our case, contexts).

2.6.3 *Program Synthesis*

Similar to the existing line of work in program synthesis, the output of our learning procedure is a program drawn from a domain-specific language. One of the main challenges for many existing program synthesis techniques [143–146] is scalability: it is still difficult to scale these approaches to the task of synthesizing large, practical programs. To address this challenge, several recent works attempt to exploit the structure of the particular task to be synthesized by using techniques such as hierarchical relational specifications [147] or shapes of independent components [148]. Such approaches allow for finding suitable decompositions easily, which in turn can significantly speed up the synthesis procedure. Another approach taken by Raza et. al. [149] proposes the use of compositional synthesis guided by examples. Finally, Kneuss et. al. [150] decompose the initial synthesis problem of discovering a recursive function into smaller subproblems.

The main difference between our work and program synthesis approaches is that these attempt to satisfy all of the provided input-output examples. This has implications on the scalability, as well as on the level of acceptable errors. Further, in our work, we consider a very different setting consisting of large and noisy datasets where our goal is to discover a decomposition *without relying on any domain-specific knowledge* on the shape of the underlying components and without using guidance from counter-examples. Finally, to deal with domain-specific languages that contain hundreds of instructions, we propose a simple extension that uses Markov chain Monte Carlo (MCMC) to sample from a large number of instructions and guides the learning towards discovering better programs.

2.7 SUMMARY

In this chapter, we proposed PHOG, a novel approach to building probabilistic models – by parametrizing the model with a learned program from a domain-specific language. The choice of expressing the language model as a program results in several advantages including easier interpretability, extensibility with new instructions and the fact one might be interested in learning the program, rather than the resulting language model (as shown in the next chapter). We demonstrated the broad applicability of our model by applying it to the tasks of learning probabilistic models of code, as well as character level language modelling. For code, we used the same approach without any modifications to train models for both JavaScript and Python, significantly improving upon prior works. Similarly,

for character language modelling we trained two models, one over programs (including comments) and second over Wikipedia articles.

The key insight of our work is a new decision tree based learning approach that: *(i)* discovers a suitable decomposition of the entire data set, *(ii)* learns the best conditioning context for each component, and *(iii)* allows usage of probabilistic models as leafs in the obtained decision tree. In our work, all three components are concisely represented as a loop-free program with branches and the underlying probabilistic model is a simple maximum estimator based on counting. However, our approach does not put any restrictions on the probabilistic model used and other models can be easily incorporated.

A particularly interesting class of models are the recent deep learning methods which have become de-factor standard models in many domains, including for code. Indeed, as we have shown in our evaluation, even though our instantiation of PHOG is competitive and sometimes also better than many neural based models (developed *after* our work), it does produce worse results than the latest state-of-the-art deep learning models. The main conceptual reason for this is that while our probabilistic model is based on discrete word representations and sparse input (similar to hard attention), neural models use continuous word representations with soft attention – making the neural models significantly more powerful. Further, even though our model is widely applicable, it is significantly more limited than deep learning models that can easily combine multiple input modalities such as images, text and programs in a single model. As an example, neural models can be easily adapted to a more complex task of predicting whole subtrees instead of single values, and has been show to achieve significantly better results than our pre-trained models [26].

As a result, a natural future work item is combining PHOG with neural networks as leafs, as predicates in the branches or by transferring some of the techniques by making them differentiable. At the same time, it is possible to explore deep learning as a way to improve the learning algorithm by: *(i)* training a neural network, *(ii)* explaining the network's predictions by computing which nodes are relevant for the prediction, and *(iii)* synthesizing a program that captures the relevant nodes. While we did not explore this particular combination for improving PHOG, we will show how similar idea is applied to improve the model robustness in Chapter 4, as well as in Chapter 5 for synthesizing interpretable programs that capture decision making of a neural policy.

3

LEARNING STATIC ANALYZERS

In Chapter 2, we presented a novel approach for building probabilistic language models by learning a program p from a domain-specific language $\mathcal{L}$ with the goal of minimizing the model loss on a dataset of samples $\mathbb{D}$, formalized as:

$$\text{[Chapter 2]} \qquad \underbrace{\arg\min_{p \in \mathcal{L}}}_{\text{learned program}} \mathbb{E}_{(\boldsymbol{x},y)\sim\mathbb{D}} \underbrace{\ell(\boldsymbol{x}, y; p)}_{\text{model loss}} \tag{3.1}$$

The key novelty was to *parametrize the model by the learned program*, where the model in Chapter 2 corresponds to a probabilistic language model and the program was used to accumulate parts of the input relevant for the current prediction.

THIS CHAPTER This chapter builds on the techniques presented so far but with a focus on the learned programs, instead of the resulting model. To be practically useful, we are interested in learning not any programs but programs that would otherwise have to be written manually by a domain expert. In particular, the goal of the system developed in this chapter is to help experts design static analyzers faster, by learning parts (programs) of the analyzer from data.

We selected static analysis as it is an important and fundamental method for automating program reasoning with a myriad of applications in verification, optimization and bug finding. At the same time, while the theory of static analysis is well understood, building an analyzer for a practical language is a highly non-trivial task, even for experts. This is because one has to address several conflicting goals, for example *(i)* the analysis must be scalable enough to handle realistic programs, *(ii)* be precise enough to not report too many false positives, *(iii)* handle tricky corner cases and specifics of the particular language (e.g., JavaScript), *(iv)* decide how to precisely model the effect of the environment (e.g., built-in and third party functions), and other concerns. Addressing all of these manually is difficult and can easily result in suboptimal static analyzers, hindering their adoption in practice.

PROBLEM STATEMENT Our goal is to develop an automated approach for creating static analyzers: instead of manually providing the various inference rules of the analyzer, the key idea is to learn these rules from a dataset of programs. We state our learning problem as follows: given a domain-specific language $\mathcal{L}$ for describing analysis rules (i.e., transfer functions, abstract transformers), a dataset $\mathbb{D}$ of programs in some programming language (e.g., JavaScript), and an abstraction

function α that defines how concrete behaviors are abstracted, the goal is to learn an analyzer $pa \in \mathcal{L}$ (i.e., the analysis rules) such that programs in $\mathbb{D}$ are analyzed as precisely as possible, subject to α. At a high level, this goal translates to the following optimization problem, which we explain in more detail next:

$$\text{[This Chapter]} \qquad \overbrace{\underset{pa \in \mathcal{L}}{\arg\min}}^{\text{learned analysis}} \mathbb{E}_{(\boldsymbol{x},y)\sim\mathbb{D}} \overbrace{\ell(\boldsymbol{x}, y; pa)}^{\text{precision}} \qquad (3.2)$$

$$\text{s.t.} \quad \forall(\boldsymbol{x}, y) \in \mathbb{D}, \underbrace{\forall\delta \subseteq \Delta(x)}_{\text{robustness}}. \underbrace{\alpha(y) \sqsubseteq pa(\boldsymbol{x} + \delta)}_{\text{soundness}}$$

KEY CHALLENGES While the optimization problem above shares some similarities to the one in Chapter 2, there are a number of key challenges specific to the fact that we are learning static analyzers:

- *Learning new rules ($pa \in \mathcal{L}$).* First, static analyzers are typically described via rules designed by experts (i.e., type inference rules, abstract transformers), while existing general machine learning techniques such as support vector machines and neural networks only produce weights over feature functions as output. If these existing techniques were directly applied to program analysis [22, 151], the result would be a (linear) combination of existing rules and no new interesting rules would be discovered. Instead, we introduce domain-specific languages for describing the analysis rules, and then learn such analysis rules (which determine the analyzer) over these languages. The main challenge is similar to traditional program synthesis – to design the language such that it is generic and expressive enough to include the necessary analysis rules, yet concise such that one can learn programs efficiently.

- *Robustness ($\delta \subseteq \Delta(x)$).* The second challenging problem is to avoid learning an analyzer that works well on the training data $\mathbb{D}$, but fails to generalize well to programs outside of $\mathbb{D}$. For static analyzers this is a natural, yet critical, requirement as they are expected to work correctly for all programs. This is in contrast to traditional machine learning techniques that are optimized for programs that are likely according to the training data $\mathbb{D}$. Further, standard techniques from statistical learning theory [152] such as *regularization* are insufficient for our purposes. The idea of regularization is that picking a simpler model minimizes the expected error rate on unseen data, but a simpler model also contradicts an important desired property of static analyzers to *correctly handle tricky corner cases*. We address this challenge via a counter-example guided learning procedure that leverages program semantics to generate new data (i.e., programs) for which the learned analysis produces incorrect results. Formally, this corresponds to developing an adversary that searches for the worst case modification of the original input $\boldsymbol{x} + \delta$, by applying a set of valid modifications $\delta \subseteq \Delta(x)$.

- *Soundness and Over-approximation* ($\alpha(y) \sqsubseteq pa(\boldsymbol{x})$). Third, when learning static analyzers we need to consider two important concepts – soundness and over-approximation. We say that a static analysis is sound if its results model all possible executions of the program under analysis. That is, any feasible program behaviour should be included as part of the result, denoted as $\alpha(y) \sqsubseteq pa(\boldsymbol{x})$ (here, α is an abstraction function introduced formally in Section 3.2). In other words, static analysis should never produce an incorrect result, only results that are less precise. To achieve this in practice, the analysis over-approximates the set of possible behaviours when it is either uncertain or in order to ensure that the computation is tractable. This is in contrast to traditional machine learning techniques which are probabilistic and frequently produce incorrect results. To address this challenge, we adapt the techniques presented in Chapter 2, by: *(i)* designing a loss function that captures the fact that the analysis has to be *sound* (unlike probabilistic models), and *(ii)* we support a way to *over-approximate* the result in cases where the most precise analysis does not exist or has not been found.

MAIN CONTRIBUTIONS Our main contributions are:

- A novel method for learning static analysis rules from a dataset of programs. The main insight is that we can express interesting rules of a static analysis via a general domain-specific language that allows traversing abstract syntax trees and accumulates values.

- A counter-example guided refinement loop that ensures that the analysis *generalizes* beyond the training data. We achieve this by designing a strong adversary that generates counter-examples (i.e., new programs) using both semantic preserving and non-semantic preserving program transformations.

- Two techniques that make the adversary significantly more efficient at generating counter-examples. In particular, we leverage the fact that the analysis *pa* is an interpretable program by: *(i)* partitioning the dataset into equivalence classes with respect to *pa*, and *(ii)* applying program modifications that trigger different *execution paths* when executing *pa*.

- An instantiation of our approach to the task of learning rules for allocation site and restricted points-to analysis for JavaScript code. These are practical and relevant problems because of the tricky language semantics and wide use of libraries. Interestingly, our system learned inference rules missed by manually crafted state-of-the-art tools, e.g., Facebook's Flow [65] and TAJS [153]. We contacted a developer from Flow's team who confirmed that extending Flow to cover the cases handled by our analysis is a highly requested feature.

OUTLINE We organize this chapter as follows. In Section 3.1 we give an overview of our approach and the challenges on a simple points-to analysis for JavaScript. In

Section 3.2 we discuss what it formally means for a learned analyzer to be correct. In Section 3.3 we present our learning algorithm, which extends the techniques presented in Chapter 2. In Section 3.4 we introduce the adversarial component which tests whether the learner analyzer is correct, and if not, returns a set of counter-examples. Then, in Section 3.5 and Section 3.6 we discuss the steps needed to instantiate our approach for learning points-to and allocation site analysis, respectively. In Section 3.7 we briefly discuss implementation details of our work, in particular, how the training datasets can be obtained automatically – without manual labelling and without access to already implemented analysis that we would like to learn. Next, in Section 3.8 we provide a detailed experimental evaluation of our approach, which includes examples of the learned analyzers. Finally, we describe the related work in Section 3.9 and provide a brief summary and discussion in Section 3.10.

3.1 OVERVIEW

We start by providing an intuitive explanation of our approach on a simple points-to analysis for JavaScript. Assume we are learning the analysis from one training data sample given in Figure 3.1 *(a)*. It consists of variables `a`, `b` and `b` is assigned a new object, where we use s_0 to denote the object identify (in our case, s_0 corresponds to the program label at which the object was allocated). Our goal is to learn that `a` may also point to s_0 due to the assignment `a = b`.

Points-to analysis is typically done by applying inference rules until fixpoint. An example of an inference rule modelling the effect of assignment is:

$$\frac{\texttt{VarPointsTo}(v_2, h) \qquad \texttt{Assignment}(v_1, v_2)}{\texttt{VarPointsTo}(v_1, h)} \;[\textsc{Assign}]$$

This rule essentially says that if variable v_2 is assigned to v_1 and v_2 may point to an object h, then the variable v_1 may also point to this object h.

DOMAIN SPECIFIC LANGUAGE (DSL) FOR ANALYSIS RULES Consider the following general shape of inference rules:

$$\frac{\texttt{VarPointsTo}(v_2, h) \qquad v_2 = f(v_1)}{\texttt{VarPointsTo}(v_1, h)} \;[\textsc{General}]$$

Here, the function f takes a program element (a variable) and returns another program element or $\top$. The rule says: use the function f to find a variable v_2 whose points-to set will be used to determine what v_1 points to. The ASSIGN rule is an instance of the GENERAL rule that can be implemented as the following function:

$$f_{\texttt{Assign}}(x) ::= \begin{array}{ll} y & \textbf{if there is } \texttt{Assignment:} x \leftarrow y \\ \top & \textbf{otherwise} \end{array}$$

```
var b = {}; // program label s0
a = b;
```

Expected points-to set
$\mathbb{D} = \{(\texttt{a} \to \{s_0\})\}$

(a) Training data

```
VarDeclaration:b
  ObjectExpression:{}
Assignment
  Identifier:a
  Identifier:b
```

(b) AST representation of *(a)*

$$f_{desired}(x) ::= \begin{cases} y & \textbf{if there is } \texttt{Assignment:}x \leftarrow y \\ y & \textbf{if there is } \texttt{VarDeclaration:x} \leftarrow y \\ \top & \textbf{otherwise} \end{cases}$$

$$f_{overfit}(x) ::= \begin{cases} y & \textbf{if } y \textbf{ is } \texttt{VarDeclaration:y} \textbf{ preceding } x \\ y & \textbf{if there is } \texttt{VarDeclaration:x} \leftarrow y \\ \top & \textbf{otherwise} \end{cases}$$

(c) Learned functions to resolve points-to queries from *(a)*

FIGURE 3.1: Two examples of learned programs for points-to analysis using the training data shown in *(a)*. While both $f_{desired}$ and $f_{overfit}$ work correctly on the dataset $\mathbb{D}$, only $f_{desired}$ generalizes to programs beyond those in $\mathbb{D}$.

Where `Assignment:`$x \leftarrow y$ denotes that the input variable x appears as the left hand side of an assignment $x = y$ in the program. The same function can however be also rewritten as follows:

$$f_{\texttt{Assign}}(x) ::= \quad \textbf{switch } \texttt{WRITE_POS UP WRITE_TYPE}$$
$$\textbf{case } \texttt{'1 Assignment'} \textbf{ then } \texttt{RIGHT}$$
$$\textbf{default } \top$$

where we first traverse the AST, shown in Figure 3.1 *(b)*, check if x is its first child (`WRITE_POS`) and whether the parent node of x is of type `Assignment` (`UP WRITE_TYPE`). If this is the case, that is, executing `WRITE_POS UP WRITE_TYPE` returns `1` and `Assignment`, then we return the right sibling of x. Otherwise $f_{\texttt{Assign}}$ returns $\top$.

Key Insight: interesting rules of a static analyzer can be expressed using a domain-specific language $\mathcal{L}$ with branches that allows traversing ASTs and accumulating values.

As can be seen from the shape of the function shown above, we can reuse the `CondGen` language presented in Chapter 2 to learn rules for points-to analysis. Just as importantly, this allows us to reuse the techniques for learning probabilistic models of code (parametrized by `CondGen`) and apply them also for the task of learning static analyzers.

THE OVERFITTING PROBLEM Unfortunately, applying the learning techniques from Chapter 2 naively will often lead to learning poor programs. To understand why this is the case, consider the two functions from Figure 3.1 *(c)*. Here, in addition to modelling assignments, the functions are also trained to handle the case

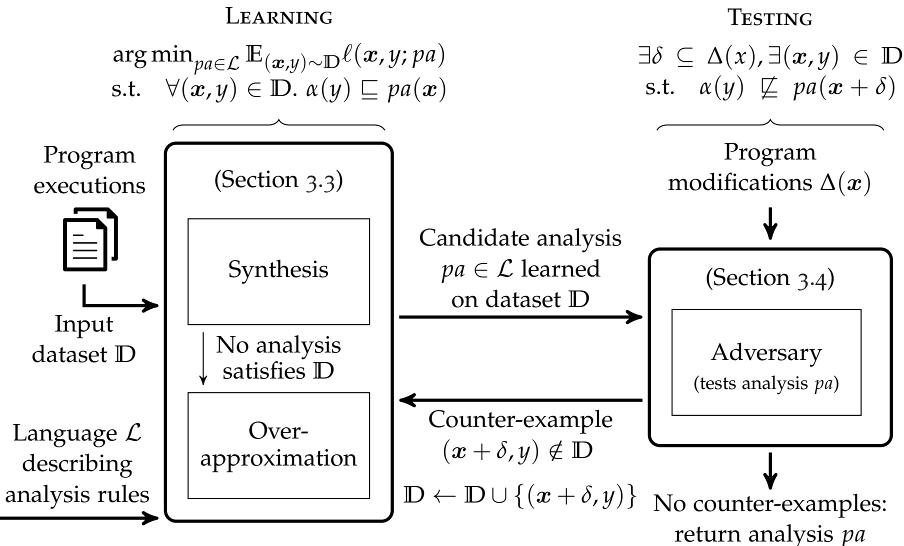


Figure 3.2: Overview of our approach to learning static analysis rules from data consisting of three components – a language $\mathcal{L}$ for describing the analysis rules, a learning algorithm and an adversary that tests the analysis – that interact in a counter-example based refinement loop.

of variable initialization (first line in the program). Because the dataset $\mathbb{D}$ typically does not determine a unique solution, both of these functions can be learned. However, while $f_{desired}$ implements the desired functionality and generalizes well, $f_{overfit}$ overfits to the training dataset. This is problematic as it leads to a non-robust model which at best is imprecise and at worst unsound. By inspecting $f_{overfit}$ we can see that it conditions on the statement prior to the current assignment instead of conditioning on the assignment itself, yet it succeeds to produce the correct analysis result on our dataset $\mathbb{D}$. While seemingly correct, this is due to the specific syntactic arrangement of statements in the training data $\mathbb{D}$ and will not generalize to other programs, beyond those in $\mathbb{D}$.

our solution To address the problem of overfitting and non-robustness, we propose a counter-example guided procedure that biases the learning towards semantically meaningful analyses, as illustrated in Figure 3.2. The inputs to the learning algorithm are a dataset $\mathbb{D}$ of programs with the ground-truth analysis results and a language $\mathcal{L}$ for describing the analysis rules. The goal of the learning is to find a sound and precise analysis that works on programs in $\mathbb{D}$, and approximates the results in cases where the most precise analysis cannot be learned. To ensure that the analysis also works on samples beyond those in the training dataset $\mathbb{D}$, we introduce an adversary and connect it with the synthesizer. This component takes as input the learned analysis pa and a set of valid program mod-

ifications $\delta \subseteq \Delta(\boldsymbol{x})$, and tries to find another program $\boldsymbol{x} + \delta$ for which *pa* fails to produce the desired result y. This counter-example $(\boldsymbol{x} + \delta, y)$ is then fed back to the synthesizer which uses it to generate a new candidate analyzer as illustrated in Figure 3.2.

Using the approach described above, a possible way to exclude $f_{overfit}$ is to insert an unnecessary statement (e.g., `var c = 1`) before the assignment `a = b` in Figure 3.1 *(a)*. Here, the analysis defined by $f_{overfit}$ produces an incorrect points-to set for variable `a` (as it points-to the value 1 of variable `c`). Once this sample is added to $\mathbb{D}$, $f_{overfit}$ is penalized and the next iteration will produce a different analysis until eventually the desired analysis $f_{desired}$ is returned.

COUNTER-EXAMPLE GUIDED LEARNING To learn a static analyzer *pa*, the synthesizer and the adversary are linked together in a counter-example guided loop. This type of iterative search is frequently used in program synthesis [144], though its instantiation heavily depends on the particular application task at hand. In our setting, the examples in $\mathbb{D}$ are programs (and not say program states) and we also deal with notions of (analysis) approximation. This also means that we cannot directly leverage off-the-shelf components (e.g., SMT solvers) or existing synthesis approaches. Importantly, the counter-example guided approach employed here is of interest to machine learning as it addresses the problem of overfitting with techniques beyond those typically used (e.g., regularization [152], which is insufficient here as it does not consider samples not in the training dataset).

RELATION TO ADVERSARIAL TRAINING Our approach presented in Figure 3.2 can be split into two separate steps – *(i)* learning an analysis *pa* that is sound and precise on a training dataset, and *(ii)* testing the correctness of the analysis *pa*, which produces a set of counter-examples. The two steps are performed iteratively by extending the original dataset with the counter-examples at each iteration, until no more counter-examples can be found.

Adversarial training [53] consists of the same two steps but used slightly differently – instead of iteratively extending the original dataset with the counter-examples, adversarial training discards the counter-examples found in previous iterations and only trains on the most recent ones. As we will see in our evaluation, the number of counter-examples found when learning static analyzers is roughly of the same magnitude as the original dataset. Therefore, we can easily afford to keep all of them and prevent the model from making the same mistakes in subsequent iterations. This is possible because in our work, one iteration of the learning algorithm (with programs) produces significantly better models compared to training a neural network for one epoch. Training neural models typically requires a large number of epochs to find a model with good performance. As a result, keeping counter-examples for all those epochs will become very expensive and make the training prohibitively slow. Even worse, most of the counter-examples will not be useful since they were obtained with poor models.

CORRECTNESS OF THE APPROACH Our method produces an analyzer that is guaranteed to be sound w.r.t to all of the examples in $\mathbb{D}$. Even if the analyzer cannot exactly satisfy all examples in $\mathbb{D}$, the synthesis procedure always returns an *over-approximation* of the desired outputs. That is, when it cannot match the target output exactly, it learns to over-approximate (e.g., can return $\top$ in some cases). The over-approximation can be seen as a measure of the model's confidence on its predictions, as lower confidence would result in larger approximations. A formal argument together with a discussion on these points is provided in Section 3.3. However, we note that our method is not guaranteed to be sound for all programs in the programming language. We see the problem of certifying the analyzer as orthogonal and complementary to our work: our method can be used to predict an analyzer that is likely correct, generalizes well, and to sift through millions of possibilities quickly, while a follow-up effort can examine this analyzer and decide whether to accept it or even fully verify it. Here, an advantage of our method is that the learned analyzer is expressed as a program, which can be easily examined by an expert, as opposed to standard machine learning models where interpreting the result is very hard and therefore difficult to verify with standard methods.

3.2 CHECKING ANALYZER CORRECTNESS

In this section, following [154], we briefly discuss what it means for a (learned) analyzer pa to be correct. The concrete semantics of a program p include all of p's concrete behaviors and are captured by a function $[\![p]\!]\colon \mathbb{N} \to \wp(\mathcal{C})$. This function associates a set of possible concrete states in $\mathcal{C}$ with each position in the program p, where a position can be a program counter or a node in the program's AST.

A static analysis pa of a program p computes an abstract representation of the program's concrete behaviors, captured by a function $pa(p)\colon \mathbb{N} \to \mathcal{A}$ where $(\mathcal{A}, \sqsubseteq)$ is an abstract domain, usually a lattice of abstract facts equipped with an ordering $\sqsubseteq$ between facts. An abstraction function $\alpha\colon \wp(\mathcal{C}) \to \mathcal{A}$ then establishes a connection between the concrete behaviors and the abstract facts. It defines how a set of concrete states in $\mathcal{C}$ is abstracted into an abstract element in $\mathcal{A}$. The abstraction function is naturally lifted to work point-wise on a set of positions in $\mathbb{N}$ (used in the definition below).

Definition 3.2.1 (Analysis Correctness)**.** A static analysis pa is correct if:

$$\forall p \in \mathcal{T}_{\mathcal{L}}.\ \alpha([\![p]\!]) \sqsubseteq pa(p) \tag{3.3}$$

Here $\mathcal{T}_{\mathcal{L}}$ denotes the set of all possible programs in the target programming language ($\mathcal{T}_{\mathcal{L}}$). That is, a static analysis is correct if it over-approximates the concrete behaviors of the program according to the particular lattice ordering.

CHECKING CORRECTNESS One approach for checking the correctness of an analyzer is to try to automatically verify the analyzer itself, that is, to prove that the analyzer satisfies Definition 3.2.1 via sophisticated reasoning (e.g., as the one

found in [155]). Unfortunately, such automated verifiers do not currently exist (though, coming up with one is an interesting research challenge) and even if they did exist, it is prohibitively expensive to place such a verifier in the middle of a counter-example learning loop where one has to discard thousands of candidate analyzers quickly. Thus, the correctness definition that we use in our approach is as follows:

Definition 3.2.2 (Analysis Correctness on a Dataset and Test Inputs)**.** A static analysis pa is correct w.r.t to a dataset of programs P and test inputs ti if:

$$\forall p \in P.\ \alpha(\llbracket p \rrbracket_{ti}) \sqsubseteq pa(p) \tag{3.4}$$

The restrictions over Definition 3.2.1 are: the use of a set $P \subseteq \mathcal{T}_{\mathcal{L}}$ instead of $\mathcal{T}_{\mathcal{L}}$ and $\llbracket p \rrbracket_{ti}$ instead of $\llbracket p \rrbracket$. Here, $\llbracket p \rrbracket_{ti} \subseteq \llbracket p \rrbracket$ denotes a subset of a program p's behaviors obtained after running the program on some set of test inputs ti.

The advantage of this definition is that we can automate its checking. We run the program p on its test inputs ti to obtain $\llbracket p \rrbracket_{ti}$ (a finite set of executions) and then apply the function α on the resulting set. To obtain $pa(p)$, we run the analyzer pa on p; finally, we compare the two results via the inclusion operator $\sqsubseteq$.

3.3 LEARNING ANALYSIS RULES

We now present our approach for learning static analysis rules from examples.

Let $\mathbb{D} = \{(\boldsymbol{x}_j, y_j)\}_{j=1}^n$ be a dataset of programs from a target language $\mathcal{T}_{\mathcal{L}}$ (i.e., JavaScript) together with outputs that a program analysis should satisfy. That is, $\boldsymbol{x}_j \in \mathcal{T}_{\mathcal{L}}$ and $y_j \in \mathcal{A}$ are the outputs to be satisfied by the learned program analysis.

Definition 3.3.1 (Analysis Correctness on Examples)**.** We say that a static analysis $pa \in \mathcal{L}$ is correct on $\mathbb{D} = \{(\boldsymbol{x}_j, y_j)\}_{j=1}^n$ if:

$$\forall (\boldsymbol{x}, y) \in \mathbb{D}\ .\ \ y \sqsubseteq pa(\boldsymbol{x}) \tag{3.5}$$

This definition is based on Definition 3.2.2, except that the result of the analysis is provided in $\mathbb{D}$ and need not be computed by running programs on test inputs.

Note that the definition above does not mention the precision of the analysis pa but is only concerned with soundness. To search for an analysis that is both sound and precise and avoids obvious, but useless solutions (e.g., always return the $\top$ element of the lattice $(\mathcal{A}, \sqsubseteq)$), we define a precision metric.

LOSS FUNCTION We define a loss function $\ell\colon \mathcal{T}_{\mathcal{L}} \times \mathcal{A} \times \mathcal{L} \to \mathbb{R}$ that takes a program in the target language, its desired program analysis output and a program analysis and computes a real-valued scored denoting the analysis quality:

$$\mathbb{E}_{(\boldsymbol{x},y)\sim\mathbb{D}}\ell(\boldsymbol{x}, y; pa) = \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x},y)\in\mathbb{D}} \begin{cases} 1 & \text{if} \quad y \not\sqsubseteq pa(\boldsymbol{x}) \\ 0 & \text{if} \quad y = pa(\boldsymbol{x}) \\ dist(y, pa(\boldsymbol{x})) & \text{otherwise} \end{cases} \tag{3.6}$$

Here, the loss is equal to one when the analysis is unsound ($y \not\sqsubseteq pa(\boldsymbol{x})$), the loss is zero if the analysis returns the most precise result ($y = pa(\boldsymbol{x})$) and otherwise, the loss is computed using a distance function $dist\colon \mathcal{A} \times \mathcal{A} \to \mathbb{R}^{(0,1)}$ that captures the analysis imprecision (i.e., the distance between y and $pa(\boldsymbol{x})$ in the lattice of abstract elements $(\mathcal{A}, \sqsubseteq)$). To remove clutter, we will also use the following notation $\ell(\mathbb{D}; pa) = \mathbb{E}_{(\boldsymbol{x},y)\sim\mathbb{D}}\ell(\boldsymbol{x}, y; pa)$ to denote the empirical loss on the dataset $\mathbb{D}$.

PROBLEM FORMULATION Given a language $\mathcal{L}$ that describes the analysis inference rules (i.e., abstract transformers) and a dataset $\mathbb{D}$ of programs with the desired analysis results, the learning should return a program analysis $pa \in \mathcal{L}$ such that:

1. pa is correct on the examples in $\mathbb{D}$ (Definition 3.3.1), and
2. the loss $\ell(\mathbb{D}, pa)$ is minimized (Equation 3.6).

The above statement essentially says that we would like to obtain a sound analysis which also minimizes the over-approximation that it makes. As the space of possible analyzers can be prohibitively large, we discuss a restriction on the language $\mathcal{L}$ and give a procedure that efficiently searches for an analyzer such that the correctness is enforced and the *cost* is (approximately) minimized.

LANGUAGE FOR DESCRIBING ANALYSIS RULES As illustrated in Section 3.1, to express interesting rules of a static analyzer, we reuse the loop-free `CondGen` language with branches from Chapter 2. The only difference is how the concrete `Move` and `Write` instructions are instantiated. We will describe the instantiation for points-to analysis in Section 3.5 and for allocation site analysis in Section 3.6.

LEARNING Because the language $\mathcal{L}$ has the same shape as `CondGen`, we can adapt the Algorithm 1 from Section 2.3. The resulting algorithm is shown in Algorithm 2 and contains three modifications:

- *Loss function*: we use the loss function from Equation 3.6.
- *Termination*: each recursive branch terminates either when the most precise program is found (line 2) or when the learning fails to find a `BranchProgram` that improves information gain (line 6).
- *Approximation*: when no further `BranchProgram` can be found, we approximate the results to ensure the analysis remains sound (line 7).

APPROXIMATION If the information gain is zero, we could not find any suitable predicate to split the dataset and the analysis p_{best} has non-zero cost. In this case, we define a function *approximate* that returns an approximate, but correct program analysis – in our implementation we return an analysis that loses precision by simply returning $\top$, which is always a correct analysis.

Algorithm 2: Algorithm for learning program analysis rules expressed in domain-specific language $\mathcal{L}$ from data.

```
def Learn (𝔻)
    Input: Dataset 𝔻 = {(x_j, y_j)}_{j=1}^n
    Output: Program pa ∈ ℒ
1   p_best ← syn_SimpleProgram(𝔻)                            ▷ Section 2.3.1
2   if ℓ(𝔻, p_best) = 0 then                   ▷ Precise analysis without branches
3       return p_best
4   p ← syn^E13_BranchProgram(p_best, 𝔻)                     ▷ Section 2.3.3
5   where p ≡ if (pred(x)) then p_a else p_b
6   if H(𝔻, p_best) − H(𝔻, p) == 0 then         ▷ No improvement over p_best
7       return approximate(𝔻)
8   p_a ← Learn({(x, y) ∈ 𝔻 | pred(x)})
9   p_b ← Learn({(x, y) ∈ 𝔻 | ¬pred(x)})
10  return p ≡ if (pred(x)) then p_a else p_b
```

In practice, this approximation does not return $\top$ for the entire analysis, but only for a few branches in the decision tree, for which the synthesis procedure fails to produce a good program using both $syn_{\texttt{SimpleProgram}}$ and $syn^{\texttt{E13}}_{\texttt{BranchProgram}}$.

In terms of guarantees, for Algorithm 2, we can state the following lemma.

Lemma 3.3.1. The analysis $pa \in \mathcal{L}$ returned by Algorithm 2 is correct according to Definition 3.3.1.

The proof of this lemma follows the definition of the algorithm and uses induction for the recursion. For our induction base case, if $\ell(\mathbb{D}, p_{best}) = 0$, the analysis is correct since by the definition from Equation 3.6, the loss is zero only if $y = pa(\boldsymbol{x})$. The analysis is also correct if *approximate* is called. In our induction step we use the fact that analyses p_a and p_b from the recursion are correct from which it follows that the composed analysis **`if`** $pred(\boldsymbol{x})$ **`then`** p_a **`else`** p_b is also correct.

3.4 THE ADVERSARY: TESTING AN ANALYZER

A key component of our approach is an adversary that can quickly test whether the current candidate analyzer is correct, and if not, to find a set of counter-examples. The adversary takes as an input a candidate analyzer pa and the dataset $\mathbb{D}$ used to learn pa and outputs a counter-example program on which pa behaves incorrectly.

Formally, finding a counter-example corresponds to solving the following satisfiability problem:

$$\exists \delta \subseteq \Delta(\boldsymbol{x}), \exists (\boldsymbol{x}, y) \in \mathbb{D} \quad \text{s.t.} \quad \alpha(y) \not\sqsubseteq pa(\boldsymbol{x} + \delta) \tag{3.7}$$

That is, the goal is to check whether there exists a sample in the given dataset $(\boldsymbol{x}, y) \in \mathbb{D}$ and a corresponding set of program transformations $\delta \subseteq \Delta(\boldsymbol{x})$, such

that running the analysis on a new program $\boldsymbol{x} + \delta$ leads to an incorrect $\alpha(y) \not\sqsubseteq pa(\boldsymbol{x} + \delta)$. Naturally, in practice we would solve the above satisfiability problem not only once but multiple times, thus obtaining a set of counter-examples.

KEY CHALLENGE A key problem the adversary must address is to *quickly* find a counter-example in the search space of all possible programs. This can be very challenging since the original dataset $\mathbb{D}$ can be very large and there can be infinitely many valid program changes $\Delta(\boldsymbol{x})$ to choose from. Furthermore, the problem is even harder than it might seem, since we are not applying a single change but a set of program changes $\delta \subseteq \Delta(\boldsymbol{x})$.

FINDING COUNTER-EXAMPLES EFFICIENTLY In our work, we develop an adversary that solves the satisfiability problem from Equation 3.7 by decomposing it into three subproblems – *(i)* selecting which sample to test $(\boldsymbol{x}, y) \in \mathbb{D}$, *(ii)* defining valid program transformations $\Delta(\boldsymbol{x})$ and *(iii)* selecting which of the valid transformations to apply $\delta \subseteq \Delta(\boldsymbol{x})$. For two of the subproblems, *(i)* and *(iii)*, the key for making efficient choices is that we leverage the information obtained by inspecting the analysis under test *pa*. In other words, the counter-example search is guided by the concrete analysis being tested.

3.4.1 *Choosing Relevant Samples to Test via Equivalence Classes*

As the first step, we need to determine which sample $(\boldsymbol{x}, y) \in \mathbb{D}$ from the dataset should be selected for testing. This is important since the training dataset is typically large and selecting samples at random will be hugely ineffective, especially if the candidate analysis *pa* is already very strong.

To address this issue, our adversary selects samples as follows:

- it partitions the dataset into equivalence classes with respect to *pa*, and
- it selects samples from each equivalence class

This ensures that samples are selected based on whether they are exploring different parts of the learned analysis and not proportional to their frequency in the dataset. In our case, the partitions correspond to execution paths of the candidate analysis *pa*. As a result, following the steps above guarantees that we select samples such that all paths are covered very early on during testing. Further, because the analysis is represented as a loop-free program with branches, obtaining the execution paths is straightforward.

3.4.2 *Defining Relevant Program Modifications*

We now define various program modifications that, when applied to an existing program, lead to potential counter-examples for the learned program analysis *pa*. We split the modifications into semantic preserving and non-semantic preserving.

Semantic Preserving Code Modifications for JavaScript	
Adding Code	
bool/string/num expressions	$\emptyset \rightarrow$ `expr`
unused variable declaration	$\emptyset \rightarrow$ `var x = expr`
anonymous function declaration	$\emptyset \rightarrow$ `(function(arg`$_1$`,...,arg`$_n$`){ expr })();`
Dead Code	
if statement	$\emptyset \rightarrow$ `if (expr) { dead_code }`
Renaming	
variables	`x` $\rightarrow$ `y`
user defined functions	`function inc(...)` $\rightarrow$ `function dec(...)`
function parameters	`function get(x)` $\rightarrow$ `function get(y)`

TABLE 3.1: Overview of semantic preserving program modifications used in our work.

SEMANTIC PRESERVING MODIFICATIONS Semantic preserving modifications are designed such that, as the name suggests, they modify the programs in a way that does not affect any of the concrete program behaviors. Further, not only they are semantic preserving with respect to the program, they are also designed to be semantic preserving with respect to the analysis *pa*. Formally, this means that all the semantic preserving modifications satisfy the following property:

$$\forall \delta \subseteq \Delta(\boldsymbol{x})^{\texttt{semantic}}_{\texttt{preserving}} \; . \;\; pa(\boldsymbol{x}) = pa(\boldsymbol{x} + \delta) \tag{3.8}$$

The intuition behind such modifications is to ensure stability by exploring *local program modifications*. If the adversary discovers a modification for which the above property is violated, the current analysis *pa* is incorrect and the counter-example program $\boldsymbol{x} + \delta$ is reported. The advantage of these modifications is that since they do not change the analysis results, the ground-truth labels can be directly reused.

We list the semantic preserving modifications used in our work in Table 3.1. These include various types of dead code insertion as well as variable, parameter and function name renamings. We use `expr` to denote random expression consisting of boolean, string and numeric constants, `x` and `y` to denote variable names and `arg` to denote argument names. As a concrete example, the modification `var x = expr` can be instantiated as `var p = 3 + 7` and denotes that we insert a statement with a fresh variable name with value equal to a randomly generated expression. Further, we use `dead_code` to denote any side-effect free code, i.e., we can generate and if statement where the body declares and invokes an anonymous function, declares a fresh variable or another nested if statement.

We note that which transformations are semantic preserving can vary depending on the kind of analysis being learned. For instance, inserting dead code that reuses existing program identifiers can affect flow-insensitive analysis, but should not affect a flow-sensitive analysis.

Non-Semantic Preserving Code Modifications for JavaScript	
Functions	
adding function parameters	`function get(x)` $\rightarrow$ `function get(x,arg`$_1$`,..., arg`$_n$`)`
adding method arguments	`obj.get(x)` $\rightarrow$ `obj.get(x,expr`$_1$`,..., expr`$_n$`)`
Changing Constants	
number substitution	`2` $\rightarrow$ `7`
string substitution	`"get"` $\rightarrow$ `"load"`
boolean substitution	`true` $\rightarrow$ `false`

Table 3.2: Overview of non-semantic preserving program modifications.

NON-SEMANTIC PRESERVING MODIFICATIONS To ensure better generalization, we are also interested in exploring changes to programs that may not be semantic preserving. This is useful as it allows us to discover new programs that exhibit behaviors further away from those seen in the training dataset.

We list the non-semantic preserving modifications used in our work in Table 3.2. Here the modifications include changing numeric, string and boolean constants as well as adding function parameters and method arguments. These modifications are useful since both points-to and allocation site analysis often depend on the number and type of the method arguments and function parameters. At the same time, they are prone to overfitting to common constants, especially in cases where the number of relevant samples is small (e.g., when handling corner cases).

Because these modifications do not preserve the program semantics, they typically also do not provide any guarantees that the analysis results remain the same. Therefore, for these modifications to be practically useful we also need a way to compute what the correct label should be *after* the modification is applied. In our work, we address this issue trivially by simply executing the modified program and observing the concrete program behaviours. This is possible since this is identical to how all our training datasets are obtained (we provide details in Section 3.7.1).

3.4.3 *Choosing Modification Positions*

Finally, having selected a program to test $\boldsymbol{x}$ and defined a set of valid program transformations $\Delta(\boldsymbol{x})$, the last step is to select which of the transformations to apply $\delta \subseteq \Delta(\boldsymbol{x})$. The key insight we use for selecting a good δ is observing that most of the modifications in $\Delta(\boldsymbol{x})$ have no effect on the analysis results. More importantly, the reason why they have no effect on the results is that they are changing parts of the program that are not even considered by the analysis when making a prediction. As an example, consider an intraprocedural analysis and a program with two functions – `A` and `B`. When analyzing function `B`, all transformations that affect function `A` will have no effect because we know that the analysis

is only intraprocedural (i.e., does not analysize any code outside of the current function). Therefore, an adversary equipped with such knowledge will only select modifications that modify the function under test.

With this intuition in mind, recall that the static analysis learned in our work comes from a domain-specific language that traverses abstract syntax trees (ASTs). For any given program, typically only a small subset of all the AST nodes determine the results of the analysis pa. We can compute exactly which these are by instrumenting the analyzer and recording all the visited AST nodes. Intuitively, if we were to change any node outside the set visited by the analysis, the results would remain unchanged (since the node was not visited as part of the execution). This gives us a simple, yet very effective, technique to prioritize positions accessed while executing the program analysis. Note however that we still allow changing all program positions, albeit with a much smaller probability. This is because we are selecting a set of modifications and not only a single one.

So far, we have described two main components of our approach, the learning that produces a candidate static analysis and the adversary that finds counter-examples for which the analysis does not yet work. In the next two sections, we will show how to instantiate our learning approach to two tasks – learning points-to and allocation site analysis for JavaScript.

3.5 POINTS-TO ANALYSIS

The goal of points-to analysis is to answer queries of the type $q\colon V \to \wp(H)$, where V is a set of program variables and H is a heap abstraction (e. g., allocation sites). That is, the goal is to compute the set of (abstract) objects to which a variable may point-to at runtime. Similar to the example illustrated in Section 3.1, to answer such queries a common line of work [156–158] uses a declarative approach where the program is abstracted as a set of facts and the analysis is defined declaratively (e. g., as a set of Datalog rules) using inference rules that are applied until a fixed point is reached.

OUR GOAL Our goal is to learn the inference rules that define the analysis, from data, as described in our approach so far. In particular, we would like to infer rules of the following general shape:

$$\frac{\texttt{VarPointsTo}(v_2, h) \quad v_2 = f(v_1)}{\texttt{VarPointsTo}(v_1, h)} \text{ [GENERAL]}$$

where the goal of learning is to find a set of functions f that, when used in the points-to analysis, produce precise results (as defined earlier). However, we focus our attention not on learning the standard and easy to define rules, like the one for assignment, but on rules that are tricky to model by hand and are missed by existing analyzers. In particular, consider the following subset of inference rules

$$H = \begin{array}{c} \top \\ h_1 \quad h_2 \quad \cdots \quad h_{n-1} \quad h_n \\ \bot \end{array}$$

FIGURE 3.3: Lattice of context-insensitive abstract heap locations H for points-to analysis.

that capture the points-to sets for the `this` variable in JavaScript. This rule has the following shape:

$$\frac{\texttt{VarPointsTo}(v_2, h) \quad v_2 = f(\texttt{this})}{\texttt{VarPointsTo}(\texttt{this}, h)} \text{ [THIS]}$$

which is an instantiation of the general rule for the `this` variable by setting $v_1 =$ `this`. In JavaScript, designing such rules is a challenging task as there are many corner cases and describing those precisely requires more inference rules than the rest of the (standard) analysis rules. Further, because assigning a value to `this` object is not allowed (i. e., using `this` as a left-hand side of an assignment expression), the value of `this` at runtime is not observed at the program level, yet assignments do occur internally in the interpreter and the runtime. Complicating matters, the actual values of the `this` reference can depend on the particular version of the interpreter.

3.5.1 *Instantiating our Learning Approach*

We now define the necessary components required to instantiate the learning approach described so far. Most of the instantiations are fairly direct except for the language $\mathcal{L}$.

LATTICE OF ABSTRACT HEAP LOCATIONS The lattice $(H, \sqsubseteq)$ used to represent the abstract domain of heap locations H is shown in Figure 3.3. The abstraction function $\alpha\colon O \to H$ maps the concrete objects seen at runtime to abstract heap locations represented using a context-insensitive allocation site abstraction H. The lattice is quite simple and consists of the standard elements $\top$, $\bot$ and elements corresponding to individual heap locations $h_1 \cdots h_n$ that are not comparable.

CONCRETE AND ABSTRACT PROGRAM SEMANTICS The concrete properties we are tracking and their abstract counterpart as described in Section 3.2 are instantiated by setting $\mathcal{C} := O$, $\mathcal{A} := H$ and $\mathbb{N} := \langle V, \mathbb{J}^* \rangle$. That is, all concrete program behaviors are captured by a function $[\![p]\!]\colon \langle V, \mathbb{J}^* \rangle \to \wp(O)$ that for each program variable V sensitive to the k-most recent call sites $\mathbb{J}$, computes a set of possible concrete objects seen at runtime O. The abstract semantics are similar except that we instantiate the abstract domain to be the lattice describing heap-allocated objects H. We will discuss how we obtain the concrete behaviors $[\![p]\!]_{ti}$ after running the program on a set of test inputs ti in Section 3.7.1.

$$
\begin{aligned}
\texttt{Move} &::= \texttt{Move}_{\texttt{core}} \cup \texttt{Move}_{\texttt{call}} \cup \texttt{Move}_{\texttt{js}} \\
\texttt{Move}_{\texttt{core}} &::= \texttt{UP} \mid \texttt{LEFT} \mid \texttt{RIGHT} \mid \texttt{DOWN_FIRST} \mid \texttt{DOWN_LAST} \mid \texttt{TOP} \\
\texttt{Move}_{\texttt{call}} &::= \texttt{GOTO_CALLER} \\
\texttt{Move}_{\texttt{js}} &::= \texttt{GOTO_GLOBAL} \mid \texttt{GOTO_UNDEF} \mid \texttt{GOTO_NULL} \mid \texttt{GOTO_THIS} \mid \texttt{UP_UNTIL_FUNC} \\
\texttt{Write} &::= \texttt{WRITE_VALUE} \mid \texttt{WRITE_POS} \mid \texttt{WRITE_TYPE} \mid \texttt{HAS_LEFT} \mid \texttt{HAS_RIGHT} \mid \texttt{HAS_CHILD}
\end{aligned}
$$

FIGURE 3.4: `Move` and `Write` instructions used to instantiate $\mathcal{L}_{pt}$ language for expressing the result of points-to query by means of traversing over ASTs.

3.5.2 *Language for Points-To Inference Rules*

Our main goal was to design a language $\mathcal{L}_{pt}$ that is fairly generic: *(i)* it does not require the designer to provide specific knowledge about the analysis rules, and *(ii)* the language can be used to describe rules beyond those of points-to analysis. The first point is especially important as specifying tricky parts of the analysis rules by hand requires substantial effort, which is exactly the process we would like to automate. Indeed, we aim at a language that is expressive enough to capture complex rules which use information from method arguments, fields, assignments, etc., yet can be automatically discovered during learning.

The main idea is to define $\mathcal{L}_{pt}$ to work over abstract syntax tree (AST) by providing means of traversing and conditioning on different parts of the tree. Further, we do not require the analysis to compute the results directly (e.g., a concrete points-to set for a given location). Instead, we allow the results to be specified indirectly by means of traversing to an AST position that determines the result. For example, such locations in the AST correspond to program positions with the same points-to set for points-to analysis, or to declaration sites for scope analysis or to program positions with the same type for type analysis.

SYNTAX We define $\mathcal{L}_{pt}$ in Figure 3.4 by instantiating the `Move` and `Write` instructions of the `CondGen` language defined in Chapter 2. We split the `Move` instructions into three groups where $\texttt{Move}_{\texttt{core}}$ includes language and analysis independent instructions that traverse over trees, $\texttt{Move}_{\texttt{js}}$ includes instructions that traverse to a set of interesting program locations that are specific to the JavaScript language, and $\texttt{Move}_{\texttt{call}}$ allows us to learn a call-site sensitive analysis.

SEMANTICS The semantics of executing programs in $\mathcal{L}_{pt}$ follows the semantics of `CondGen` programs defined in Section 2.1.3 with two changes:

- For cases when the `Move` instruction fails to execute or approximates the result, the remaining instructions are not executed. This is in contrast to the original semantics from Figure 2.11, which continues the execution without changing the position in the program. We introduce this change as it makes

$$\frac{m \in \texttt{Move} \qquad i' = mv(m, x, i) \qquad i' \notin \{\bot, \top\}}{\langle m :: s, x, i, ctx, counts \rangle \to \langle s, x, i', ctx, counts \rangle} \text{[MOVE]}$$

$$\frac{m \in \texttt{Move} \qquad i' = mv(m, x, i) \qquad i' \in \{\bot, \top\}}{\langle m :: s, x, i, ctx, counts \rangle \to \langle \epsilon, x, i, ctx, counts \rangle} \text{[MOVE-FAIL]}$$

FIGURE 3.5: Small-step semantics of `Move` instructions. We differentiate between the case where the `Move` instruction can be executed ([MOVE]) and when it fails ($\bot$) or approximates ($\top$) the result ([MOVE-FAIL]).

the learned programs easier to understand by domain experts. The corresponding small-step semantics are shown in Figure 3.5.

- The program state additionally includes the information about the k-most recent call sites. This does not affect the program semantics and only provides additional information used by the $\texttt{Move}_{\texttt{call}}$ instructions.

WRITE INSTRUCTIONS The semantics of the write instructions are described by the [WRITE] rule in Figure 2.11. Recall from Section 2.1.3 that each write accumulates a value c to the context ctx according to the function wr:

$$wr \colon \texttt{Write} \times \mathbb{X} \times \mathbb{N} \to \Sigma$$

The semantics of `WRITE_TYPE`, `WRITE_VALUE` and `WRITE_POS` remain unchanged. The semantics of the new instructions are defined as follows:

- $wr(\texttt{HAS_LEFT}, x, i)$ and $wr(\texttt{HAS_RIGHT}, x, i)$ return 1 if the node at position i has a left (right) sibling and 0 otherwise.

- $wr(\texttt{HAS_CHILD}, x, i)$ returns 1 if the node at position i has at least one child and 0 otherwise.

- $wr(\texttt{HAS_CALLER}, x, i)$ returns 1 if the current call trace is non-empty and 0 otherwise.

MOVE INSTRUCTIONS Move instructions are described by the [MOVE] and [MOVE-FAIL] rules in Figure 3.5 and use the function mv:

$$mv \colon \texttt{Move} \times \mathbb{X} \times \mathbb{N} \to \mathbb{N} \cup \{\bot, \top\}$$

The semantics of `UP`, `LEFT`, `RIGHT`, `DOWN_FIRST` and `DOWN_LAST` are as defined in Section 2.1.3 with one exception – when they cannot be executed, they return $\bot$. The semantics of the new instructions are defined as follows:

- $mv(\texttt{GOTO_GLOBAL}, x, i) = i'$, where i' is a node position corresponding to the `global` JavaScript object in the input x. For this and the other `GOTO` opera-

tions, the returned position i' is always unique and independent of the starting position i. That is, regardless of the position from which `GOTO_GLOBAL` is executed, it will point to the same `global` JavaScript object.

- $mv(\texttt{GOTO_THIS}, x, i) = i'$, where i' is a node position corresponding to the object to which `this` keyword points-to in the top-level scope. In a web browser this is the `window` object while in a `Node.js` application it is `module.exports`.
- $mv(\texttt{GOTO_UNDEF}, x, i) = i'$, where i' is a node position corresponding to the `undefined` JavaScript object in the input x. Similarly, for $mv(\texttt{GOTO_NULL}, x, i) = i'$, i' is the position corresponding to the `null` value.
- $mv(\texttt{GOTO_CALLER}, t, n, i \cdot i') = n' \times i'$, where n' is the node corresponding to call site of the top method i from call trace and i' is the call trace with the method i removed. If the call trace is empty then $n' = \perp$.
- $mv(\texttt{UP_UNTIL_FUNC}, x, i) = i'$ traverses recursively (using the `UP` instruction) to the position of the first node whose parent is a function declaration or the root of the tree is reached.
- $mv(\texttt{TOP}, x, i) = \top$ denotes that the analysis approximates the result to the $\top$ element in the lattice.

3.6 ALLOCATION SITE ANALYSIS

Next, we describe the instantiation of our approach to the task of learning allocation site analysis. The goal of allocation site analysis is to answer queries of the type $q \colon L \to \{true, false\}$, where L is a set of program locations. That is, for each program location, the analysis returns a boolean value denoting whether the location is an allocation site or not.

OUR GOAL Our goal for allocation site analysis is to learn inference rules from data in the following shape:

$$\frac{f(l) = true}{\texttt{AllocSite}(l)} \;[\textsc{Alloc}]$$

We illustrate the expected output and some of the complexities of allocation site analysis on a small example shown in Figure 3.6. The goal of the analysis is to determine all the program locations at which a new object is allocated. In JavaScript, there are various ways how an object can be allocated, some of which are shown in Figure 3.6. These include creating a new object without calling a constructor explicitly (for example by creating a new array or object expression inline), creating a new object by calling a constructor explicitly using `new`, creating a new object by calling a method or new objects created by throwing an exception. Further, some of the cases might also depend on the actual values passed as arguments. For example, calling a `newObject(obj)` constructor with `obj` as an argument does not create a new object but returns the `obj` passed as argument instead.

```
var obj = {a: 7};
var arr = [1, 2, 3, 4];
if (obj.a == arr.slice(0,2)) { ... }
var n = new Number(7);
var obj2 = new Object(obj);
try {  ...  } catch (err) {  ...  }
```

Allocation Sites
(new object allocated)

FIGURE 3.6: Illustration of program locations (highlighted in green) for which the allocation site analysis should report that a new object is allocated.

$$H_a = \begin{array}{c} \top \\ true \quad false \\ \bot \end{array}$$

FIGURE 3.7: Lattice used for allocation site analysis.

Next, let us consider the following simple, but unsound and imprecise allocation site analysis:

$$f_{alloc}(x) = \begin{cases} true & \textbf{if there is } \texttt{Argument:x} \textbf{ or } \texttt{NewExpression:x} \\ false & \textbf{otherwise} \end{cases} \quad (3.9)$$

which states that a location x is an allocation site if it is either an argument or a new expression. This analysis is imprecise because there are other ways to allocate an object (e.g., when creating arrays, strings, boxed values or by calling a function). It is also unsound, because the JavaScript compiler might not create a new object even when `NewExpression` is called (e.g., `newObject(obj)` returns the same object as the given `obj`). Instead of defining such tricky corner cases by hand, we use our approach to learn this analysis automatically from data, as described next.

3.6.1 *Instantiating our Learning Approach*

We now define the necessary components required to instantiate the learning approach described in our work.

ABSTRACT LATTICE Figure 3.7 shows the lattice $(H_a, \sqsubseteq)$ used to represent the abstract domain for allocation site analysis. The abstraction function $\alpha\colon L \rightarrow H_a$ maps the concrete program locations to the elements *true* and *false* which denote whether the program location is an allocation site or not.

CONCRETE AND ABSTRACT PROGRAM SEMANTICS The concrete properties we are tracking and their abstract counterpart as described in Section 3.2 are in-

$$\texttt{Move} ::= \texttt{UP} \mid \texttt{LEFT} \mid \texttt{RIGHT} \mid \texttt{DOWN_FIRST} \mid \texttt{DOWN_LAST} \mid \texttt{TOP}$$
$$\texttt{PREV_NODE_VALUE} \mid \texttt{PREV_NODE_TYPE}$$
$$\texttt{Write} ::= \texttt{WRITE_VALUE} \mid \texttt{WRITE_POS} \mid \texttt{WRITE_TYPE} \mid \texttt{HAS_LEFT} \mid \texttt{HAS_RIGHT} \mid \texttt{HAS_CHILD}$$
$$\texttt{HAS_PREV_NODE_TYPE} \mid \texttt{HAS_PREV_NODE_VALUE} \mid \texttt{NEW_ALLOC} \mid \texttt{NO_ALLOC}$$

FIGURE 3.8: `Move` and `Write` instructions used to instantiate the $\mathcal{L}_{alloc}$ language for expressing the result of allocation site query by means of traversing over ASTs.

stantiated by setting $\mathcal{C} := \{true, false\}$, $\mathcal{A} := H_a$ and $\mathbb{N} := L$, where L is a set of all program locations (nodes in an AST). That is, all concrete program behaviors are captured by a function $[\![p]\!]\colon \langle L\rangle \to \{true, false\}$ that for each program location L computes whether it is an allocation site. The abstract semantics are similar except that we instantiate the abstract domain to be the lattice $(H_a, \sqsubseteq)$. We will discuss how we obtain the concrete behaviors $[\![p]\!]_{ti}$ after running the program on a set of test inputs ti in Section 3.7.1.

LANGUAGE FOR ALLOCATION SITE ANALYSIS The DSL language $\mathcal{L}_{alloc}$ used to instantiate the learning of the allocation site analysis is very similar to the $\mathcal{L}_{pt}$ used for points-to analysis and `CondGen` used for learning probabilistic models of code. The syntax of the language is shown in Figure 3.8 and is based on the same idea of traversing over the abstract syntax tree of a given program. It contains four additional `Write` instructions:

- $wr(\texttt{HAS_PREV_NODE_TYPE}, x, i)$ returns 1 if the node at position i can successfully execute the instruction `PREV_NODE_TYPE` (i.e., there exists a node with the same type prior to position i) and 0 otherwise.
- $wr(\texttt{HAS_PREV_NODE_VALUE}, x, i)$ returns 1 if the node at position i can successfully execute the instruction `PREV_NODE_VALUE` and 0 otherwise.
- $wr(\texttt{NEW_ALLOC}, x, i)$ and $wr(\texttt{NO_ALLOC}, x, i)$ return a special value that denotes the analysis result, that is, whether a given location is an allocation site or not. These two instructions are used only in the leafs of the learned analysis.

3.7 IMPLEMENTATION

In this section we describe the implementation details of our approach.

OBTAINING TRAINING DATA Our learning approach uses the dataset of examples $\mathbb{D} = \{(\boldsymbol{x}_j, y_j)\}_{j=1}^{n}$ consisting of pairs $(\boldsymbol{x}_j, y_j)$ where $\boldsymbol{x}_j$ is a program (and a location in the program) and y_j is the desired output of the analysis when applied to $\boldsymbol{x}_j$. In general, obtaining such labeled training data for machine learning purposes is a tedious task. In our setting, however, this process can be automated because: *(i)* in static analysis, there is a well understood notion of correctness, namely, the

analyzer must *approximate* (in the sense of lattice ordering) the concrete program behaviors, and *(ii)* thus, we can simply run a large number of programs in a given programming language with some inputs, and obtain a subset of the concrete semantics for each program.

3.7.1 *Obtaining Concrete Behaviors of a Program*

We extract the relevant concrete behaviours $[\![p]\!]_{ti}$ of the program p by instrumenting the source code (not the interpreter) such that when executed, p produces a trace π consisting of all object reads, method entry points, method exit points and call sites. Additionally, at each method entry, we record the reads of all the parameters and the value of `this`. Further, every element in the trace contains a mapping to the location in the program (in our case to the corresponding node in the AST) and object reads record the unique identifier of the object being accessed.

TRAINING DATASET FOR POINTS-TO ANALYSIS Given such a trace π, we create a dataset $\mathbb{D}_{pt}$ used for points-to analysis by generating one input-output example for each position in the trace π at which `this` variable was read. Further, we select only the first read of `this` in each scope as all such references point to the same object.

TRAINING DATASET FOR ALLOCATION SITE ANALYSIS Given such a trace π, we create a dataset $\mathbb{D}_{alloc}$ used for the allocation site analysis by generating one input-output example for each position in the trace π as follows:

1. we select all the positions in the trace π where an object was read.
2. for each position we select only the first read in the trace, i. e., the first loop iteration or the first method invocation.
3. we filter reads of `this` object and field access.

From a trace π, we determine the correct label by assigning the label *true* to all the positions in π for which the corresponding identifier of the object being accessed was not seen previously within the same method call (or global scope) and $false$ otherwise. That is, intuitively we say that a program location is an allocation site if the object being read was not seen before. We consider method call boundaries to make the analysis modular and independent of the current program call trace.

3.7.2 *Checking Analysis Correctness*

POINTS-TO ANALYSIS For a program analysis *pa* and a dataset $\mathbb{D}$, we are interested in checking whether the analysis results computed for the program p are correct with respect to the concrete values seen during the execution of p. Given a

training example $(x, y) \in \mathbb{D}$, recall that executing the program analysis $i = pa(x)$ on the input x results in a position i in the input program (via tree traversal using `Move` instructions) or $\top$ (in case the analysis approximates the result). If the analysis returns $\top$ then it is trivially correct, otherwise we distinguish between two cases. If i corresponds to the original position, we say that the analysis is correct if the value has not been seen in the trace π before position i. This is true when position i is a new allocation site. Otherwise, we say that the analysis is correct if the value o has been seen previously in the trace at position i. Note that there might be multiple positions in the program that all point to the same result. Therefore, it is important that we consider all of them as correct.

ALLOCATION SITE ANALYSIS Checking the correctness of the allocation site analysis is trivial as executing the analysis $pa \in \mathcal{L}_{alloc}$ on an input example produces one of the labels `NewAlloc`, `NoAlloc` or `Top` which can be directly compared to the expected output $y \in \{true, false\}$.

3.7.3 *JavaScript Restrictions*

Finally, we remove from the training data programs that use the `eval` function, dynamic function binding using `Function.prototype.bind` and the `Function` object constructor. These are language features that require a combination of analyses to be handled precisely and are, therefore, typically ignored by static analyzers [159]. We also filter accesses to `arguments` object for the allocation site analysis. This is a limitation of our instrumentation that instruments reads and methods calls by means of wrapper functions that affect the binding of `arguments` object.

3.8 EVALUATION

In this section, we provide a detailed experimental evaluation instantiated to two practical analysis problems for JavaScript – learning points-to analysis rules and learning allocation site rules. In our experiments, we show that:

- The approach can learn program analysis rules for tricky cases involving JavaScript's built-in objects. These rules cover interesting corner cases that even existing state-of-the-art analyzers handle only partially.
- The counter-example based learning is critical for ensuring that the learned analysis generalizes well and does not overfit to the training dataset.
- Our adversary for generating programs to test the analysis is an order of magnitude more efficient at finding counter-examples than naively modifying the programs in the dataset $\mathbb{D}$.

These experiments were performed on a 28 core machine with 2.60Ghz Intel(R) Xeon(R) CPU E5-2690 v4 CPU, running Ubuntu 16.04. In our implementation, we parallelized both the learning and the search for the counter-examples.

```
global.length = 4;
var dat = [5, 3, 9, 1];
function isBig(value) {
  return value >= this.length;
}
```

```
// this points to global
dat.filter(isBig); // [5, 9]
// this points to boxed 42
dat.filter(isBig, 42); // []
// this points to dat object
dat.filter(isBig, dat); // [5, 9]
```

FIGURE 3.9: JavaScript code snippet illustrating a subset of the objects to which `this` can point to depending on the context in which the method `isBig` is invoked.

TRAINING DATASET We use the official ECMAScript (ECMA-262) conformance suite (`https://github.com/tc39/test262`) – the largest and most comprehensive test suite available for JavaScript containing over 20 000 test cases. As the suite also includes the latest version of the standard, all existing implementations typically support only a subset of the testcases. In particular, the `NodeJS` interpreter v4.2.6 used in our evaluation can execute (i. e., does not throw a syntax error) 15 675 tests which we use as the training dataset for learning.

3.8.1 *Learning Points-to Analysis Rules for JavaScript*

We now evaluate the effectiveness of our approach for the task of learning a points-to analysis for the JavaScript built-in APIs that affect the binding of `this`. This is useful because existing analyzers either model this only partially [65, 157] (i. e., by covering a subset of the behaviors of `Function.prototype` APIs) or not at all [158, 160], resulting in potentially unsound results.

We illustrate some of the complexity of determining the objects to which `this` points to within the same method in Figure 3.9. Here, `this` points to different objects depending on how the method is invoked and what values are passed in as arguments. In addition to the values shown in the example, other values may be seen during runtime if other APIs are invoked, or the method `isBig` is used as an object method or as a global method.

LEARNED ANALYZER A summary of our learned analyzer is shown in Table 3.3. For each API we collected all its usages in the ECMA-262 conformance suite, ranging from only 6 to more than 600, and used them as the initial training dataset for learning. In all the cases, a significant amount of counter-examples were needed to refine the analysis and prevent overfitting to the initial dataset. On average, for each API, the learning finished in 14 minutes, out of which 4 minutes were used to synthesise the program analysis and 10 minutes used in the search for counter-examples (cumulatively across all refinement iterations). The longest learning time was 57 minutes for the `Function.prototype.call` API for which we also learn the most complex analysis – containing 97 instructions in $\mathcal{L}_{pt}$. We note that even though the APIs in `Array.prototype` have very similar semantics, the learned programs vary slightly. This is caused by the fact that a different number and type

JavaScript Points-to Analysis			
Function Name	Dataset Size	Counter-examples Found	Analysis Size*
Function.prototype			
`call`	26	372	97 (18)
`apply`	6	182	54 (10)
Array.prototype			
`map`	315	64	36 (6)
`some`	229	82	36 (6)
`forEach`	604	177	35 (5)
`every`	338	31	36 (6)
`filter`	408	76	38 (6)
`find`	53	73	36 (6)
`findIndex`	51	96	28 (6)
Array			
`from`	32	160	57 (7)
JSON			
`stringify`	18	55	9 (2)

* Number of instructions in $\mathcal{L}_{pt}$ (Number of branches)

TABLE 3.3: Dataset size, number of counter-examples found and the size of the learned points-to analysis for JavaScript APIs that affect the points-to set of `this`.

of examples were available as the initial training dataset, which means that the adversary had to find different types of counter-examples.

To illustrate the complexity of the learned program analysis and the fact that it is easy for it to be interpreted by a human expert, we show the learned analysis for the API `Array.prototype.filter` in Figure 3.10. By inspecting the programs in the branches we can see that the analysis learns three different locations in the program to which the `this` object can point-to: the `global` object, a newly allocated object, or the second argument provided to the `filter` function. The analysis also learns the conditions determining which location to select. For example, `this` points to a new allocation site only if the second argument is a primitive value, in which case it is boxed by the interpreter. Similarly, `this` points-to the second argument (if one is provided), except for cases where the second argument is `null` or `undefined`.

For better readability, we replaced the sequence of instructions in $\mathcal{L}_{pt}$ used as branch conditions and branch targets with their informal descriptions. For example, the learned sequence that denotes the second argument of the calling method is `GOTO_CALLER DOWN_FIRST RIGHT RIGHT`. It is important to note that we were not

```
Array.prototype.filter ::=
  if caller has one argument
    points-to global object
  else if 2nd argument is Identifier
    if 2nd argument is undefined
      points-to global object
    else
      points-to 2nd argument
  else if 2nd argument is this
    points-to 2nd argument
  else if 2nd argument is null
    points-to global object
  else // 2nd argument is a primitive value
    points-to fresh allocation site
```

FIGURE 3.10: Learned points-to analysis for JavaScript API `Array.prototype.filter`. For better readability, we replaced the sequence of `Move` and `Write` instructions from $\mathcal{L}_{pt}$ with their informal description.

required to manually provide any such sequences in the language but that the learning algorithm discovered such relevant sequences automatically.

3.8.2 *Learning Allocation Site Analysis for JavaScript*

Next, we evaluate the effectiveness of our approach on a second analysis task – learning allocation sites in JavaScript. This is an analysis that is used internally by many existing analyzers. The analysis computes which statements or expressions in a given language result in an allocation of a new heap object. For an illustration of the expected output and some of the complexities of allocation site analysis, we refer the reader to the example presented earlier in Figure 3.6.

For this task, we obtain 134 721 input-output examples from the training data, which are further expanded with additional 905 counter-examples found during 99 refinement iterations of the learning algorithm. For this number of examples (much higher than in the other analyzer), the synthesis time was 184 minutes while the total time required to find counter-examples was 7 hours. The learned program is relatively complex and contains 135 learned branches, including the tricky case where `NewExpression` does not allocate a new object. Compared to the trivial, but wrong analysis f_{alloc} from Equation 3.9, the synthesized analysis marks over twice as many locations in the code as allocation sites ($\approx$ 21 000 vs $\approx$ 45 000).

PROGRAM LEARNED FOR OBJECT ALLOCATION USING `NewExpression` As illustrated in Figure 3.6, calling `new` in a JavaScript program does not necessarily

```
if WriteType == NewAllocation
  if constructor for given object was used before
    NewAlloc
  else if last argument is LiteralNumber
    NewAlloc
  else if last argument is LiteralString
    NewAlloc
  else if constructor with no arguments
    NewAlloc
  else if last argument is LiteralBoolean
    NewAlloc
  else if last argument is UnaryExpression
    NewAlloc
  else if last argument is ArrayExpression
    NewAlloc
  else if last argument is null
    NewAlloc
  else
    if last argument has been used before
      Top
    else
      Top
```

FIGURE 3.11: Learned analysis for object allocation by invoking the constructor explicitly.

lead to the allocation of a new object. The exception are the semantics of the built-in `Object` class that are defined as follows[1]:

> *"The `Object` constructor creates an object wrapper for the given value. If the value is `null` or `undefined`, it will create and return an empty object, otherwise, it will return an object of a Type that corresponds to the given value. If the value is an object already, it will return the value."*
>
> — `Object constructor`

By inspecting the learned program shown in Figure 3.11 we can see that it learns the above semantics by checking the type of the argument passed to the constructor. If the argument is one of the primitive types or a `null` value, then it will be always wrapped to a new object (marked by returning `NEW_ALLOC` as the leaf program). The program also learns that if the constructor has no arguments, then it always allocates a new object. Further, if the argument was used before, then the analysis chooses to conservatively approximate the result. Note that the program shown in Figure 3.11 corresponds to only a small part of the full analysis learned, as there are many more constructs that can lead to new object allocations.

1 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object

	Programs explored until first counter-example is found	
Difficulty	Random Modifications	Guided by Analysis
Easy ($\approx 60\%$)	146	13
Hard ($\approx 40\%$)	> 3000	130

Table 3.4: The effect of using the learned analysis to guide the counter-example search.

3.8.3 *Analysis Generalization*

We study how well the learned program for points-to analysis works for unseen data. First, we manually inspected the learned analysis at the first iteration of the learning procedure (without any counter-examples generated). We did that to check if we overfit to the initial dataset and found that indeed, the initial analysis would *not* generalize to some programs outside the provided dataset. This happened because the learned rules conditioned on unrelated regularities found in the data (such as variable names or fixed positions of certain function parameters). Our adversary, and the counter-example learning procedure, however, eliminate such kinds of non-semantic analyses by introducing additional function arguments and statements in the test cases.

Overfitting to the initial dataset was also caused by the large search space of possible programs in the DSL for the analysis. However, we decided not to restrict the language, because a more expressive language means more automation. Also, we did not need to provide upfront partial analysis in the form of a sketch [144].

ADVERSARY EFFECTIVENESS FOR FINDING COUNTER-EXAMPLES We evaluate the effectiveness of our adversary to find counter-examples by comparing it to a random adversary that applies all possible modifications to a randomly selected program from the training dataset. For both adversaries, we measure the average number of programs explored before a counter-example is found and summarize the results in Table 3.4. In the table, we observe two cases: *(i)* early in the analysis loop when the analysis is imprecise and finding a counter-example is *easy*, and *(ii)* later in the loop when *hard* corner cases are not yet covered by the analysis. In both cases, our *adversary guided by analysis* is an order of magnitude more efficient.

IS THE COUNTER-EXAMPLE REFINEMENT LOOP NEEDED? We also compare the effect of learning with a refinement loop to learning with a standard "one-shot" machine learning algorithm, but with more data provided up-front. For this experiment, we generate a dataset $\mathbb{D}_{huge}$ by applying all possible program modifications (as defined in Table 3.1 and Table 3.2) on all programs in $\mathbb{D}$. For comparison, let the dataset obtained at the end of the counter-example based algorithm on $\mathbb{D}$ be $\mathbb{D}_{ce}$. The size of $\mathbb{D}_{ce}$ is two orders of magnitude smaller than $\mathbb{D}_{huge}$.

An analysis that generalizes well should be sound and precise on both datasets $\mathbb{D}_{ce}$ and $\mathbb{D}_{huge}$, but since we use one of the datasets for training, we use the other one to validate the resulting analysis. For the analysis that is learned using counter-examples (from $\mathbb{D}_{ce}$), the precision is 99.9%, the fraction samples over-approximated to the top element of the lattice is 0.01%, and there are no unsound results. However, evaluating the analysis learned from $\mathbb{D}_{huge}$ on $\mathbb{D}_{ce}$ has precision of only 70.1%, with 0.08% of samples over-approximated to the top element of the lattice and the remaining 29.1% of the cases being *unsound*! This means that $\mathbb{D}_{ce}$ indeed contains interesting cases critical to the soundness and precision of the learned analysis.

SUMMARY Overall, our evaluation shows that the learning approach presented in our work can learn static analysis rules that handle various cases such as the ones that arise in JavaScript built-in APIs. The learned rules generalize to cases beyond the training data and can be inspected and integrated into existing static analyzers that miss some of these corner cases.

3.9 RELATED WORK

We next discuss prior works that are most closely related to the approach presented in this chapter.

SYNTHESIS FROM EXAMPLES Similar to our work, synthesis from examples typically starts with a domain-specific language (DSL) which captures a hypothesis space of possible programs together with a set of examples the program must satisfy and optionally an oracle to provide additional data points in the form of counter-examples using CEGIS-like techniques [144]. Examples from this research direction include discovery of bit manipulating programs [146], string processing [161], functional programs [162], or data structure specifications [163]. A recent work has shown how to generalize the setting to large and noisy datasets [9].

Other recent works [164, 165] synthesize models for library code by collecting program traces which are then used as a specification. The key differences with our approach are that we *(i)* use a large dataset covering hundreds of cases and *(ii)* we synthesize an analysis that generalizes beyond the provided dataset.

PROGRAM ANALYSIS AND MACHINE LEARNING Recently, several works applied machine learning in the domain of program analysis for tasks such as probabilistic type prediction [22, 166], reducing the false positives of an analysis [151], or as a way to speed up the analysis [167–169] by learning various strategies used by the analysis. We can think of the probabilistic type prediction approaches [22, 166] as being a kind of non-interpretable program analyzer that predicts types of variables with a certain probability. In the next chapter, we will investigate applying state-of-the-art deep learning techniques for the type inference task with the focus on the robustness of these models.

Other work connecting analysis and machine learning is that of Mangal *et al.* [151]. Here the authors start with the analysis rules already specified, in their case, in Datalog. Since the analysis tends to produce false positives, the authors then learn weights assigned to the rules which allow them to avoid deriving certain facts at the fixed point (and avoid some false positives). The follow-up work of Si *et al.* [170] also targets analyses written in Datalog with the goal of learning to compose in predefined rules into a Datalog program that performs well on the training dataset. To achieve this efficiently, the authors extend Datalog to the continuous setting where the learning is performed on a weighted combination of all possible programs and the best program is then discretized. Another recent work [167] explores the use of machine learning for analysis, this time however, the goal is to find a strategy for deciding which variables should be tracked flow insensitively. This approach aims to optimize the performance of the analysis by tracking fewer variables flow sensitively.

A key difference compared to our work is that we present a method to learn the static analysis rules which can then be applied in an iterative manner. This is a different task than [22, 166] which do not learn rules that can infer program specific properties and [167–169] which assume the rules are already provided and typically learn a classifier on top of them. Further, in our work, we learn static analysis rules expressed in a domain-specific language that allows traversing and writing values from trees. This is in contrast to approaches that assume the rules are given and consider the challenging task of composing them in a Datalog programs, either using machine learning techniques [151, 170] or using program synthesis techniques [171–173].

LEARNING INVARIANTS In an orthogonal effort there has also been work on learning program invariants using dynamic executions. For recent representative examples of this direction, we refer the reader to [142, 174]. The focus of all these works is rather different: they work on a per-program basis, exercising the program, obtaining observations and finally attempting to learn the invariants. Counter-example guided abstraction refinement (CEGAR) [175] is a classic approach for learning an abstraction (typically via refinement).

SCALABLE PROGRAM ANALYSIS Another line of work considers scaling program analysis in hard to analyse domains such as JavaScript at the expense of analysis soundness [158, 176]. These works are orthogonal to ours and follow the traditional way of designing the static analysis components by hand. An interesting future work, however, is applying the techniques presented in our work with the goal of automating parts of the manual design process that is currently required to develop such tools.

CHECKING CORRECTNESS OF THE STATIC ANALYSIS Finally, a related line of work addresses the problem of checking the correctness of existing static analyzers [177, 178]. At a high-level, this is exactly the same task that the adversary

in our work has to perform. Similar to our work, the correctness is assessed by generating new inputs or programs and checking whether the concrete program semantics (i.e., obtained by executing the new programs) satisfy the specification. The main difference to our work is the specification complexity; we use a generic specification that can be applied to a wide range of different analyzers and focus on the challenging task of learning the static analyzer itself. In contrast, the works [177, 178] assume that the analyzer is given, but focus on a much more complex specifications defined with respect to a concrete static analysis. As a result, the specification of [177, 178] can discover more errors, for example, by encoding properties that transformers of a numerical static analyzer needs to satisfy.

3.10 conclusion

In this chapter, we presented a new approach for learning static analyzers from examples. The key insight of our work is that: *(i)* we can express interesting static analysis rules using a domain-specific language that does not require the designer to provide specific knowledge about the analysis being learned, *(ii)* we can obtain the training dataset automatically, without having access to an existing static analysis that solves the task we are trying to learn, and *(iii)* we use a counter-example guided loop which iteratively tests the learned analysis and generates new programs for which the analysis fails.

The first point is especially important as specifying tricky parts of the analysis rules by hand requires substantial effort, which is exactly the process we would like to automate. Here, the main idea behind our domain-specific language is the same as in the previous chapter – to express programs that traverse over abstract syntax trees and accumulate values. This allows us to model rules of points-to analysis by means of traversing to the program position with the same points-to set, or for allocation site analysis by means of conditioning on the context relevant for deciding whether a new object is allocated or not. Similarly, as a future work one could apply the same ideas to learn type inference rules (by traversing to a program location with the same type) or call site analysis (by traversing to the method invocation).

However, at the same time, the domain-specific language is also the main limitation of our approach. This is because the most challenging part required developing a commercial grade program analysis tools, such as Facebook's Flow [65], is often designing the right abstractions suitable for the task at hand. As a concrete example, the main reason why Flow misses many of the rules learned by our approach is not because the Flow developers are not aware of some of these cases, but rather because incorporating them into the existing analysis would require significant design changes. Therefore, we see our work either as a technique for learning simple full analyses such as allocation site, or as a technique that helps experts design more complex static analyzers faster, by learning tricky parts and corner cases of the analysis from data.

Finally, in this chapter, we have used several steps required to learn analyzer rules that generalize well to program beyond those seen in the training dataset. In particular, we used concepts such as over-approximation, counter-example based learning instantiated with a set of program modifications or compositionality of the analysis where rules are applied until a fixpoint is reached. These techniques are of interest beyond our work as a way to train better and more robust models. In fact, we will see all of the above steps incorporate in the next chapter, with the exception that the technical solution will be very different – we will address a general class of neural models and do not assume any prior knowledge rather than decision tree learning over a domain-specific language.

4

ROBUSTNESS FOR MODELS OF CODE

In Chapter 3, we presented a new approach for learning static analyzers pa from a domain-specific language $\mathcal{L}$ and a training dataset $\mathbb{D}$, formalized as:

$$\text{[Chapter 3]} \qquad \overbrace{\arg\min_{pa \in \mathcal{L}}}^{\text{learned analysis}} \mathbb{E}_{(\boldsymbol{x},y)\sim\mathbb{D}} \overbrace{\ell(\boldsymbol{x}, y; pa)}^{\text{precision}} \qquad (4.1)$$
$$\text{s.t.} \quad \forall(\boldsymbol{x}, y) \in \mathbb{D}, \underbrace{\forall\delta \subseteq \Delta(x)}_{\text{robustness}}. \underbrace{\alpha(y) \sqsubseteq pa(\boldsymbol{x} + \delta)}_{\text{soundness}}$$

To achieve this, the two key contributions were: *(i)* the fact that we learn an interpretable program, which can be inspected and understood by a domain expert, and *(ii)* that we address the problem of learning robust and sound static analysis that generalize well to samples beyond those included in the dataset $\mathbb{D}$.

existing models of code For static analyzers, ensuring soundness and robustness is a natural, yet critical, requirement as they are expected to work correctly for all programs. This is in contrast to traditional machine learning techniques that typically solve the following optimization problem:

$$\begin{array}{c}\text{Accurate Models of Code} \\ \text{[Prior Works]}\end{array} \qquad \overbrace{\arg\min_{\theta}}^{\text{learned parameters}} \mathbb{E}_{(\boldsymbol{x},y)\sim\mathbb{D}} \overbrace{\ell(\boldsymbol{x}, y; \theta)}^{\text{precision}} \qquad (4.2)$$

That is, the only goal considered during learning is how well does the model perform on programs that are likely according to the dataset $\mathbb{D}$. In fact, both our work on learning probabilistic models of code[1] presented in Chapter 2, as well as recent deep learning models for code use a variant of the loss from Equation 4.2. This includes models applied to a wide range of tasks, including code completion [26, 27], code captioning [34–36], code classification [38, 39] and bug detection [41–43].

this chapter In this chapter, we address the challenging problem of training (adversarially) robust neural models for code. This is a very important problem shown to affect neural models in different domains [53–55], yet despite substantial progress on training accurate neural models of code, the issue of robustness for code has been overlooked. Similar to Chapter 3, we focus on tasks that compute

1 See the loss function from 2.5, which is the same as Equation 4.2, except for the regularization term.

program properties (e.g., type inference), usually addressed via handcrafted static analysis, but for which a number of recent neural models with high accuracy have been introduced [30, 31, 179]. Unsurprisingly, as these works do not consider robustness, their adversarial accuracy can drop significantly. However, training both *robust and accurate* models of code in this setting is non-trivial and requires one to address several key challenges:

- *Program Representation.* Programs are highly structured and long, containing hundreds of lines of code. This makes the task of training robust models very challenging since neural networks typically condition on the *full* input and any program modification (which there can be infinitely many) can potentially affect the result. To address this challenge, our goal is to:

  Enable neural networks to learn the parts of the program which are relevant for the prediction, without conditioning on the entire program.

- *Approximation.* Because the property prediction problem is usually undecidable, the static analyzers approximate the ideal solution. Naturally, the same should apply when using neural networks to solve the same prediction task. As a result, the challenge we address is to:

  Enable neural models to approximate the result when uncertain.

- *Compositionality.* Currently, there is a conceptual gap between the design of static analysis tools that typically perform a computation until a fixpoint is reached and neural networks that make predictions in constant time (even though the networks ca be very large). This is problematic, since making all the predictions at the same time results in models prone to learning statistical regularities around the predicted position rather than the actual rules. Our goal is to bridge this gap and to:

  Extend neural models such that they take advantage of learning compositional rules.

ACCURATE AND ROBUST MODELS OF CODE To address these challenges, we propose a novel method that combines three key components that lead to the following optimization problem:

ROBUST MODELS OF CODE [This Chapter]

$$\underbrace{\overbrace{\arg\min_{\theta,\alpha}}^{\text{learned parameters } \theta}}_{\text{learned abstraction } \alpha} \mathbb{E}_{(\boldsymbol{x},y)\sim\mathbb{D}} \underbrace{\max_{\delta\subseteq\Delta(\boldsymbol{x})}}_{\text{robustness}} \overbrace{\ell((f_\theta, \underbrace{g_\theta^h}_{\text{abstain}})(\alpha(\boldsymbol{x}+\delta)), y)}^{\text{precision}} \tag{4.3}$$

We illustrate the main components in Figure 4.1 and briefly describe them next (we provide more detailed description in Section 4.1). First, we train a model that *abstains* [180] from making a prediction when uncertain (denoted as function g_θ^h), effectively partitioning the dataset into two parts: one part where the model makes

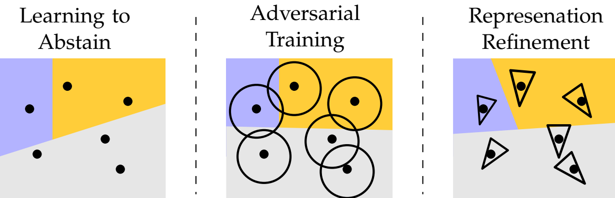


FIGURE 4.1: Illustration of the three key components used in our work. Each point represents a sample, ☐ is a region where the model abstains from making predictions, ☐ and ☐ are regions of model prediction, ◯ is the space of valid modifications for a given sample, and ▷ is the learned (reduced) space of valid modifications.

predictions (☐, ☐) that should be *accurate* and *robust*, and one (☐) where the model abstains and it is enough to be *robust*. Second, we instantiate *adversarial training* [53] to the domain of code, which corresponds to training using a worst case modification of the original inputs ($\max_{\delta \subseteq \Delta(\boldsymbol{x})}$). Third, we develop a new method to *refine the representation* used as input to the model by learning the parts of the program relevant for the prediction (denoted as the function α). This reduces the number of places that affect the prediction and helps to make adversarial training for code effective. Finally, we create a new algorithm that trains multiple models, each learning a specialized representation that makes robust predictions on a different subset of the dataset (not shown in Figure 4.1).

CONTRIBUTIONS Our main contributions are:

- We propose a novel combination of abstaining and robustness for neural models. To the best of our knowledge, this combination has not been explored apart from the concurrent work on training image classifiers [181].
- An instantiation of adversarial training [53] to models of code via a rich set of both semantic and label preserving program modifications.
- A new method that learns to refine (sparsify) the program representation used as input to the model.
- A new training algorithm that produces multiple models, each based on a specialized (learned) program representation necessary to make robust predictions on a different subset of the dataset.
- An implementation and thorough evaluation of our proposed system for the task of type inference. We show the effectiveness of our approach by successfully training a model that improves robustness by 15% while preserving high accuracy 87.7%. At the same time, extending the model with the ability to abstain allows us to train highly accurate and robust models *for a subset of the dataset*, with 99.9% accuracy and 99.9% robustness for 29% of the samples.

OUTLINE We organize this chapter as follows. In Section 4.1 we present an overview of our approach and all the key components on a running example. In Section 4.2 we briefly describe a recent method proposed by Liu et. al. [180] used in our work to extend neural models with the ability to abstain from making a prediction. In Section 4.3 we describe our instantiation of adversarial training. Next, we formally describe our novel contributions – learning to refine program representations (Section 4.4) and our training algorithm that combines all the components together by learning multiple specialized models (Section 4.5). In Section 4.6 we provide an experimental evaluation of our approach on five neural model architectures. Finally, we describe the related work in Section 4.7 and provide a brief summary with discussion in Section 4.8.

4.1 OVERVIEW: ACCURATE AND ROBUST MODELS OF CODE

In this section, we present an overview of our approach. Without loss of generality, we define an input program p to be a sequence of words $p = w_{1:n}$. The words can correspond to a tokenized version of the program, nodes in an abstract syntax tree corresponding to p or other suitable program representations. Further, let $l \in \mathbb{N}$ be a position in the program p that corresponds to a word $w_l \in \mathbb{W}$. A training dataset $\mathbb{D} = \{(\boldsymbol{x}_j, y_j)\}_{j=1}^n$ contains a set of samples, where $x \in \mathbb{X}$ is an input tuple $x = \langle p, l \rangle$ consisting of a program p and a position in the program l, while $y \in \mathbb{Y}$ contains the ground-truth label. As an example, the code snippet in Figure 4.2 contains 12 different samples (x, y), one for each position where a prediction should be made (annotated with their ground-truth types y).

Our goal is to learn a function $f\colon \mathbb{X} \to \mathbb{R}^{|\mathbb{Y}|}$, represented as a neural network, which for a given input program and a position in the program, computes the probability distribution over the labels. The model's prediction then corresponds to the label with the highest probability according to f.

TRAINING DATASET $\mathbb{D} = \{(\boldsymbol{x}_j, y_j)\}_{j=1}^n$

```
(hex^str, radix^num) => {
  v^num = parseInt^num(
    hex^str.substring^str(1^num),
    radix^num
  );
  red^num = v^num >>^num 16^num;
  ...
```

$\xrightarrow{\mathbb{D}}$

LEARNING TO ABSTAIN (SECTION 4.2)

```
(hex, radix) => {
  v^num,1.0 = parseInt^num,1.0(
      hex.substring^str,0.9(1^num,1.0),
      radix
    );
  red = v >> 16^num,1.0;
```

Learned selection function g_h that makes a prediction only if confident enough and abstains otherwise.

FIGURE 4.2: Illustration of learning to abstain. Even though the dataset contains 12 predictions, the model trained to abstain is confident in predicting only 5.

STEP 1: AUGMENT THE MODEL WITH AN (UN)CERTAINTY SCORE We start by augmenting the standard neural model f with an option to abstain from making a prediction. To achieve this, we adopt the recently proposed approach by [180] and introduce a selection function $g_h \colon \mathbb{X} \to \mathbb{R}$, which measures the model's certainty. Then, we define the model to make a prediction only if g_h is confident enough, in our work when $g_h(x) \geq h$, and abstain from making a prediction otherwise. Here, $h \in \mathbb{R}$ is an associated threshold that controls the desired level of confidence. For example, using a high threshold $h = 0.9$, the model learns to make only five predictions for the program in Figure 4.2 and will abstain from uncertain predictions such as predicting parameter types.

The first insight from our work is that allowing the model to abstain is beneficial for achieving robustness. This step leads to simpler models, since learning to abstain is easier than learning to predict the correct label. This is in contrast with forcing the model to learn the correct label for *all* samples, which is infeasible for most practical tasks.

LEARNING TO ABSTAIN (SECTION 4.2)

```
(hex, radix) => {
  v^{num,1.0} = parseInt^{num,1.0}(
      hex.substring^{str,0.9}(1^{num,1.0}),
      radix
    );
  red = v >> 16^{num,1.0};
```

$\xrightarrow{x+\delta}$

ADVERSARIAL TRAINING (SECTION 4.3)

```
(color^{num,0.9}, radix) => {
  v^{num,0.4} = parseInt^{num,0.6}(
      color.substring^{bool,0.9}(1^{num,1.0}),
      radix
    );
  red = v >> 16^{num,1.0};
```

$\delta = [$`rename hex` $\to$ `color`$]$

Trained with the worst case modifications $x + \delta$

FIGURE 4.3: Illustration of adversarial training which trains on the worst case modifications of the original samples.

STEP 2: ADVERSARIAL TRAINING Next, we instantiate adversarial training to the domain of code. Concretely, let $\Delta(x)$ be a set of valid modifications of the sample x and let $x + \boldsymbol{\delta}$ denote a new input obtained by applying the modifications in $\boldsymbol{\delta} \subseteq \Delta(x)$ to x. As a concrete example, Figure 4.3 shows a refactoring of the program from Figure 4.2 by renaming `hex` to `color`. Even though this change does not affect the types in the program, the model suddenly predicts incorrect types for both the `color` parameter and the `substring` function. Further, even though the types of `parseInt` and `v` are still correct, the model became much more uncertain.

Intuitively, our goal is to address this issue and to ensure that the model is robust for all valid modifications $\boldsymbol{\delta} \subseteq \Delta(x)$ – when evaluated on $x + \boldsymbol{\delta}$, the model either abstains or predicts the correct label. Concretely, we use adversarial training [53], which instead of minimizing the expected loss on the original distribution

$\mathbb{E}_{(x,y)\sim D}[\ell((f,g_h)(x),y)]$ as usually done in standard training, minimizes the expected *adversarial loss*:

$$\mathbb{E}_{(x,y)\sim D}[\max_{\boldsymbol{\delta}\subseteq\Delta(x)} \ell((f,g_h)(x+\boldsymbol{\delta}),y)] \tag{4.4}$$

That is, we minimize the worst case loss obtained by applying a valid modification to the original sample x. Similar to other domains, the main challenge in this setting is solving the inner $\max_{\boldsymbol{\delta}\subseteq\Delta(x)}$ efficiently for the domain of code.

STANDARD ADVERSARIAL TRAINING IS INSUFFICIENT Although adversarial training has been successfully applied in many domains [97–100], in our work we show that for code, adversarial training alone is insufficient to achieve model robustness. The key reason is that, existing neural models of code typically process *the entire program* which can contain hundreds of lines of code. This is problematic as it means that any program change will affect *all* predictions and there can be infinitely many program changes in $\Delta(x)$. Further, a single discrete program change is much more disruptive in affecting the model than a slight continuous perturbation of a pixel value. At the same time, in our evaluation we show that adversarial training, while not sufficient, can be used to improve robustness by 0% to 7%, depending on the model architecture.

FIGURE 4.4: Illustration of learned representation refinement that keeps the parts of the program relevant for prediction while abstracting the rest.

STEP 3: REPRESENTATION REFINEMENT To address the issue that adversarial training alone does not work well, we develop a novel technique that: *(i)* learns which parts of the input program are relevant for the given prediction, and *(ii)* refines the model representation such that only relevant program parts are used as input to the neural network. Essentially, the technique automatically learns an abstraction α which given a program, produces a relevant representation of that program. Figure 4.4 shows an example of a possible abstraction α that takes as input the entire program but keeps only the parts relevant for predicting the type of `parseInt` – it is a method call with name `parseInt` which has two arguments. To

learn the abstraction α, we first represent programs as graphs and then phrase the refinement task as an optimization problem that minimizes the number of graph edges, while ensuring that the accuracy of the model before and after applying α stays roughly the same.

Finally, we apply adversarial training, but this time on the abstraction α obtained via representation refinement, resulting in new functions f and g_h. Overall, this results in an adversarially robust model $m_i = \langle f, g_h, \alpha \rangle$.

Samples at Iteration i

```
(hex, radix) => {
  v = parseInt(
      hex.substring(1),
      radix
   );
  red = v >> 16;
```

apply model $m_i = \langle f, g_h, \alpha \rangle$ →

Samples at Iteration $i+1$

```
(hex, radix) => {
  v = parseInt<num>(
      hex.substring(1<num>),
      radix
   );
  red = v >> 16<num>;
```

Figure 4.5: Illustration of applying predictions produced by a robust model m_i directly to the dataset. As a result, the subsequent models can depend on these predictions rather than relearning them from scratch.

step 4: learning accurate models Although the model m_i is robust, it provides predictions only for a subset of the samples for which it has enough confidence (i.e., $g_h(x) \geq h$). To increase the ratio of samples for which our approach makes a prediction (i.e., does not abstain), we perform two steps: *(i)* generate a new dataset $\mathbb{D}_{i+1}$ by annotating the program with the predictions made by the learned model m_i (illustrated in Figure 4.5), and removing successfully predicted samples, and *(ii)* learn another model m_{i+1} on the new dataset $\mathbb{D}_{i+1}$. We repeat this process for as long as the new learned model predicts some of the samples in $\mathbb{D}_{i+1}$.

Training multiple models is beneficial because: *(i)* the models are easier to train, as well as easier to make robust, as they do not try to learn all predictions, *(ii)* it allows conditioning on the predictions learned by earlier models which helps both interpretability and robustness. For example, the model m_{i+1} can learn that the left hand side of the assignment `v = parseInt` has the same type as the right hand side, since the type of `parseInt` was already predicted by m_i. Interestingly, if we think of each model as a learned set of rules, we can essentially apply the models to a given program in a fixed point style (similar to how a traditional sound static analysis works), and *(iii)* each model learns a different representation α that is specialized for the predictions it makes. For example, while predicting the type of `parseInt` is independent of the argument values (`parseInt(_, _)`), predicting the second argument type is not (`parseInt(_, radix)`). Using a single abstraction to predict both would lead to either reduced robustness or accuracy, depending on which abstraction is used.

SUMMARY Given a training dataset $\mathbb{D}$, our approach learns a set of robust models, each of which makes robust predictions for a different subset of $\mathbb{D}$. To achieve this, we extend existing neural models of code with three key components – the ability to abstain (with associated uncertainty score), adversarial training, and learning to refine the representation. Next, we formally describe each of the components (in Sections 4.2, 4.3 and 4.4) and then present our training algorithm that combines all of them together (in Section 4.5).

4.2 TRAINING NEURAL MODELS TO ABSTAIN

We now present a method for training neural models of code that provide an uncertainty measure and can abstain from making predictions. This is important as essentially all practical tasks contain some examples for which it is not possible to make a correct prediction (e.g., due to the task difficulty or because it contains ambiguities). In the machine learning literature this problem is known as selective classification (supervised-learning with a reject option) and is an active area with several recently proposed approaches [180, 182–185]. In our work, we use one of these methods [180] which is briefly summarized below. For a full description, we refer the reader to the original paper [180].

Let $\mathbb{D} = \{(\boldsymbol{x}_j, y_j)\}_{j=1}^{n}$ be a training dataset and $f\colon \mathbb{X} \to \mathbb{Y}$ an existing model trained to make predictions on $\mathbb{D}$. The existing model f is augmented with an option to abstain from making a prediction by introducing a selection function $g_h\colon \mathbb{X} \to \mathbb{R}^{(0,1)}$ with an associated threshold $h \in \mathbb{R}^{(0,1)}$, which leads to the following definition:

$$(f, g_h)(x) := \begin{cases} f(x) & \text{if } g_h(x) \geq h \\ \texttt{abstain} & \text{otherwise} \end{cases} \tag{4.5}$$

That is, the model makes a prediction only if the selection function g_h is confident enough (i.e., $g_h(x) \geq h$) and abstains from making a prediction otherwise. Although conceptually the model is now defined by two functions f (the original model) and g_h (the selection function), it is possible to adapt the original classification problem such that a single function f' encodes both. To achieve this, an additional `abstain` label is introduced and a function $f'\colon \mathbb{X} \to \mathbb{Y} \cup \{\texttt{abstain}\}$ is trained in the same way as f (i.e., same network architecture, hyper-parameters, etc.) with two exceptions: *(i)* f' is allowed to predict the additional `abstain` label, and *(ii)* the loss function used to train f' is changed to account for the additional label. After f' is obtained, the selection function is defined as $g_h := 1 - f'(x)_{\texttt{abstain}}$, that is, the probability of selecting any label other than `abstain` according to f'. Then, f is defined to be the re-normalized probability distribution obtained by taking the distribution produced by f' and assigning zero probability to the `abstain` label. Essentially, as long as there is sufficient probability mass h on labels outside `abstain`, f decides to select one of these labels.

LOSS FUNCTION FOR ABSTAINING To gain an intuition behind the loss function used for training f', recall that the standard way to train neural networks is to use cross entropy loss:

$$\ell_{\texttt{CrossEntropy}}(\boldsymbol{p}, \boldsymbol{y}) := -\sum_{i=1}^{|\mathbb{Y}|} y_i \log(p_i) \tag{4.6}$$

Here, for a given sample $(x, y) \in \mathbb{D}$, $\boldsymbol{p} = f(x)$ is a vector of probabilities for each of the $|\mathbb{Y}|$ classes computed by the model and $\boldsymbol{y} \in \mathbb{R}^{|\mathbb{Y}|}$ is a vector of ground-truth probabilities. Without loss of generality, assume only a single label is correct, in which case $\boldsymbol{y}$ is a one-hot vector (i.e., $y_j = 1$ if j-th label is correct and zero elsewhere). Then, the cross entropy loss for an example where the j-label is correct is $-\log(p_j)$. Further, the loss is zero if the computed probability is $p_j = 1$ (i.e., $-\log(1) = 0$) and positive otherwise.

Now, to incorporate the additional `abstain` label, the abstain cross entropy loss is defined as follows:

$$\ell_{\texttt{AbstainCrossEntropy}}(\boldsymbol{p}, \boldsymbol{y}) := -\sum_{i=1}^{|\mathbb{Y}|} y_i \log(p_i o_i + p_{\texttt{abstain}}) \tag{4.7}$$

Here $\boldsymbol{p} \in \mathbb{R}^{|\mathbb{Y}|+1}$ is a distribution over the classes (including `abstain`), $o_i \in \mathbb{R}$ is a constant denoting the weight of the i-th label and $p_{\texttt{abstain}}$ is the probability assigned to `abstain`. Intuitively, the model either: *(i)* learns to make "safe" predictions by assigning the probability mass to $p_{\texttt{abstain}}$, in which case it incurs constant loss of $p_{\texttt{abstain}}$, or *(ii)* tries to predict the correct label, in which case it potentially incurs smaller loss if $p_i o_i > p_{\texttt{abstain}}$. If the scaling constant o_i is high, the model is encouraged to make predictions even if it is uncertain and potentially makes lot of mistakes. As o_i decreases, the model is penalized more and more for making mis-predictions and learns to make "safer" decisions by allocating more probability mass to the `abstain` label.

OBTAINING A MODEL WHICH NEVER MIS-PREDICTS ON THE DATASET $\mathbb{D}$ For the $\ell_{\texttt{AbstainCrossEntropy}}$ loss, it is possible to always obtain a model f' that never mis-predicts on samples in $\mathbb{D}$. Such a model f' corresponds to minimizing the loss incurred by Equation 4.7 which corresponds to maximizing $p_i o_i + p_{\texttt{abstain}}$ (assuming i is the correct label). This can be simplified and bounded from above to $p_i + p_{\texttt{abstain}} \leq 1$, by setting $o_i = 1$ and for any valid distribution it holds that $1 = \sum_{p_i \in \boldsymbol{p}} p_i$. Thus, $p_i o_i + p_{\texttt{abstain}}$ has a global optimum trivially obtained if $p_{\texttt{abstain}} = 1$ for all samples in $\mathbb{D}$. That is, the correctness (no mis-predictions) can be achieved by rejecting all samples in $\mathbb{D}$. However, this leads to zero recall and is not practically useful.

BALANCING CORRECTNESS AND RECALL To achieve both correctness and high recall, similar to Liu *et al.* [180], we train our models using a form of annealing. We start with a high $o_i = |\mathbb{Y}|$, biasing the model away from abstaining, and then train

for a number of epochs n. We then gradually decrease o_i to 1 for a fixed number of epochs k, slowly nudging it towards abstaining. Finally, we keep training with $o_i = 1$ until convergence. We note that the threshold h is not used during training. Instead, it is set after the model is trained and is used to fine-tune the trade-off between recall and correctness (precision). Further, note that $o_i = 1$ is used only if the desired accuracy is 100% and otherwise we use $o_i = 1 + \epsilon$. Here, ϵ is selected by decreasing the value o_i as before but stopping just before the model abstains from making all predictions.

SUMMARY We described an existing technique [180] for training a model that learns to abstain from making predictions, allowing us to trade-off correctness (precision) and recall. A key advantage of this technique is its generality – it works with any existing neural model with two simple changes: *(i)* adding an `abstain` label, and *(ii)* using the loss function in Equation 4.7. To remove clutter and keep discussion general, the rest of our work interchangeably uses $f(x)$ and $(f, g_h)(x)$.

4.3 ADVERSARIAL TRAINING FOR CODE

In Section 4.2, we described how to learn models that are correct on a subset of the training dataset $\mathbb{D}$ by allowing the model to abstain from making a prediction when uncertain. We now discuss how to achieve robustness (that is, the model either abstains or makes a correct prediction) for a much larger (potentially infinite) set of samples beyond those included in $\mathbb{D}$ via so-called *adversarial training* [53].

ADVERSARIAL TRAINING The goal of adversarial training [97–100] is to minimize the expected adversarial loss:

$$\mathbb{E}_{(x,y)\sim D}[\max_{\boldsymbol{\delta} \subseteq \Delta(x)} \ell(f(x + \boldsymbol{\delta}), y)] \tag{4.8}$$

In practice, as we have no access to the underlying distribution but only to the dataset $\mathbb{D}$, the expected adversarial loss is approximated by *adversarial risk* (which training aims to minimize):

$$\frac{1}{|\mathbb{D}|} \sum_{(x,y)\in\mathbb{D}}^{|\mathbb{D}|} \max_{\boldsymbol{\delta} \subseteq \Delta(x)} \ell(f(x + \boldsymbol{\delta}), y) \tag{4.9}$$

Intuitively, instead of training on the original samples in $\mathbb{D}$, we train on the worst perturbation of each sample. Here, $\boldsymbol{\delta} \subseteq \Delta(x)$ denotes an ordered sequence of modifications while $x + \boldsymbol{\delta}$ denotes a new input obtained by applying each modification $\delta \in \boldsymbol{\delta}$ to x. Recall that each input $x = \langle p, l \rangle$ is a tuple of a program p and a position l in that program for which we will make a prediction. Applying a modification $\delta \colon \mathbb{X} \to \mathbb{X}$ to an input x corresponds to generating both a new program as well as updating the position l if needed (e.g., in case the modification inserted or reordered program statements). That is, δ can modify *all* positions in p,

not only those for which a prediction is made. Further, note that the sequence of modifications $\boldsymbol{\delta} \subseteq \Delta(x)$ is computed for each x separately, rather than having the same set of modifications applied to all samples in $\mathbb{D}$.

Similar to the adversary defined in Section 3.4, using adversarial training for code requires a set of program modifications $\Delta(x)$, and a technique to solve the optimization problem $\max_{\boldsymbol{\delta} \subseteq \Delta(x)}$ efficiently. We elaborate on both of these next.

4.3.1 *Program Modifications*

To use adversarial training we define two classes of program modifications – label preserving and semantic preserving. Here, label preserving modifications are modifications that do not change the ground-truth label y. Semantic preserving modifications additionally ensure that also the overall program semantics do not change. Our use of label preserving modifications is motivated by the fact that preserving programs semantics is for many properties unnecessarily strict and restricts the variety of modifications that can be used. The modifications in both classes are further divided into three types – word substitutions, word renaming, and sequence substitutions.

- *Word substitutions* are allowed to substitute a word at a single position in the program with another word (not necessarily contained in the program). Examples of word substitutions include changing constants or values of binary/unary operators.

- *Word renaming* is a modification which includes renaming variables, parameters, fields or methods. In order to produce valid programs, this modification needs to ensure that the declaration and all usages are replaced jointly. Because of this, renaming a single variable in practice always corresponds to making multiple changes to the program (i.e., $|\delta| > 1$ unless the variable is used only once).

- *Sequence substitution* is the most general type of modification which can perform any label preserving program change such as adding dead code or reordering independent program statements.

The main property differentiating the modification types is that word renaming and substitution do not change the program structure. This is used both to compute which substitution should be made, as well as to provide formal correctness guarantees (discussed in Section 4.5). Further, is it used for efficiency, as we can implement word substitutions and word renaming directly on the batched tensors, thus making them fast. In contrast, sequence substitutions require parsing batched tensors back to programs, applying modifications on the programs and the processing the resulting programs back to batched tensors. In our experiments, our optimized implementation of word substitutions and renaming leads to $\approx 70\times$ runtime improvement, which is critical for fast training.

Additionally, it is also possible to define modifications that are not label preserving (i.e., change the ground-truth label), in which case the user has to additionally provide an oracle that computes the correct label y. We have provided an example of such oracle in Section 3.4 where we obtained ground-truth labels by executing the modified programs. For type inference, we could also execute programs to obtain concrete traces or we could run an existing static analyzer after each modification.

4.3.2 *Finding Adversarial Examples*

Given a program x, its associated ground-truth label y, and a set of valid modifications $\Delta(x)$ that can be applied on x, our goal is to select a subset $\boldsymbol{\delta} \subseteq \Delta(x)$ such that the inner term in the adversarial risk formula $\max_{\boldsymbol{\delta} \subseteq \Delta(x)} \ell(f(x + \boldsymbol{\delta}), y)$ is maximized. Solving for the optimal $\boldsymbol{\delta}$ is highly non-trivial since: *(i)* $\boldsymbol{\delta}$ is an ordered sequence rather than a single modification, *(ii)* the set of valid modifications $\Delta(x)$ is typically very large, and *(iii)* the modification can potentially perform arbitrary rewrites of the program (due to sequence substitutions). Thus, we focus on solving this maximization approximately, inline with how it is solved in other domains. In what follows, we discuss three approximate approaches to achieve this and discuss their advantages and limitations.

4.3.2.1 *Greedy Search*

The first approach is a greedy search that randomly samples a sequence of modifications $\boldsymbol{\delta} \subseteq \Delta(x)$. The sampling can be performed for a predefined number of steps with the goal of maximizing the adversarial risk, or until an adversarial example is found (i.e., $f(x + \boldsymbol{\delta}) \neq f(x)$). Concretely, for a given input $x = \langle p, l \rangle$, let us define the space of valid modifications $\Delta(x) \subseteq \Delta(p, l_1) \times \Delta(p, l_2) \times \cdots \times \Delta_n(p, l_n)$ as the Cartesian product of possible modification applied to each position in the program $l_{1:n}$. We select $\boldsymbol{\delta}$ using the following procedure: sample a threshold value $t \sim \mathcal{N}(0.1, 0.4)$ and apply the modification at each location with probability t. If $|\Delta_i(p, l_i)| > 1$, then the modification to apply is sampled at random from the set $\Delta_i(p, l_i)$. Sampling of the threshold value t is done per each sample x and ensures variety in the number of modifications applied.

LIMITATIONS AND ADVANTAGES The main advantage of this technique is that it is simple, easy to implement, and very fast. Given its simplicity, this technique is independent of the actual modification and applies equally to words substitutions, word renamings as well as sequence substitutions. However, a natural limitation of this technique is that is uses no information about which positions and which values are important for the prediction.

4.3.2.2 *Gradient-based Search*

Another approach to guide the search is to find which program positions to change. Recall that in Section 3.4.3, this was achieved by instrumenting the analyzer and recording all the visited AST nodes. Then, the visited nodes were simply sampled with higher probability. For neural networks, the same approach cannot be applied as the model *visits* all the nodes. However, an alternative solution for neural models exists – we can use gradient information to measure the importance of each position for a given prediction. Then, we can sample the positions to modify with probability proportional to their importance.

Formally, for a given sample $(x, y) \in \mathbb{D}$, we compute the importance of each node to the prediction made by the model f by computing:

$$\boldsymbol{a}(f, x, y) = \left[\|\mathbf{G}_{i,:}\|_1\right]_{i=1}^{|p|} \tag{4.10}$$

where $|p|$ is the program length[2] and $\mathbf{G} = \nabla_x \ \ell(f(x), y) \in \mathbb{R}^{|p| \times emb}$ denotes the gradient with respect to the input $x = \langle p, l \rangle$ and a given prediction y. As positions in p correspond to discrete words, the gradient is computed with respect to their embedding $emb \in \mathbb{R}$. The score for each position in p is computed by applying the L^1-norm over the embedding gradients, producing a vector of unnormalized scores $\boldsymbol{a} \in \mathbb{R}^{|p|}$. To obtain a probability distribution $\hat{\boldsymbol{a}}(f, x, y)$ over all positions in p, we normalize the entries in $\boldsymbol{a}$ accordingly.

Additionally, as shown in the concurrent work [106], the gradients can also be used to select both the program position and the new value to be used (instead of sampling from all valid values uniformly at random).

LIMITATIONS AND ADVANTAGES The main advantage of the gradient-based approach is that the decision of which position to change, as well as what the new value should be is guided, rather than random. Further, for renaming modifications, such approach was shown to be quite effective [106] at finding the adversarial examples. However, the main limitation of this approach is that it works only for replacing a single value (i.e., word substitutions and word renaming) and not when the value is a complex structure (i.e., sequence substitution). Sequence substitutions are an important class of modifications which are however hard to optimize for, as in general, they can perform arbitrary changes to the program (e.g., adding dead code, adding/removing statements, etc.).

4.3.2.3 *Reducing the Search Space*

The third technique is orthogonal to the first two and aims to reduce the search space of relevant modifications a priori, rather than searching it more efficiently. We achieve this by refining the program representation over which the model is learned, described in detail in the following section. As a concrete example, consider the code snippet x shown in Figure 4.6 *(a)*. Here we use the red dashed cirle

2 In our case, the program length corresponds to the number of nodes in the AST.

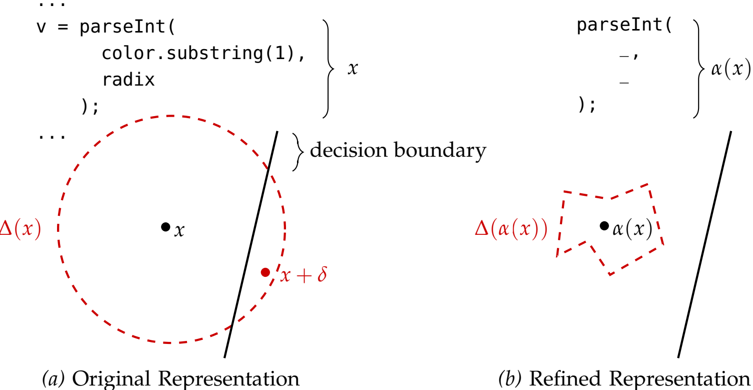


FIGURE 4.6: Illustration that shows the benefit of refined representation for learning robust models. It can be seen that the space of program modifications that needs to be considered when using refined representation $\Delta(\alpha(x))$ is significantly smaller compared to considering the program directly $\Delta(x)$.

to illustrate the set of valid program modifications $\Delta(x)$ and represent the code snippet as a point (in the latent space of the model) placed in the center of this region. The goal of a gradient based optimization would then be to iteratively find a modification $\delta \subseteq \Delta(x)$ which maximizes the loss and ideally, crosses the model's decision boundary. In our work, we refine the program representation by learning an abstraction function α that removes parts of the program unrelated to the prediction. In this case, when predicting the return type of the function `parseInt`, the refined representation might contain only the fact that it is a function `parseInt` with two arguments. However, the actual argument values (`color.substr(1)` and `radix`), as well as assignment (`v = ...`) are abstracted away. As a result, the refined space of valid modifications $\Delta(\alpha(x))$ is significantly smaller and easier to optimize. For example, the modification δ that crossed the decision boundary in Figure 4.6 *(a)* can no longer affect the model as it has been abstracted away.

LIMITATIONS AND ADVANTAGES The main advantage of this approach is that it applies to both renaming and structural modifications. The main disadvantage is that it depends on the fact that the dependencies between program locations can be checked efficiently and learned as part of the training. While we show how this can be done for graph neural networks, our approach currently does not support other models such as recurrent neural networks.

SUMMARY In this section, we described how adversarial attacks can be applied to code via a set of program modifications. The adversarial attacks we consider are applied on the discrete input (i.e., the attack always correspond to a concrete program) rather than considering attacks in the latent space that are not directly

interpretable. We describe two existing techniques that can be used to guide the search for adversarial attacks (greedy search and gradient-based search) and one that makes the attacks easier by reducing the search space. These techniques are quite general and can be applied to a number of tasks over code. We will discuss the concrete program modifications used in our evaluation in Section 4.6.

4.4 LEARNING TO REFINE REPRESENTATIONS

As motivated in Section 4.1, a key issue with many existing neural models for code is that the model prediction $f(x)$ depends on the *full* program p, even though only small parts of p are typically relevant. We address this issue by learning an abstraction α that takes as input p and produces only the parts relevant for the prediction. That is, α refines the representation given as input to the neural model.

OVERVIEW Our method works as follows:

1. we convert the program into a graph representation,
2. then we define the model to be a graph neural network (e.g., [92, 186–188]), which at a high level works by propagating and aggregating messages along graph edges,
3. because dependencies in graph neural networks are defined by the structure of the graph (i.e., the edges it contains), we phrase the problem of refining the representation as an optimization problem which removes the maximum number of graph edges (i.e., removes the maximum number of dependencies) without degrading the model's accuracy, and
4. we show how to solve the optimization problem efficiently by transforming it to an integer linear program (ILP).

FROM PROGRAMS TO GRAPHS Following the approach from Chapters 2 and 3, we start by representing programs using their corresponding abstract syntax trees (AST). These are further transformed into graphs, as done in [26, 43], by including additional edges.

Definition 4.4.1. (Directed Graph) A directed graph is a tuple $G = \langle V, E, \xi_V, \xi_E \rangle$ where V denotes a set of nodes, $E \subseteq V^2$ denotes a set of directed edges, $\xi_V : V \to \mathbb{N}^k$ is a mapping from nodes to their associated attributes and $\xi_E : E \to \mathbb{N}^m$ is a mapping from edges to their attributes.

We associate two attributes with each node – *type* which corresponds to the type of the AST node (e.g., `Block`, `Identifier`, `BinaryExpression`, etc.) and *value* associated with the AST node (e.g., $+$, $-$, `0`, `1`, "`GET`", `x`, `data`, etc.). For edges we use a single attribute the edge *type*, which can be: *(i) ast*, for the edges that correspond to those included in the AST, *(ii) last usage*, for edges introduced between any two

usages (either read or write) of the same variable, and *(iii) returns-to*, for edges introduced between a return statement and the function declaration. All edges are initially added in both directions, but can be later removed during training. Depending on the task, more edge types can be easily added.

REPRESENTATION REFINEMENT Our goal is to learn an abstraction function $\alpha\colon \langle V, E, \xi_V, \xi_E \rangle \to \langle V, E' \subseteq E, \xi_V, \xi_E \rangle$ that removes a subset of the edges from the graph. To quantify the size of the abstraction, we use $|\alpha(x)| := |E'|$ to denote the number of edges after applying α on x.

DEFINING VALID GRAPH REFINEMENTS Because the goal of representation refinement is to reduce the number of nodes on which a prediction depends, we need to ensure that α itself does not depend on all the graph nodes. This is necessary as otherwise we only shift the dependency on the *entire* program from the model f to the representation refinement α. To achieve this, the decision to include or remove a given edge is done *locally*, based only on the edge attributes and attributes of the nodes it connects.

Concretely, for a given edge $\langle s, t \rangle \in E$, we define an edge feature $\phi(\langle s, t \rangle) := \langle \xi_E(\langle s, t \rangle), \xi_V(s), \xi_V(t) \rangle$ to be a tuple of the edge attributes and attributes of the nodes it connects. As a form of regularization, we condition only on the *type* attribute of each node. We denote the set of all possible edge features Φ to be the range of the function ϕ evaluated over all edges in $\mathbb{D}$. Further, we define the refinement α as a subset of edge features $\alpha \subseteq \Phi$. Finally, the semantics of executing α over edges E is that only edges whose features are in α are kept, i.e., $\{e \mid e \in E \land \phi(e) \in \alpha\}$.

PROBLEM STATEMENT We formulate the refinement learning problem as follows – minimize the expected size of the refinement $\alpha \subseteq \Phi$ subject to the constraint that the expected loss of the model f stays approximately the same:

$$\arg\min_{\alpha \subseteq \Phi} \sum_{(x,y) \in \mathbb{D}} |\alpha(x)| \tag{4.11}$$

subject to

$$\textstyle\sum_{(x,y) \in \mathbb{D}} \ell(f(x), y) \approx \sum_{(x,y) \in \mathbb{D}} \ell(f(\alpha(x)), y)$$

Our problem statement is quite general and can be instantiated by using both the loss $\ell_{\texttt{AbstainCrossEntropy}}$ (Section 4.2) and using *adversarial risk* (Section 4.3).

Allowing the model to abstain from making predictions is important in order to obtain small $|\alpha|$ (i.e., sparse graphs). This is because the restriction that the model accuracy is roughly the same is otherwise too strict and would require that most edges are kept. Further, note that the problem formulation is defined over *all* samples in $\mathbb{D}$, not only those for which the model f predicts the correct label. This is necessary since the model needs to make a prediction for all samples, even if that prediction is to abstain.

OPTIMIZATION VIA INTEGER LINEAR PROGRAMMING (ILP) To solve Equation 4.11 efficiently, the key idea is that for each sample $(x, y) \in \mathbb{D}$ we: *(i)* capture the relevance of each node to the prediction made by the model f by computing the *attribution* $\boldsymbol{a}(f, x, y) \in \mathbb{R}^{|V|}$ (using Equation 4.10), and *(ii)* include the minimum number of edges necessary for a path to exist between every relevant node (according to the attribution $\boldsymbol{a}$) and the node where the prediction is made. Preserving all paths between the prediction and relevant nodes encodes the constraint that the expected loss stays approximately the same, since it allows propagating information throughout the graph neural network. This optimization can be naturally encoded as minimum-cost maximum-flow problem and solved efficiently with off-the-shelf integer linear programming solvers.

Concretely, let us define a *sink* to be the node for which the prediction is being made while *sources* are defined to be all nodes v with *attribution* $a_v > t$. Here, the threshold $t \in \mathbb{R}$ is used as a form of regularization. To encode the *sources* and the *sink* as an ILP program, we define an integer variable r_v associated with each node $v \in V$ as:

$$r_v = \begin{cases} -\sum_{v' \in V \setminus \{v\}} r_{v'} & \text{if } v \text{ is predicted node [\textit{sink}]} \\ \lfloor 100 \cdot a_v \rfloor & \text{else if } a_v > t \qquad \text{[\textit{sources}]} \\ 0 & \text{otherwise} \end{cases}$$

That is, r_v for a source is its attribution value converted to an integer and r_v for a sink is a negative sum of all source values. Note that in our definition it is not possible for a single node to be both *source* and a *sink*. For cases when the *sink* node has a non-zero attribution, this attribution is simply left out since every node is trivially connected to itself.

We then define our ILP formulation of the problem as shown below:

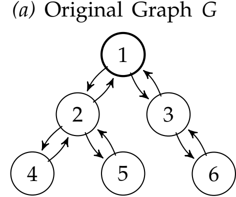
$$\min_{\forall(\langle V, E, \xi_V, \xi_E \rangle, y) \in \mathbb{D}} \quad \sum_{q \in \Phi} cost_q \tag{4.12}$$

subject to

$$\begin{aligned} 0 \le f_{st} \le cost_{\phi(\langle s,t \rangle)} & \qquad \forall \langle s, t \rangle \in E \quad \text{[edge capacity]} \\ r_v + \sum_{\{s | (s,v) \in E\}} f_{sv} = \sum_{\{t | (v,t) \in E\}} f_{vt} & \qquad \forall v \in V \quad \text{[flow conservation]} \end{aligned}$$

Here $cost_q$ is an integer variable associated with each edge feature and denotes the edge capacity (i.e., the maximum amount of flow allowed to go trough the edge with this feature), f_{st} is an integer variable denoting the amount of flow over the edge $\langle s, t \rangle$, the constraint $0 \le f_{st} \le cost_{\phi(\langle s,t \rangle)}$ encodes the edge capacity, and $r_v + \sum_{\{s|(s,v) \in E\}} f_{sv} = \sum_{\{t|(v,t) \in E\}} f_{vt}$ encodes the flow conservation constraint which requires that the flow generated by the node r_v together with the flow from all the incoming edges $\sum_{\{s|(s,v) \in E\}} f_{sv}$ has to be the same as the flow leaving the node $\sum_{\{t|(v,t) \in E\}} f_{vt}$. The solution to this ILP program is a *cost* associated with each edge feature $q \in \Phi$. If the cost for a given edge feature is zero, it means that this

(a) Original Graph G



(e) Problem Statement

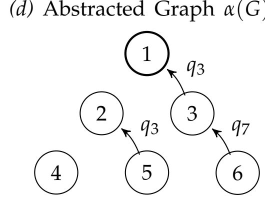
OPTIMIZATION PROBLEM

minimize $\sum_{i=1}^{7} cost_{q_i}$ subj. to

edge capacity constraints
flow conservation constraints

SOLUTION $\alpha = \{q_3, q_7\}$

(d) Abstracted Graph $\alpha(G)$



(b) Graph Nodes V

V	ξ_V	$\boldsymbol{a}$	r_v	flow conservation constraints
1	A	0.3	$r_1 = -70$	$r_1 + f_{21} + f_{31} = f_{12} + f_{13}$
2	A	0	$r_2 = 0$	$r_2 + f_{12} + f_{42} + f_{52} = f_{21} + f_{24} + f_{25}$
3	B	0	$r_3 = 0$	$r_3 + f_{13} + f_{63} = f_{31} + f_{36}$
4	C	0	$r_4 = 0$	$r_4 + f_{24} = f_{42}$
5	B	0	$r_5 = 0$	$r_5 + f_{25} = f_{52}$
6	D	0.7	$r_6 = 70$	$r_6 + f_{36} = f_{63}$

(c) Graph Edges E

E	ξ_E	$\phi(\langle s,t \rangle)$	edge capacity constraints	optimal cost
$\langle 1,2 \rangle$	*ast*	$q_1 = \langle ast, A, A \rangle$	$0 \leq f_{12} \leq cost_{q1}$	$cost_{q1} = 0$
$\langle 1,3 \rangle$	*ast*	$q_2 = \langle ast, A, B \rangle$	$0 \leq f_{13} \leq cost_{q2}$	$cost_{q2} = 0$
$\langle 3,1 \rangle$	*ast*	$q_3 = \langle ast, B, A \rangle$	$0 \leq f_{31} \leq cost_{q3}$	$cost_{q3} = 70$
$\langle 5,2 \rangle$	*ast*	$q_3 = \langle ast, B, A \rangle$	$0 \leq f_{52} \leq cost_{q3}$	
		...		
$\langle 6,3 \rangle$	*ast*	$q_7 = \langle ast, D, B \rangle$	$0 \leq f_{63} \leq cost_{q7}$	$cost_{q7} = 70$

FIGURE 4.7: An example of the ILP encoding from Equation 4.12 on a single graph *(a)* where the prediction should be made for node 1. The refined graph is shown in *(d)* and contains only three edges necessary to connect node 1 and 6.

feature was not relevant and can be removed. As a result, we define the refinement $\alpha = \{q \mid q \in \Phi \wedge cost_q > 0\}$ to contain all edge features with non-zero weight.

Example As a concrete example, consider the initial graph shown in Figure 4.7 *(a)* and assume that the prediction is made for node 1. For simplicity, each node has a single attribute ξ_V, as shown in Figure 4.7 *(b)*, and all edges are of type *ast*. The edge feature for edge $\langle 1,3 \rangle$ is therefore $\langle ast, A, B \rangle$, since $\xi_E(\langle 1,3 \rangle) = ast$, $\xi_V(1) = A$ and $\xi_V(3) = B$, as shown in Figure 4.7 *(c)*. The *attribution* $\boldsymbol{a}$ reveals two relevant nodes for this prediction – the node itself with score 0.3 and node 6 with score 0.7. Therefore, we define a single source $r_6 = 70$ and a sink $r_1 = -70$ and encode both the edge capacity constraints, and the flow conservation constraints as shown in Figure 4.7 (note that according to Equation 4.12, we would encode all samples in $\mathbb{D}$ jointly). The minimal cost solution assigns cost 70 to edge features q_3 and q_7

which are needed to propagate the flow from node 6 to node 1. The graph obtained by applying the abstraction $\alpha = \{q_3, q_7\}$ is shown in Figure 4.7 *(d)* and makes the prediction independent of the subtree rooted at node 2. Note however, that an additional edge is included between nodes 5 and 2. This is because α is computed using *local* edge features ϕ only, which are the same for edges $\langle 3, 1\rangle$ and $\langle 5, 2\rangle$.

SUMMARY We presented a novel approach to refine the program representation used as input to the neural model. To achieve this, we represent programs as graphs and learn to remove edges not relevant for the prediction (i.e., to sparsify the graph). We encode this problem as an integer linear program which can be solved very efficiently with off-the-shelf solvers – in all our experiments the solver takes less than a second to complete. We note, however, that an end-to-end trainable solution is also possible. For example, one could make α continuous by defining a learnable weight for each edge feature ϕ, encode the sparsity on α as part of the loss, and extend the graph neural network such that each message propagated along an edge e is scaled using the corresponding value of the edge feature $\phi(e)$.

4.5 TRAINING ALGORITHM

We now describe our algorithm that combines learning to abstain, adversarial training and representation refinement.

TRAINING A SINGLE ADVERSARIALLY ROBUST MODEL The training procedure used to learn a single adversarially robust model is shown in Algorithm 3. The input is a training dataset $\mathbb{D}$ and the desired accuracy t_{acc} that the learned model should have. Here, setting $t_{\mathsf{acc}} = 1.0$ corresponds to a model that makes no mis-prediction (i.e., 100% accuracy), while $t_{\mathsf{acc}} = 0$ corresponds to training a model that never abstains.

We start by training a model f and a selection function g_h as described in Section 4.2 (line 3). At this point we do not use adversarial training and train with a weaker threshold $t_{\mathsf{acc}} - \epsilon$, as our goal is only to obtain a fast approximation of the samples that can be predicted with high certainty. We use f and g_h to obtain an initial representation refinement α (line 5) which is applied to the dataset $\mathbb{D}$ to remove edges that are not relevant according to f and g_h (line 9). After that, we perform adversarial training (line 10) as described in Section 4.3. However, instead of training from scratch, we reuse the model f and g_h learned so far, which speeds-up training. Next, we refine the representation again (line 5) and if the new representation is smaller (line 6), we repeat the whole process. Note that the adversarial training also uses threshold $t_{\mathsf{acc}} - \epsilon$ to account for the fact that the suitable representation is not known in advance. After the training loop finishes, we set the threshold h used by g_h to match the desired accuracy t_{acc}. The final result is a model consisting of the function f trained to make adversarially robust predictions, the selection function g_h and the abstraction α.

Algorithm 3: Training procedure used to learn a single adversarially robust model $\langle f, g_h, \alpha \rangle$.

```
def RobustTrain(D, t_acc)
    Input: Dataset D = {(x_j, y_j)}_{j=1}^n, threshold t_acc
    Output: Model f, selection function g_h, abstraction α ⊆ Φ
1   α_last ← Φ                      ▷ Start with no refinement, i.e., all edge features
2   D_refined ← D
3   f, g_h ← Train(D, t_acc − ε)
4   while true do
5       α ← RefineRepresentation(D_refined, f, g_h)
6       if |α| ≥ |α_last| then
7           break
8       α_last ← α
9       D_refined ← {(α(x), y) | (x, y) ∈ D}          ▷ Apply α on the dataset
10      f, g_h ← AdversarialTrain(D_refined, f, g_h, t_acc − ε)          ▷ Retrain
11  set threshold h in g_h such that the accuracy is t_acc
12  return ⟨f, g_h, α⟩
```

INCORPORATING ROBUST PREDICTIONS Once a single model is learned, it makes robust predictions on a subset of the dataset $\mathbb{D}_{\texttt{predict}} = \{(x, y) \mid (x, y) \in \mathbb{D} \wedge g_h(\alpha(x)) \geq h\}$ and abstains from making a prediction on the remainder of the samples $\mathbb{D}_{\texttt{abstain}} = \mathbb{D} \setminus \mathbb{D}_{\texttt{predict}}$. Next, for all samples in $\mathbb{D}_{\texttt{predict}}$, we use the learned model to annotate the position l in the program p (recall that each $x = \langle p, l \rangle$ consists of a program p and a position l) with the ground-truth label y (denoted as `Apply` in Algorithm 4). Annotating a program position corresponds to either defining a new attribute (as illustrated in Figure 4.5) or replacing an existing attribute (e.g., the *value* attribute) of a given node. Note that annotating programs is useful only in cases where the same program p is shared by multiple samples $(x, y) \in \mathbb{D}$ (i.e., multiple predictions are computed for different positions in the same program).

MAIN TRAINING ALGORITHM Our main training algorithm is shown in Algorithm 4. It takes as input the training dataset $\mathbb{D}$ and learns multiple models M, each of which makes robust predictions on a different subset of $\mathbb{D}$ (as motivated in Section 4.1). The number of models and the subsets for which they make predictions is not fixed a priori and is learned as part of our training. Model training (line 3) and model application (line 4) are performed as long as a non-empty robust model exists (i.e., it makes at least one prediction). If the goal is to make predictions for all the samples in $\mathbb{D}$, the Algorithm 4 is run iteratively, with decreasing values of $t_{\texttt{acc}}$ until the full dataset is covered.

Algorithm 4: Training multiple adversarially robust models, each of which learns to make predictions for a different subset of the dataset $\mathbb{D}$.

```
def AccurateAndRobustTrain(𝔻, t_acc = 1.0)
    Input: Dataset 𝔻 = {(x_j, y_j)}_{j=1}^n, threshold t_acc
    Output: Sequence of models M
1   M ← []
2   while true do
3       ⟨f, g_h, α⟩ ← RobustTrain(𝔻, t_acc)          ▷ From Algorithm 3
4       𝔻_abstain ← Apply(𝔻, f, g_h, α)
5       if |𝔻_abstain| = |𝔻| then
6           break
7       𝔻 ← 𝔻_abstain
8       M ← M · ⟨f, g_h, α⟩
9   return M
```

VERIFYING MODEL CORRECTNESS A natural extension of our approach is to formally verify that the learned models are correct. Even though formally verifying the correctness of all samples is typically infeasible, it is possible to verify a subset of them. This can be achieved since using representation refinement significantly simplifies the problem of proving correctness of *all* positions (nodes) in the program to a much smaller set of relevant positions. In fact, for some cases the refined representation is so small that it is possible to simply enumerate all valid modifications (e.g., a finite set of valid variable renamings) and check that the model is correct for all of them. Additionally, it would be possible to adapt the recently proposed techniques [189, 190] based on Interval Bound Propagation, which verify robustness with respect to any valid word renaming and word substitution modifications. However, applying these techniques to realistic networks in a scalable and precise ways is an open problem beyond the scope of our work.

4.6 EVALUATION

We instantiated our approach to a well studied task – predicting types for dynamically typed languages `JavaScript` and `TypeScript` [9, 30, 31, 179]. In this task, the need for model robustness is natural since the model is queried each time a program is modified by the user. Our key results are:

- *Our approach learns accurate and adversarially robust models* for the task of type inference, achieving 87.7% accuracy while improving robustness from 52.1% to 67.0%.
- We train highly accurate and robust models *for a subset of the dataset*, with 99.9% accuracy and 99.9% robustness for 29% of the samples.

All our models are implemented in PyTorch [191]. The graph neural networks are implemented using the DGL library v0.4.3 [192] and to solve the integer linear program we use Gurobi solver v8.11 [193]. All our experiments used a single Nvidia TITAN RTX. We make all the models, datasets and code available at:

`https://github.com/eth-sri/robust-code`

DATASET To obtain the datasets used in our work, we extend the infrastructure from `DeepTyper` [30], collect the same top starred projects on GitHub, and perform similar preprocessing steps – remove `TypeScript` header files, remove files with less than 100 or more than 3 000 tokens and split the projects into train, validation and test datasets such that each project is fully contained in one of the datasets. Additionally, we remove exact file duplicates and files that are similar to each other ($\approx 10\%$ of the files). We measure file similarity by collecting all 3-grams (excluding comments and whitespace) and removing files with Jaccard similarity greater than 0.7.

We compute the ground-truth types using the `TypeScript` compiler version 3.4.5 based on manual type annotations, library specifications and analyzing all project files. While we reuse the same GitHub projects and part of `DeepTyper`'s infrastructure[3] to obtain the dataset, the datasets are not directly comparable for a number of reasons. First, we fixed a bug due to which some type annotations were incorrectly included as part of the input. Second, the projects we used are a subset of those used in `DeepTyper` since some are no longer available and were removed from GitHub. Third, we additionally predict the types corresponding to all intermediate expressions and constants (e.g., the expression $x + y$ contains three predictions for x, y and $x + y$). This improves model performance as it is explicitly trained also on the intermediate steps required to infer the types. Finally, we train all the models to predict four primitive types (`string`, `number`, `boolean`, `void`), four function types (`() ⇒ string`, `() ⇒ number`, `() ⇒ boolean`, `() ⇒ void`) and a special `unk` label denoting all the other types. While this is similar to types predicted by some other works (e.g., [22]), it is only a subset of the types considered in `DeepTyper`.

DATASET SIZE We trained our models using a dataset that contains 3 000 programs split equally between training, validation and test datasets. Because each program contains multiple type predictions, the number of training samples is significantly higher than the number of programs. Concretely, there are 139 915, 223 912 and 121 153 samples in training, validation and test datasets. We note that this is only a subset of the full dataset that can be obtained by processing all the files included in the projects used by `DeepTyper`.

During adversarial training, we explore 20 different modifications $\boldsymbol{\delta} \subseteq \Delta(x)$ applied to each sample $(x, y) \in \mathbb{D}$ which effectively increases the dataset size by up to two orders of magnitude, since for each training epoch the modifications are different. For the purposes of evaluation, we increase the number of explored

3 `https://github.com/DeepTyper/DeepTyper`

modifications to 1 300 for each sample – 1 000 for renaming modifications and a further 300 for renaming together with structural modifications. These thresholds are rather high and were selected with the goal of closely estimating the true number of adversarial samples. Further, note that since $\boldsymbol{\delta} \subseteq \Delta(x)$ is a set, each iteration explores a set of concrete program modifications.

evaluation metrics We use two main evaluation metrics:

- *Accuracy* is computed over the unmodified dataset $\mathbb{D}$ and corresponds to the accuracy used in prior works. Concretely, the accuracy is defined as the ratio of samples (x, y) for which the most likely label according to the model f, denoted $f(x)_{\texttt{best}}$, is the same as the ground truth label y:

$$\frac{1}{|\mathbb{D}|} \sum_{(x,y)\in\mathbb{D}} \begin{cases} 1 & \text{if } f(x)_{\texttt{best}} = y \\ 0 & \text{otherwise} \end{cases}$$

- *Robustness* is the ratio of samples $(x, y) \in \mathbb{D}$ for which the model f evaluated on all valid modifications $\boldsymbol{\delta} \subseteq \Delta(x)$ either abstains or makes a correct prediction:

$$\frac{1}{|\mathbb{D}|} \sum_{(x,y)\in\mathbb{D}} \begin{cases} 0 & \text{if } \exists_{\boldsymbol{\delta}\subseteq\Delta(x)} f(x+\boldsymbol{\delta})_{\texttt{best}} \notin \{y, \texttt{abstain}\} \\ 1 & \text{otherwise} \end{cases}$$

models We evaluate five neural model architectures:

- `LSTM` is a bidirectional LSTM with attention which takes as input a sequence of AST nodes, including both types and values, obtained using pre-order traversal.

- `DeepTyper` is a model proposed by Hellendoorn *et al.* [30] and consists of a bidirectional LSTM layer, followed by a single layer graph neural network that connects all variables with the same name (referred as consistency layer), followed by another bidirectional LSTM layer. Our only modification is that the input to our model is a sequence of AST types and values, instead of using syntactic program tokens.

- `GCN`, `GGNN` and `GNT` are three graph neural networks that use as input the graph program representation described in Section 4.4. Here, `GCN` is a Graph Convolutional Network [186], `GGNN` is Gated Graph Neural Network [188] and `GNT` is a graph implementation of a recently proposed transformer neural network architecture [116, 194].

All models were trained with an embedding and hidden size of 128, batch size of 32, dropout 0.1 [195], initial learning rate of 0.001, using Adam optimizer [196] and between 10 to 20 epochs.

Substitutions and Renaming	Examples
Semantic Preserving	
variable renaming	`x` → `y`
object field renaming	`obj.x` → `obj.y`
property assignment renaming	{`x` : `obj`} → {`y` : `obj`}
Label Preserving	
number substitution	2 → 7
string substitution	"`get`" → "`load`"
boolean substitution	`true` → `false`

TABLE 4.1: List of renaming and substitution program modifications used in our work.

Structural Modifications	Examples
Label Preserving	
new function parameters	`def inc(x)` → `def inc(x, y)`
new method arguments	`inc(x)` → `inc(x, expr)`
Semantic Preserving	
ternary expressions	$\texttt{expr}_1 \rightarrow (\texttt{expr}_2)\ ?\ \texttt{expr}_1 : \texttt{expr}_1$
array access	`expr` → `[expr, expr][const]`
side-effect free expressions	∅ → `expr`
adding object expressions	∅ → {`x` : `y`, `z` : `expr`}

TABLE 4.2: List of structural program modifications used in our work.

PROGRAM MODIFICATIONS We instantiate the adversarial training with both semantic preserving and label-preserving modifications shown in Table 4.1 and Table 4.2. More importantly, we use a range of modifications that include word substitutions, word renamings, as well as structural modifications, such as assing side-effect free expressions or wrapping existing expressions in ternary expressions. Here, `expr` is either an existing expression or a new expression consisting of a random binary expression over constants up to depth 3, `const` is a randomly selected constant that results in a valid expression and `x`, `y`, `z` are variables in the program scope. The concrete modification we use are similar to those in Section 3.4.2, but we do note that the list is still not exhaustive and can be extended in the future.

REDUCING DEPENDENCIES VIA DYNAMIC HALTING We further strengthen our `GNT` model by implementing the adaptive computation time (ACT) [197] which dynamically learns how many computational steps each node requires in order to make a prediction. This is in contrast to using a fixed amount of steps as done

in [26, 43]. In our experiments, ACT significantly reduces the number of steps each node performs (half of the nodes perform $\leq$3 steps).

To implement ACT, recall that for each node $v_i \in V$ in the graph, a graph neural network computes a sequence of hidden states $\boldsymbol{s}_t^i$ where $t \in \mathbb{N}$ is the timestep[4]. Following [197], the number of timesteps that the model performs is controlled by introducing an extra sigmoidal halting unit $h_t^i \in \mathbb{R}^{(0,1)}$ with associated learnable weight matrix W_h and bias b_h:

$$h_t^i = \sigma(W_h s_t^i + b_h)$$

The output of the halting unit is then used to determine the halting probability p_t^i as follows:

$$p_t^i = \begin{cases} 1 - \sum_{k=0}^{t-1} h_k^i & \text{if } t = T \text{ (last timestep)} \\ 1 - \sum_{k=0}^{t-1} h_k^i & \text{if } \sum_{k=0}^{t} h_k^i \geq 1 - \epsilon \\ h_t^i & \text{otherwise} \end{cases}$$

where $T \in \mathbb{N}$ is the maximum allowed number of timesteps and $\epsilon \in \mathbb{R}^{(0,1)}$ is a small constant introduced to allow the network to stop after a single step (we use $\epsilon = 0.01$ in our experiments). Finally, the halting probability p_t^i is used to define the final state $\boldsymbol{s}_T^i$ of a node v_i as a weighted average of its intermediate states:

$$\boldsymbol{s}_T^i = \sum_{t=0}^{T} p_t^i \cdot \boldsymbol{s}_t^i$$

4.6.1 *Accurate and Adversarially Robust Models*

We summarize the main results in Table 4.3. The second column (left) shows the median test *accuracy* and standard deviation of various models (across three trials trained with different random seeds). The `GCN` achieves the worst accuracy of 82.6% as it corresponds to a very simple graph neural networks architecture. The `GGNN` architecture improves the accuracy to 86.7%, which is further improved by our `GNT` graph transformer model to 89.3%. In comparison, the accuracy of the traditional `LSTM` model is slightly worse but still very hight at 88.2%. Similarly, the `DeepTyper` achieves slightly worse but very similar accuracy of 88.4%. Note that for both `LSTM` and `DeepTyper` models, the accuracy is reported when trained using the AST program representation as the input. We have also compared to using syntactic program tokens as proposed originally in `DeepTyper` and found that using this simpler representation leads to ≈1% accuracy decrease for both `LSTM` and `DeepTyper`. As a result, all the results for these models are reported using the stronger AST based representation.

4 Note that we assume only that the model computes $\boldsymbol{s}_t^i$ for each timestep, not how $\boldsymbol{s}_t^i$ is computed. As a result, the approach is independent of the concrete graph neural architecture used to compute $\boldsymbol{s}_t^i$.

	$\ell(f(x), y)$ **Standard Training**		$\max_{\boldsymbol{\delta} \subseteq \Delta(x)} \ell(f(x+\boldsymbol{\delta}), y)$ **Adversarial Training**	
Model	Accuracy	Robustness	Accuracy	Robustness
`LSTM`	88.2% $\pm$ 0.2	44.9% $\pm$ 1.3	87.5% $\pm$ 0.4	51.9% $\pm$ 1.3
`DeepTyper`	88.4% $\pm$ 0.2	52.4% $\pm$ 1.2	87.1% $\pm$ 0.3	55.1% $\pm$ 2.6
`GCN`	82.6% $\pm$ 0.6	49.1% $\pm$ 1.1	81.9% $\pm$ 0.5	49.3% $\pm$ 3.1
`GNT`	89.3% $\pm$ 0.9	47.4% $\pm$ 1.0	88.3% $\pm$ 0.4	50.0% $\pm$ 0.5
`GGNN`	86.7% $\pm$ 0.4	52.1% $\pm$ 0.4	86.1% $\pm$ 0.2	57.9% $\pm$ 1.5

TABLE 4.3: Comparison of accuracy and robustness across various models when using standard training compared to adversarial training.

EXISTING MODELS ARE NOT ROBUST While highly accurate, all models are also non-robust for up to half of the samples in the dataset. In other words, for every second sample x in our dataset, there exists a modification $\boldsymbol{\delta} \subseteq \Delta(x)$ for which $f(x)$ predicts the type correctly while $f(x + \boldsymbol{\delta})$ mis-predicts it. However, since these models were not trained with the goal of adversarial robustness, it is expected for them to be (at least partially) non-robust.

ADVERSARIAL TRAINING ALONE IS INSUFFICIENT To improve the robustness, we next train the models using adversarial training as described in Section 4.3 and present the results in Table 4.3 (last two columns). Unfortunately, while the adversarial training increase the robustness, it does so only slightly. The best improvement was achieved for `LSTM` and `GGNN` models (7% and 5.8%, respectively). For `DeepTyper` and `GNT` the robustness increased by $\approx$ 2.5% while for `GCN` it is only 0.2%. This illustrates that while useful, if used alone, *adversarial training* is not enough.

OUR WORK: TRAINING ACCURATE MODELS THAT ABSTAIN In Table 4.4 we show the main results achieved by the models trained in our work. First, we trained our models to be both *accurate* and *robust* on a subset of the dataset. This can be achieved by setting the desired accuracy threshold, in our case $t_{acc} = 1.00$, which corresponds to training the model to make only correct predictions. For $t_{acc} = 1.00$, our approach learns an almost perfect model that is both accurate and robust for $\approx$30.0% of the samples. Here, `GNT` learned 7 models and achieved 99.98% robustness while `GGNN` learned 8 models with robustness of 99.01%. Learning multiple models is crucial for achieving higher coverage, as a single model would not abstain for only $17 - 20$% of the samples, compared to 30% using multiple models.

The model did not achieve 100% accuracy and robustness for $t_{acc} = 1.00$ due to several samples included in the test set. These samples were mis-predicted because they contained code structures not seen during training and not covered by the modifications $\boldsymbol{\delta} \subseteq \Delta(x)$. This illustrates that it is important that the samples

$\max_{\delta \subseteq \Delta(x)} \ell_{\texttt{AbstainCE}}((f, g_h)(\alpha(x+\delta)), y)$ **Abstain + Adversarial + Refinement**		
Model	Accuracy	Robustness
$t_{\text{acc}} = 1.00$ (Abstain $\approx 70\%$)		
GNT	99.93%	99.98%
GGNN	99.80%	99.01%

$\max_{\delta \subseteq \Delta(x)} \ell_{\texttt{AbstainCE}}((f, g_h)(\alpha(x+\delta)), y)$ **Abstain + Adversarial + Refinement**		
Model	Accuracy	Robustness
$t_{\text{acc}} = 0.00$		
GNT	86.6%	62.3%
GGNN	87.7%	67.0%

TABLE 4.4: Comparison of accuracy and robustness when using our approach with all three components – learning to abstain, adversarial trining and representation refinement. Note that the adversarial training and the ability to abstain is applicable to all the models. The representation refinement is designed specifically to models defined over graphs, including GCN, GGNN and GNT.

in $\mathbb{D}$ are diverse and contain all the language features and corner cases of the programs, or that the modifications $\Delta(x)$ are expressive enough such that these can be discovered automatically during training.

OUR WORK: IMPROVING ROBUSTNESS Next, we train models that take advantage of the highly accurate and robust models trained using $t_{acc} = 1.00$, but make predictions for all the samples (i.e., do not abstain). This can be achieved by continuing the training while reducing t_{acc} to zero and conditioning on all the models trained with higher t_{acc}. In our experiments, we train a single additional model by directly setting $t_{acc} = 0$ after training with $t_{acc} = 1.00$. The results are shown in Table 4.4 *(right)* and lead to additional robustness increase of 9.2% and 12.3% compared to using adversarial training only for GGNN and GNT, respectively. For GNT, the accuracy slightly decreases by 1.7% which is expected as increasing model robustness typically comes at the cost of reduced accuracy [198]. Interestingly, for GGNN our robust training increases the accuracy over both the adversarial training, as well as standard training by 1.9% and 1.0%, respectively.

ADVERSARIAL ROBUSTNESS BREAKDOWN Table 4.5 provides a detailed breakdown of the *robustness* metric for the GNT and GGNN models trained with $t_{acc} = 1.00$ from Table 4.4. Here, $\mathbb{D}_{\texttt{abstain}}$ contains the samples for which the model abstains from making a prediction and $\mathbb{D}_{\texttt{correct}}$ contains the samples for which the model evaluated on a non-adversarial input (i.e., x without any modification) makes a correct prediction. We use $\forall$ Correct to denote that a sample (x, y) is correct for *all* possible modifications $\boldsymbol{\delta} \subseteq \Delta(x)$, the $\exists$ incorrect has the same definition as robustness (i.e., there exists a modification that leads to an incorrect prediction), and abstain denotes the remaining samples.

The GNT is precise and keeps predicting the correct label in 90% of cases and abstain in the rest. This is even though the requirements for $\forall$ Correct are very strict and require that all samples are correct. When considering $\mathbb{D}_{\texttt{abstain}}$, the GNT model

Model	Dataset	Size	Robustness ∀ Correct	 ∃ Incorrect	 Abstain
GNT with $t_{\texttt{acc}} = 1.00$	$\mathbb{D}_{\texttt{correct}}$	29.3%	90.0%	0.00%	10.00%
	$\mathbb{D}_{\texttt{abstain}}$	70.6%	–	0.01%	99.99%
GGNN with $t_{\texttt{acc}} = 1.00$	$\mathbb{D}_{\texttt{correct}}$	30.6%	75.5%	0.06%	23.94%
	$\mathbb{D}_{\texttt{abstain}}$	69.3%	–	1.46%	98.54%

∀ Correct $:= \forall_{\delta \subseteq \Delta(x)} (f, g_h)(\alpha(x + \delta))_{\texttt{best}} = y$

∃ Incorrect $:= \exists_{\delta \subseteq \Delta(x)} (f, g_h)(\alpha(x + \delta))_{\texttt{best}} \notin \{y, \texttt{abstain}\}$

Table 4.5: Robustness breakdown for the GNT and GGNN models trained using our approach from Table 4.4.

is also precise and produces incorrect predictions for only one sample (0.01%). For GGNN the results are similar but the model is both less precise (keeps the correct prediction in 75.5% of the cases) and less robust (1.46% of the samples in $\mathbb{D}_{\texttt{abstain}}$ can be modified to cause a mis-prediction). This shows that the majority of robustness errors from Table 4.3 are due to mis-predicted samples for which the model originally abstained.

4.7 RELATED WORK

Our work is related to a number of different lines of work from adversarial machine learning and learning over code.

MODEL CERTAINTY Several approaches have been recently proposed to extend neural models with certainty measures [180, 182–185]. In our work, we use the method proposed by Liu *et al.* [180] but in a novel way – applied to the adversarial setting with the goal of training robust models.

ADVERSARIAL TRAINING Even though the problem of adversarial robustness of code has been overlooked, the adversarial training has already been applied to related domains – natural language processing [55, 101–105] and graphs [199–201].

In the domain of *graphs*, existing works focus on attacking the graph structure [199–201] by considering that the nodes are fixed and edges can be added or removed. While this setting is natural for modelling many types of graphs, such approaches do not apply for the domain of code where graph edges cannot be added and removed arbitrarily.

In *natural language processing*, existing approaches generally: *(i)* measure the contribution of individual words or characters to the prediction (e.g., using gradients [105], forward derivatives [55] or head/tail scores [104]), and *(ii)* replace or remove those whose contribution is high (e.g., using dictionaries [190], charac-

ter level typos [102–104], or handcrafted strategies [105]). The adversarial training used in our work operates similarly, except our modifications are designed over programs.

PROGRAM REPRESENTATIONS A core challenge of using machine learning for code is designing a suitable program representation used as model input. Due to its simplicity, the most commonly used program representation is a sequence of words, obtained either by tokenizing the program [30] or by linearizing the abstract syntax tree [27]. This, however, ignores the fact that programs do have a rich structure – an issue addressed by representing programs as graphs [26, 43] or as a combination of abstract syntax tree paths [34]. In our work, we follow the approach proposed in recent works and represent programs as graphs. More importantly, we develop a novel technique that learns to refine the representation based on model predictions instead of fixing it a priori. As shown in our evaluation, this is crucial for learning robust models.

ADVERSARIAL ATTACKS FOR CODE Recall that in Chapter 3 we introduced a counter-example based training algorithm for learning static analyzers that generalize beyond the training dataset [5]. Concretely, we instantiated our approach with a range of semantic and non-semantic preserving modifications and an adversary that checked whether the learned analysis is sound (robust) to those. The only difference was that our model was not based on deep learning but instead, was a synthesized program in a domain-specific language.

Concurrent to our work, a number of works explored the task of generating adversarial attacks for variable renaming modifications and neural-based models, either via gradient based optimization [106] or using Metropolis-Hastings sampling [202]. In comparison, the key contributions of our work are focused on developing techniques that improve the model robustness, rather than on finding the adversarial examples. In particular, we propose to combine the ability to abstain with adversarial training and introduce an approach to reduce the problem complexity by learning to refine the representation (which also reduces the search space an adversarial attack needs to consider). In terms of program modifications, our work is not restricted to variable renamings and considers a richer class of modifications, including those that change the program structure. However, we do see these works as complementary to ours since finding adversarial examples efficiently and the ability to make models robust against them go hand in hand.

Furthermore, recent work of Ramakrishnan *et al.* [203] explored the notion of the robustness to a *k-adversary*. Here, the adversarial training considers only a single program modification ($k = 1$), which leads to an easier optimization problem compared to considering an arbitrary number of program modifications (as done in our work). However, the authors show that training with a single modification is already sufficient to improve the model robustness, even when evaluated with up to 5 modifications. Overall, we believe that techniques like [203] and [106, 202]

will soon become critical for scaling the robust training to ever increasing range of program modifications that are being considered.

TYPE INFERENCE We evaluated our work on the task of type inference for which a number of recent works improve accuracy by proposing a new neural architecture. In contrast, the goal of our work is to study and improve the robustness of these models. To achieve this, we compare to two prior works in our evaluation [30, 179]. In addition to predicting types from source code, Malik, Patra & Pradel [31] showed that it is possible to predict parameter types using natural language information obtained from method docstrings. Here, existing attacks on text (LSTM) can be used to assess its robustness but evaluating text models is outside the scope of our work. Finally, two concurrent works have proposed new models to improve accuracy: `Typilus` [32] and `LambdaNet` [33]. Both of these works represent programs as graphs and use graph neural networks as the underlying model architecture, which makes our approach applicable. However, we note that for `LambdaNet` we expect the model to be quite robust as the authors manually designed a sparse graph representation (by designing a static analysis to extract the type dependence graph) over which to learn.

4.8 CONCLUSION AND DISCUSSION

We presented a new technique to train *accurate* and *robust* neural models of code. Our work addresses two key challenges inherent to the domain of code: the difficulty of computing the correct label for all samples (i.e., the input is incomplete code snippet, program semantics are unknown) and the fact that programs are significantly larger and more structured compared to images or natural language.

To address the first challenge, we allow the model to abstain from making a prediction, rather than forcing the model to make predictions for all samples (as done in prior works). To address the second challenge, we learn which parts of the program are relevant for the prediction, and abstract the rest (instead of using the *entire* program as input).

Further, we introduce a new procedure that trains multiple models, instead of one. This has several advantages, as each model is simpler and thus easier to train robustly, the learned representation is specialized to the kind of predictions it makes, and the model directly conditions on predictions of prior models (instead of having to re-learn them). However, a disadvantage of our approach is that the models are learned sequentially which slows down the training (i.e., training 10 models will take $10\times$ more time). To speed up the training, it would be interesting to allow learning multiple models in parallel at each sequential step and then combine them as explored by Shazeer *et al.* [204].

We believe than our work is only one step in addressing the task of adversarially robust models of code and that many challenges remain open. For example, it remains to be seen how effective our approach is at other tasks over code, beyond type inference. Further, we optimize for the worst case adversarial robust-

ness, which corresponds to learning a robust model for all programs. An interesting future work is to optimize with respect to those modification that are common among developers, especially if it is not possible to be robust for all of them. While we checked robustness for a wide range of program modifications, these are still far from exhaustive and more work is needed in defining new ones. Finally, as the number of possible program modifications is large and growing, an interesting area is designing how they can be searched and combined more efficiently. In our work, the side-effect of learning to refine the program representation is that the search space of possible program modifications is reduced (since parts of the program become independent of each other). However, more work in area is needed, for example, as explored recently by Ramakrishnan *et al.* [203], Yefet, Alon & Yahav [106] or Zhang, Albarghouthi & D'Antoni [205].

5 FASTSMT: LEARNING STRATEGIES TO SOLVE FORMULAS

In Chapter 3 and Chapter 4 we have presented two approaches to achieve the same task – to learn robust and precise models that learn to predict program properties (e.g., type inference, points-to analysis), usually addressed via handcrafted static analyses. The difference between these approaches is that in Chapter 3, we showed how this can be achieved by learning an *interpretable program* in a domain-specific language that can be inspected and adapted by a domain expert. In contrast, Chapter 4 showed how training robust and precise models can be achieved by extending the latest state-of-the-art deep learning models, which are much harder to interpret but are also much more powerful and flexible.

THIS CHAPTER In this chapter, we present a new approach for learning to solve satisfiability modulo theories (SMT) formulas that combines the strengths of both deep learning as well as program synthesis techniques. Concretely, our approach works in two phases:

1. given a dataset of formulas whose solution is initially unknown, we first use reinforcement learning to train a neural policy that learns both how to solve these formulas as well as to solve them fast, and then,

2. we synthesize a loop-free program with branches that captures the policy decision making in an interpretable manner.

This combination allows us to train models that are extremely effective in practice – we solve 27% more formulas over a range of benchmarks and achieve up to 100$\times$ runtime improvement over a state-of-the-art SMT solver.

THE NEED FOR EFFICIENT SOLVERS SMT solvers are a powerful class of automated theorem provers that can deduce satisfiability and validity of first-order formulas in particular logical theories (e.g., real numbers, arrays, bit vectors). SMT solvers are more general than SAT solvers (which are restricted to the satisfiability of boolean formulas) and over the years have become and integral part in a variety of application domains including verification (e.g., neural networks [206]), program synthesis, static analysis, scheduling, and others [207].

To efficiently solve complex real-world problems, state-of-the-art SMT solvers (e.g., `Z3` [66], `Yices` [208], `CVC4` [107], `MathSAT5` [209], `Boolector` [210]) contain hand-crafted heuristics combining algorithmic proof methods and satisfiability search techniques. Indeed, crafting the right heuristic is critical and can be the difference between solving a complex formula in seconds or not at all.

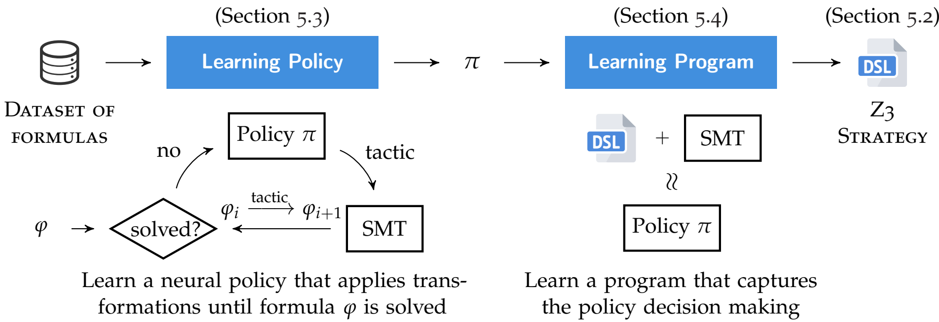


Figure 5.1: Main components of our approach to learn to solve SMT formulas.

LIMITATIONS OF HANDCRAFTED HEURISTICS Unfortunately, manually discovering heuristics that work well across a range of formulas can be difficult for several reasons: *(i)* fundamentally, one cannot anticipate all possible formulas a solver will be invoked with, meaning that as long as the solver's users cannot influence the heuristic construction (i.e., the heuristics are fixed by the developers and shipped to all the users), there will inevitably be cases where the performance will be suboptimal [211], *(ii)* even if one can anticipate all possible formulas (which is infeasible in practice since both the applications and formulas change over time), finding the right heuristic is still challenging and requires expert knowledge, and *(iii)* there is not a single best heuristic that works well for all the formulas. Instead, the suitable heuristics need to be selected dynamically based on the concrete formula and context in which it is invoked.

To address the first issue, several modern SMT solvers provide a mechanism for end users to control and define new combinations of heuristics that target a particular problem. For example, Z3 [66] allows users to define custom heuristics, called tactics, that the solver follows when processing a formula. Typically, such tactics are combined and performed in sequence (as specified by the user), forming an (interpretable) program called a *strategy*. However, the resulting strategies can end up being quite complex (e.g., contain loops and conditionals) and this fact, combined with the vast search space, means that manually finding a well-performing strategy can be very difficult, even for experts. As a result, while already highly tuned for many applications, the existing heuristics used in the state-of-the-art SMT solvers are often suboptimal in practice.

OUR WORK: FASTSMT We present a new approach for learning *strategies* to solve SMT formulas. We phrase the challenge of solving SMT formulas as a tree search problem where at each step a transformation is applied to the input formula until the formula is solved. Our approach works in two phases as illustrated in Figure 5.1: first, given a dataset of unsolved formulas we learn a policy that for each formula selects a suitable transformation (a tactic) to apply at each step in order to solve the formula, and second, we synthesize a *strategy* in the form of a loop-

free program with branches. This strategy is an interpretable representation of the policy decisions and is used to guide the SMT solver to decide if a formula is satisfiable more efficiently (i.e., to solve the formula).

Because the resulting strategy is represented as a program that the SMT solver supports, it can be directly integrated with the solver without requiring any additional modifications. Note that while often overlooked, the ease of integration is in fact critical for the system to be practically useful. Further, an added benefit of executing the synthesized program is zero runtime overhead during inference. This is in contrast to trying to evaluate a neural policy at each step. This would not only cause significant runtime overhead but also non-trivial software dependencies that need to be shipped alongside the solvers and would depend on the availability of specialized hardware such as GPUs. Finally, we note that our approach does not require any changes to the solver's internals and thus can work with any decision procedure that cleanly exports its tactics.

MAIN CONTRIBUTIONS Our main contributions are:

- A method that leverages classic learning models such as Bilinear Models and Imitation Learning for the task of discovering SMT strategies (represented as a neural policy). Importantly, our learning does not require any prior knowledge – we assume a dataset of formulas with unknown satisfiability and no existing strategies used to bootstrap the learning.
- A synthesis procedure that takes as input the learned model and generates an interpretable strategy in the form of a program that captures the model's decision-making.
- An extensive experimental evaluation of our approach on formulas of varying complexity across 3 different theories (`QF_NRA`, `QF_BV` and `QF_NIA`). We demonstrate the effectiveness of our approach by successfully synthesizing strategies that solve 27% formulas more and achieve up to 100× runtime improvement over state-of-the-art `Z3` solver [66].

OUTLINE We organize this chapter as follows. In Section 5.1 we give an overview of our approach and how SMT solvers work. In Section 5.2 we describe a domain-specific language `Strategy` used by `Z3` to enable control over the core solver heuristics. In Section 5.3 we describe how we train a neural policy that learns to solve SMT formulas. Then, in Section 5.4 we describe how we synthesize a program in the `Strategy` language that captures the neural policy decision making. In Section 5.5 we provide an experimental evaluation of our approach by applying it to improve the state-of-the-art SMT solver Z3. Finally, we describe the related work in Section 5.6 and provide a brief summary and discussion in Section 5.7.

5.1 OVERVIEW

At a high level, an SMT solver takes as input a first-order logical formula and then tries to decide if the formula is satisfiable. In the process, solvers employ various heuristics that first transform the input formula into a suitable representation and then use search procedures to check its satisfiability. Existing heuristics include:

- `simplify`, which applies basic simplification rules such as constant folding $(x + 0) \rightarrow x$ or removal of redundant expressions $(x - x) \rightarrow 0$,
- `gaussian_elim`, which eliminates variables $(x = 1 \wedge y \geq x + z) \rightarrow (y \geq 1 + z)$ using Gaussian elimination,
- `elim_term_ite`, which replaces the term if-then-else with fresh auxiliary declarations $\big((\texttt{if}\ x > y\ \texttt{then}\ x\ \texttt{else}\ y) > z\big) \rightarrow \big(k > z \wedge (x > y \implies k = x) \wedge (x \leq y \implies k = y)\big)$,
- `bit_blast`, which reduces bit-vector expressions by introducing fresh variables, one for each bit of the original bit-vector (e.g., a bit vector of size 4 is expanded into four fresh variables), and many more.

In total, the Z3 SMT solver defines more than 100 such heuristic transformations (called tactics) that can be combined together to define a custom strategy. For example, a strategy for integer arithmetic can be defined as[1]:

```
using simplify with (som: true); normalize_bounds;
        lia2pb; pb2bv; bit_blast; sat
```

Here, `normalize_bounds`, `lia2pb`, `bit_blast` and `sat` denote individual tactics while `using simplify with (som: true)` denotes a tactic `simplify` executed with a specific set of parameters, in this case using `som: true` that controls how polynomials should be represented. Although the above sequence of transformations (tactics) works well for some types of input formulas (e.g., in case every variable has a lower and an upper bound), for other formulas a different set of tactics is more suited. In some cases, the suitable set of tactics can be obtained by a small modification of the original tactic while in others, a completely different set of tactics needs to be defined. As a concrete example, consider the following strategy implemented in the Yices SMT solver [208, 211]:

$$\texttt{if}\ \left(\neg\texttt{diff} \wedge \frac{\texttt{num_atoms}}{\texttt{dim}} < k\right)\ \texttt{then}\ \texttt{simplex}\ \texttt{else}\ \texttt{floyd_warshall}$$

Here, two high level tactics can be applied to solve an input formula – the Simplex algorithm or the algorithm based on Floyd-Warshall all-pairs shortest distance. The Simplex algorithm is used if the input formula is not in the difference logic fragment (denoted using $\neg$`diff`) and the ratio of inequalities divided by the number of uninterpreted constants is smaller than a threshold k.

1 For more information and examples we refer the reader to the official Z3 tutorial available online at: https://rise4fun.com/z3/tutorial/strategies

PROBLEM STATEMENT Given a dataset of SMT formulas $\mathcal{F} = \{f_i\}_{i=1}^{n}$, our goal is to find a strategy expressed as a program in the domain-specific language `Strategy`:

$$\underset{q \in \texttt{Strategy}}{\arg\min} \sum_{f \in \mathcal{F}} cost(f,q) \quad \text{where} \quad cost(f,q) \stackrel{\text{def}}{=} \begin{cases} \texttt{runtime}(q(f)) & \text{if } q \text{ solves } f \\ t_{\texttt{timeout}} & \text{otherwise} \end{cases} \tag{5.1}$$

Here, $\texttt{runtime}(q(f)) \in \mathbb{Q}$ denotes the runtime required for strategy q to solve formula f and $t_{\texttt{timeout}} \in \mathbb{Q}$ is a constant denoting the penalty for not solving f (either due to timeout or because the strategy q is not powerful enough). Our goal is to find a strategy that minimizes the time required to solve the formulas in the dataset $\mathcal{F}$. Note that our *cost* function reflects the fact that we aim to synthesize a strategy that solves as many formulas as possible yet one that is also the fastest. Generally, optimizing for Equation 5.1 directly is problematic as using runtime makes the optimization problem inherently noisy and non-deterministic. It also makes the learning hard to parallelize due to significant impact of hardware and environment on overall runtime. Thus, instead of runtime, we use the amount of work measured as the number of basic operations performed by the solver required to solve a formula (e.g., implemented via the `rlimit` counter in `Z3`).

CHALLENGES There are two main challenges we address in our work:

- *Interpretability*. We are interested in learning a model that is not only efficient at solving a given set of formulas but also expressible as a program in the `Strategy` language. This is important, as the learned strategies can then be directly used as input to existing solvers.
- *No prior domain knowledge*. Our learning does not assume any prior knowledge about the dataset $\mathcal{F}$ and which strategies work best in general – initially the solution to all the formulas in the dataset is unknown, no existing strategies are used to boostrap the learning (not even the default strategies already written by the SMT solver developers) and we do not rely on any heuristics (and their combination) that may be useful in solving formulas from $\mathcal{F}$. Indeed, this represents the most challenging setting for learning.

OUR APPROACH The key idea behind learning a program $q_{best} \in$ `Strategy` efficiently is to take advantage of the fact that each program q can be decomposed into a set of smaller branch-free programs $q_{1,\ldots,k}$, each q_i corresponding to one execution path of q. This is possible because programs in the `Strategy` language do not contain state since the state is implicitly captured in the formula being solved. As a result, in our approach we learn the program q_{best} via a two step process:

- *Learn a neural policy*. First, we learn a policy which finds a set of candidate strategies consisting of only sequences of tactics where each strategy performs well on a different subset of the dataset $\mathcal{F}$. This allows us to phrase the learning problem as a tree search over tactics for which state-of-the-art models can be used. This step is described in Section 5.3.

- *Synthesize a combined strategy.* Then, given a set of candidate strategies, we combine these into a single best strategy q_{best} by synthesizing control structures such as branches and loops.This step is described in Section 5.4.

5.2 LANGUAGE FOR EXPRESSING SMT SOLVER STRATEGIES

We start by formally defining the language used by Z3 to enable control over the core solver heuristics, as shown in Figure 5.2. The core building blocks of the language are called tactics and represent various heuristic transformations that might be applied during the search. Optionally, each tactic defines a set of parameters that affect its behavior. For example, the `simplify` tactic contains >50 parameters that control which simplifications are performed (e.g., if `elim_and: true` then `simplify` rewrites conjunctions using negation and disjunctions). The tactics are combined into larger programs either sequentially or using one of the following control structures:

- `if` p `then` q_1 `else` q_2: If the predicate p evaluates to `true` apply q_1, otherwise apply q_2. The predicate can contain arithmetic expressions as well as `Probes` which collect statistics of the current formula (e.g., `num_consts` returns the number of non-boolean constants).
- q_1 `else` q_2: First apply q_1 and if it fails then apply q_2 on the original input.
- `repeat` q, c: Keep applying q until it does not modify the input formula any more or the number of iterations is greater than the constant c.
- `try` q `for` c: Apply q to the input and if it does not return in c ms then fail.
- `using` t `with` params: Apply the given tactic t with the parameters params.

(Strategy)	q	::=	t | q; q | if p then q else q | q else q | repeat q, c | try q for c | using t with params
(Tactics)	t	$\in$	Tactics = { bit_blast, solve_eqs, elim_uncnstr ... }
(Predicates)	p	::=	$p \wedge p \mid p \vee p \mid$ expr $\bowtie$ expr
(Expressions)	expr	::=	c | probe | expr $\oplus$ expr
(Constants)	c	$\in$	Consts = $\mathbb{Q}$
(Probes)	probe	::=	Probe $\rightarrow \mathbb{Q}$, Probe = { num_consts, is_pb, ... }
(AOperators)	$\oplus$	::=	$+ \mid - \mid * \mid /$
(BOperators)	$\bowtie$	::=	$> \mid < \mid \geq \mid \leq \mid = \mid \neq$
(Parameter)	param	::=	(Param, $\mathbb{Q}$), Param = { hoist_mul, flat, som, ... }
(Parameters)	params	::=	ϵ | param; params

FIGURE 5.2: Syntax of the `Strategy` language used to express SMT strategies in Z3 [66].

Goal: learn a policy $\pi(a \mid s)$ that selects the next action given the current state

Best Strategy: $q = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3$ which solves the formula in 12 seconds

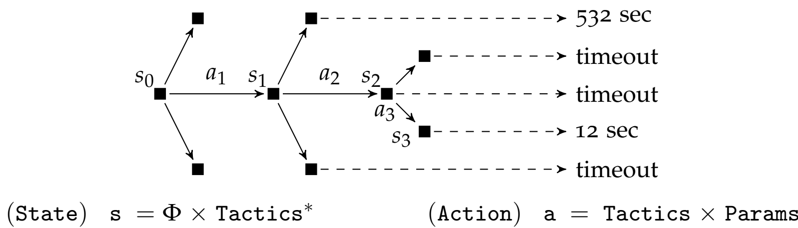


Figure 5.3: Illustration of how solving the formula can be phrased as a tree search problem. Here, the nodes correspond to states and edges correspond to actions (tactics) applied to transform the input formula. The goal of the neural policy is to select the next action to explore given the current state.

5.3 Learning a Neural Policy

We phrase the problem of learning candidate strategies as a tree search problem, as illustrated in Figure 5.3. We start from an initial state s_0 and at each timestep t we choose an action a_t that expands the tree by exploring a single edge. In our setting, a state corresponds to a tuple consisting of an SMT formula and a strategy used to compute the formula. An action is described by a tactic and its parameters (if any) $a \in \texttt{Tactics} \times \texttt{Params}$. Applying an action transforms the formula into another, equisatisfiable formula. Terminal states are those that decide the initial formula f, that is, f was reduced to a form that is either trivially satisfiable or unsatisfiable. Further, for practical reasons, terminal states also include those to which applying an action (tactic) leads to a timeout.

A terminal state defines a strategy q (i.e., a sequence of tactics starting from the tree root) with an associated $cost(f, q)$ (as defined in Equation 5.1) that we aim to minimize. In the example from Figure 5.3, the best strategy is a sequence of actions a_1, a_2, a_3 which results in solving the formula in 12 seconds. Our goal is to learn a policy, denoted as $\pi(a \mid s)$, that represents a probability distribution of actions conditioned on a state s. Ideally, the learned distribution should be such that selecting the most likely action at that state minimizes the expected $cost$. In what follows, we first describe the models used to represent the policy π considered in our work and then describe how the models are trained, including how to construct a suitable training dataset.

5.3.1 *Models*

We define two models – a simpler bilinear model and a more complex neural based policy. Further, in our evaluation we also include tree baseline models that perform random search, bread-first search and search using an evolutionary algorithm.

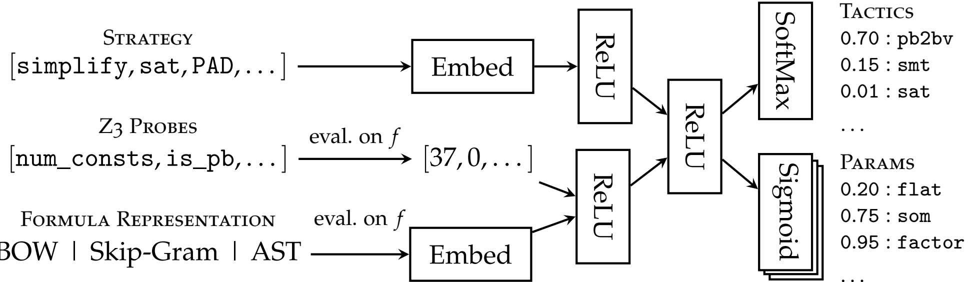


Figure 5.4: Neural network architecture used to predict next tactic and arguments.

bilinear model Based on matrix factor models used in unsupervised learning of word embeddings [70, 212], as well as an supervised text classification [213], we define a bilinear model as follows:

$$\pi(a \mid s) = \sigma(\boldsymbol{U}\boldsymbol{V}\phi(s,a)) \tag{5.2}$$

where $\phi(s,a)$ is a bag of features computed for an action a taken in state s, $\boldsymbol{U} \in \mathbb{R}^{k\times h}$ and $\boldsymbol{V} \in \mathbb{R}^{h\times|V|}$ are the learnable weight matrices, V is the input vocabulary and σ is softmax computing the probability distribution over output classes (in our case actions to be taken in a given state s). As the set of all possible actions is too large to consider, we randomly generate a subset of parameters for each tactic before training starts, thus obtaining the overall set of actions $S \subseteq \texttt{Tactics} \times \texttt{Params}$. We define $\phi(s,a)$ to be all n-grams constructed from the strategy of the state s. For example, for the strategy $a_1; a_2; a_3$, the extracted n-grams (features) are described by the vector: $\langle a_1, a_2, a_3, a_1a_2, a_2a_3, a_1a_2a_3\rangle$. Then, the vocabulary set V is simply the collection of all possible n-grams formed over S.

neural network Our second model is based on a neural network and improves over the bilinear model in two key aspects – it considers a richer state when making the predictions and predicts tactics, as well as their corresponding parameters. The architecture of the neural network model is illustrated in Figure 5.4 and uses two inputs: *(i)* the strategy in the current state (as in the bilinear model), and *(ii)* the formula f to be solved in the current state. The strategy is encoded by first padding it to be of fixed size, concatenating the embedding of each action in the strategy into a single vector and finally feeding the resulting vector into a fully-connected layer. We encode the input formula in two ways: first, by computing a set of formula measures such as the number of expressions and constants, and second, by learning a representation of the formula itself. The formula measures are computed using probes[2] supported by Z3 and are a subset of the possibilities

2 In our implementation, we use the following 16 probes – `num-consts`, `num-exprs`, `size`, `depth`, `ackr-bound-probe`, `is-qfbv-eq`, `arith-max-deg`, `arith-avg-deg`, `arith-max-bw`, `arith-avg-bw`, `is-unbounded`, `is-pb`, `num-bv-consts`, `num-arith-consts` and `is-qfbv-eq`.

Algorithm 5: Iterative algorithm used to train the policy π.

Input: Dataset of formulas $\mathcal{F}$, Number of iterations N, Number of formulas to sample K, Exploration rates $\boldsymbol{\beta}$, Exploration policy $\pi_{explore}$ (e.g., random policy)

Output: Trained policy π, Explored strategies $\mathcal{Q}$

1 $\mathcal{D} \leftarrow \emptyset$; $\mathcal{Q} \leftarrow \emptyset$; $\pi \leftarrow$ policy initialization
2 **for** $i = 1$ **to** N **do**
3   $\hat{\pi} \leftarrow \beta_i \pi + (1 - \beta_i)\pi_{explore}$ ▷ policy $\hat{\pi}$ explores with probability $(1 - \beta_i)$
4   $\mathcal{Q} \leftarrow \mathcal{Q} \cup (\bigcup_{f \in \mathcal{F}}$ `FindTopKUnseenStrategies`$(f, \hat{\pi}, K)$ $)$ ▷ Algorithm 6
5   $\mathcal{D} \leftarrow \mathcal{D} \cup$ Extract training dataset from strategies $\mathcal{Q}$
6   $\pi \leftarrow$ Retrain model π on $\mathcal{D}$
7 **return** $\pi, \mathcal{Q}$

one could define. For the learned representation of the formula we experimented with three different approaches:

- *Bag-of-words (BOW)*: The formula is treated as a sequence of tokens from the SMT-LIB language. We obtain its bag-of-words and use it as the formula embedding.
- *Abstract Syntax Tree (AST)*: From the formulas AST, we extract all subtrees of depth at most two. The bag-of-words over such subtrees is used as the formula embedding. Futher, as a form of regularization, we discard subtrees that appear in fewer than 5% of the formulas in $\mathcal{F}$.
- *Skip-gram*: Each formula in the dataset is treated as a sequence of tokens over which we learn a skip-gram model. We define the embedding of the formula as the average of all embeddings of its tokens.

The network output consists of two parts – a probability distribution over the tactics to apply next, and an assignment to each tactic parameter. The possible tactic parameters are captured by the set `Param` from Figure 5.2. We provide a full list of tactics and their parameters in Section 5.5. To compute arguments for the tactic parameters, the network introduces a separate output layer for each parameter type. The layer performs regression and outputs normalized values in the range $[0, 1]$. For boolean parameters, values 1 and 0 correspond to `true` and `false`, respectively. For integer parameters, the output of the network is mapped and discretized into the range of allowed values.

5.3.2 *Training*

Our training is based on the `DAgger` method [214] and is shown in Algorithm 5. The training starts with a randomly initialized policy used to sample the top K

Algorithm 6: Procedure for finding top K strategies.

```
def FindTopKUnseenStrategies(f, π, K)
    Input: Formula f, Model π, Number of strategies to sample K
    Output: Top K most likely unseen strategies according to the model π
1   global G ← ∅          ▷ contains all visited states throughout the training
2   s_0 ← (f, ϵ)          ▷ initial state is the formula without any actions applied
3   S ← ∅
4   queue ← PRIORITY_QUEUE()
5   queue.PUSH(s_0, ϵ, 1)
6   while |S| < K and |queue| > 0 do
7       (s_j, a_j, p_j) ← queue.POP()
8       s'_j ← state obtained by applying action a_j in state s_j
9       if s'_j ∉ G then
10          S ← S ∪ {s'_j.strategy};    G ← G ∪ {s'_j}
11      if ¬ PRUNED(s'_j) then
12          for a ∈ ACTIONS(s'_j, π) do
13              queue.PUSH(s'_j, a, p_j · π(a | s'_j))
14  return S
```

most likely unseen strategies for each formula in the training dataset $\mathcal{F}$ (line 4). The selected strategies are evaluated and used to create a training dataset (line 5) on which the policy π is retrained (line 6). As the model is initially untrained, the strategies are sampled at random and only as the training progresses the policy will learn to select strategies that perform well on formulas from $\mathcal{F}$. As an alternative, one could pre-train the initial policy using strategies supplied by an expert in which case the algorithm would correspond to imitation learning. However, in our work, we assume such expert strategies are not available and therefore we start with a model that is initialized randomly.

Finding most likely strategies A key step of the training presented in Algorithm 5 is a procedure that finds the top K most likely strategies to solve a given formula. Importantly, we are interested in returning strategies that were not yet explored during the training. This is because the already explored strategies are kept in the dataset (line 5 in Algorithm 5) so it is wasteful to explore the same strategies multiple times. This algorithm, shown in Algorithm 6, takes as input a single formula $f \in \mathcal{F}$, a model π (e.g., a neural network policy, bilinear model, etc.) and an integer K (the number of strategies to explore).

During the search we keep a priority queue of tuples (s_j, a_j, p_j) consisting of a state, a possible action and its associated probability, initialized with $(s_0, \epsilon, 1)$, s_0 denoting the initial state. At each step, we remove the tuple with highest probabil-

ity from the queue, apply its action to obtain a new state s'_j and update the priority queue with new tuples $(s'_j, a, p_j \cdot \pi(a \mid s'_j))$ for all actions $a \in \texttt{Tactics} \times \texttt{Params}$ capturing the possible transitions from s'_j. For practical reasons we approximate the set of all possible actions (denoted as $\textsc{actions}(s'_j, \pi)$) as follows:

- If we are using the neural network policy, we consider the most likely parameters for each tactic according to that policy, or
- If we are not using the neural network policy and instead are using models which do not predict parameters, we consider 20 different parameter configurations for each tactic which are selected at random before training starts.

We additionally perform pruning of states, which cannot possibly lead to an optimal strategy (line 9, described next). Finally, we note that in practice, we perform the search for batches of formulas at once in order to leverage the parallelization capabilities of our system.

PRUNING VIA EQUIVALENCE CLASSES A challenge in training the models presented above is that whether a strategy solves the formula is known only at terminal states. This is especially problematic for datasets where the majority of the effort is spent on finding any successful strategy. To address this issue, we take advantage of the fact that some information can be learned also from partial strategies – namely their current runtime and their transformed formula. This allows us to check if multiple transformed formulas are equivalent (we consider two formulas equivalent if their abstract syntax tress are identical) and keep only the one which was fastest to reach (and prune the others).

BUILDING THE TRAINING DATASET Each sample in our training dataset $\mathcal{D} = \{\langle(\boldsymbol{t}_i, \boldsymbol{p}_i), s_i\rangle\}_{i=1}^{M}$ for the neural network consists of a state and its associated training label. Here, the label is a tuple consisting of a probability distribution over tactics $\boldsymbol{t} \in \mathbb{R}^{|\texttt{Tactics}|}$ and the values of all tactic parameters $\boldsymbol{p} \in \mathbb{R}^{|\texttt{Param}|}$. The intuition behind this choice is that for a state s, the vector $\boldsymbol{t}$ encodes the likelihood that a given tactic is fast at solving the formula whereas $\boldsymbol{p}$ contains the best parameter values found so far during training. Generating the dataset in this way encodes the preference for strategies that are most efficient in contrast to finding any strategy that solves the input formula. To train the neural network model using such a dataset, the loss is constructed as a weighted average of the cross-entropy loss for tactic prediction and mean-squared-error for parameter prediction.

We build the dataset as follows. First, we evaluate each strategy in $\mathcal{Q}$ on the formula for which it was generated and keep only those that succeeded in solving the formula. Second, let us denote with $r(t, s_i)$ the best runtime (or `timeout`) achieved from state s_i by first applying tactic t, with $r_{best}(s_i)$ the best runtime achieved from state s_i, and with $v(p, s_i)$ the value assigned to parameter p in tactics with the best runtime from state s_i. We obtain r, r_{best} and v by considering all the states and actions performed when evaluating the strategies in $\mathcal{Q}$. Then, for each non-terminal

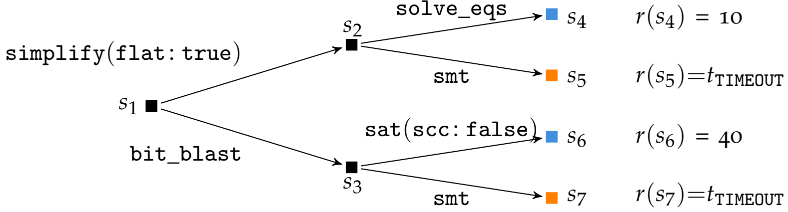


FIGURE 5.5: Example with all states visited while solving a formula, including the runtime associated with each terminal state. Terminal states are colored blue (■) if formula was solved, and orange (■) otherwise.

State	Target tactics	Target parameters
$s_1 = (\varphi_1, \epsilon)$	$\Pr(\texttt{simplify}) = 0.8$	$\texttt{flat} = \texttt{true}$
	$\Pr(\texttt{bit_blast}) = 0.2$	-
$s_2 = (\varphi_2, \texttt{simplify}(\texttt{flat} = \texttt{true}))$	$\Pr(\texttt{solve_eqs}) = 1$	-
$s_3 = (\varphi_3, \texttt{bit_blast})$	$\Pr(\texttt{sat}) = 1$	$\texttt{scc} = \texttt{false}$

TABLE 5.1: Training dataset constructed from the example shown in Figure 5.5.

state s_i that eventually succeeded, we create one training sample $\langle(\sigma(\tilde{\boldsymbol{t}}_i), \boldsymbol{p}_i), s_i\rangle$ where $\tilde{\boldsymbol{t}}_i = [1/r(t, s_i)]_{t \in \texttt{Tactics}}$, σ normalizes $\tilde{\boldsymbol{t}}_i$ to a valid probability distribution and $\boldsymbol{p}_i = [v(p, s_i)]_{p \in \texttt{Param}}$. To generate the training dataset for bilinear model we follow the same steps except that we use $\tilde{\boldsymbol{t}}_i = 1[r_{best}(s_i) = r(t, s_i)]_{t \in \texttt{Tactics}}$ which assigns probability 1 to the best tactic and 0 to others.

Example Let us consider an example where the model explored seven different states as shown in Figure 5.5. Further, in addition to the visited states we also keep information about the cumulative runtime required to compute the state (starting from the initial state s_0), as well as whether the state successfully solves the formula. Then, the dataset is constructed by generating one training sample $\langle(\sigma(\tilde{\boldsymbol{t}}_i), \boldsymbol{p}_i), s_i\rangle$ for each non-terminal state that eventually succeeded in solving the formula. In our example, this corresponds to generating one training sample for all three non-terminal states s_1, s_2 and s_3.

The resulting dataset is shown in Table 5.1, where we use ϵ to denote that no tactic was applied so far. For state s_1, there are two possible tactics which lead to solving the formula ($\texttt{simplify}(\texttt{flat} = \texttt{true})$ and bit_blast). However, the best strategies in the respective subtrees have different runtimes, hence the probabilities assigned to the corresponding tactics are different and reflect the fact that using $\texttt{simplify}(\texttt{flat} = \texttt{true})$ is significantly faster compared to using bit_blast. Finally, in case both states s_6 and s_7 would not solve the formula, it would mean that no training example is generated for state s_3 (since we generate training samples only for non-terminal states that eventually succeeded in solving the formula).

5.4 SYNTHESIZING A COMBINED, INTERPRETABLE STRATEGY

The policy π learned in Section 5.3 can be used to extend the existing SMT solvers as follows: invoke the solver with the current formula f and the action a_0 selected by π, obtain a new intermediate formula f_1, then again invoke the solver from scratch with f_1 and a new action a_1, obtaining another intermediate formula f_2, and so on. While possible, wrapping existing SMT solvers in such iterative loop is very problematic in practice because: *(i)* executing the neural policy at each step would incur significant run-time overhead and introduce additional hardware requirements (e.g., GPU support), and *(ii)* more importantly, we would have lost the connection with the original formula f, as at each step we are making a new, fresh invocation of the SMT solver. This is problematic for tasks (e.g., planning) that require more information from SMT solvers, beyond satisfiability of f, for instance, the model itself. To address both of these challenges, we synthesize an interpretable policy q_{best} that follows π and can be expressed in the `Strategy` language from Figure 5.2. Programs in this language can be then given directly as input to the SMT solver, without having to modify the solver internals in any way.

Recall from Figure 5.2 that the `Strategy` language defines two types of statements used to combine programs: `if-then-else` and `or-else`. However, notice that the `or-else` is in fact a special version of the `if` statement with a condition that checks if a given tactic terminates within `c` milliseconds. As a result, we can reduce the problem of synthesizing programs in the `Strategy` language to the problem of synthesizing branches with predicates over a set of candidate strategies.

Note that this is the same language fragment that we used in Chapter 3 to learn $\mathcal{L}_{pt}$ and $\mathcal{L}_{alloc}$ programs. However, the synthesis problem considered in this section is easier compared to Chapter 3. This is because here, we trained a neural policy to obtain a set of candidate strategies (straight-line programs in `Strategy`) and the goal of the synthesis is to combine them via branches with predicates. In contrast, in Chapter 3 the programs were learned directly with the goal of optimizing the performance on a given dataset, without any access to good building blocks from which they can be composed.

SYNTHESIZING q_{best} To synthesize branches with predicates, we adapt the techniques based on decision tree learning. Consider the tree illustrated in Figure 5.6 (top left) with the same structure as the one used during the search (i.e., edges denoting actions and nodes denoting states) but formed from the two candidate strategies q_1 and q_2. Each SMT formula is evaluated by taking a path in the tree and applying all the corresponding actions. Intuitively, at each node with more than one outgoing edge a decision needs to be taken to determine which path to take. To encode such decisions, our goal is to introduce one or more branches at each such node, denoted as orange square (■) in Figure 5.6.

Formally, let $\mathcal{Q} = \{q_i\}_{i=1}^{n}$ be a set of candidate strategies, $\mathcal{F}$ a dataset of formulas and $b ::=$ `if p then` $\boldsymbol{q}_{\texttt{true}}$ `else` $\boldsymbol{q}_{\texttt{false}}$ a branch that partitions $\mathcal{F}$ into two parts – $\mathcal{F}_{\texttt{true}}$ are formulas on which predicate `p` evaluates to true and $\mathcal{F}_{\texttt{false}}$ for the rest.

a3
a2
a1
a5
a4
strategies q_1 and q_2

	f_1	f_2	f_3
$q_1 = \mathtt{a}_1;\mathtt{a}_2;\mathtt{a}_3$	10	100	100
$q_2 = \mathtt{a}_1;\mathtt{a}_4;\mathtt{a}_5$	30	TIMEOUT	20

formulas and their associated costs

f_1	f_2	f_3
200	250	30

num_expr

formula measure

`if true then` $\mathtt{a}_2$ `else _` Branch Cost: $0.276 = (-1/3 \cdot \log(1/3) - 2/3 \cdot \log(2/3)) + 0$

`if num_expr > 100 then` $\mathtt{a}_2$ `else` $\mathtt{a}_4$ Branch Cost: $0.2 = -2/3 \cdot \log(1/2) + 1/3 \cdot 0$

Figure 5.6: Illustration of the decision tree representation of strategies, scored formulas, their measures and two sample programs in `Strategy` with their corresponding branch scores (according to Equation 5.4).

To obtain $\mathcal{Q}$, we evaluate the learned policy π over the training formulas, as well as collect all the successful strategies explored during learning. We define a notion of multi-label entropy of a dataset of formulas, denoted as $H(\mathcal{F})$ [215]:

$$H(\mathcal{F}) = -\sum_{q \in \mathcal{Q}} p(q)\log(p(q)) + (1 - p(q))\log(1 - p(q)) \qquad (5.3)$$

where $p(q)$ denotes the ratio of formulas solved by strategy q in $\mathcal{F}$. The goal of synthesis is then to discover branches that partition $\mathcal{F}$ into smaller sets, each of which has small entropy (i.e. either none or all formulas are solved). Using the entropy, we define a cost associated with a branch b as:

$$cost(b, \mathcal{F}_{\mathtt{true}}, \mathcal{F}_{\mathtt{false}}) = \frac{|\mathcal{F}_{\mathtt{true}}|}{|\mathcal{F}|} H(\mathcal{F}_{\mathtt{true}}) + \frac{|\mathcal{F}_{\mathtt{false}}|}{|\mathcal{F}|} H(\mathcal{F}_{\mathtt{false}}) \qquad (5.4)$$

That is, the branch cost is a weighted sum of entropies in each of the resulting branches. With this scoring function we build the decision tree in a top-down fashion – for each node with multiple outgoing edges we recursively synthesize predicates that minimize the *cost*. If the dataset size for some node is below a certain threshold, we greedily select the strategy which can solve the most formulas, breaking ties using runtimes. To express branches, we consider the following types of predicates: *(i)* `true` which allows expressing a default choice, *(ii)* `Probes` with arithmetic expression as defined in Figure 5.2, and *(iii)* `try s for c` which allows checking whether the tactics terminate within `c` ms.

Example Consider the example shown in Figure 5.6 that contains two candidate strategies q_1 and q_2 and a dataset consisting of three formulas $\mathcal{F} = \{f_1, f_2, f_3\}$. By evaluating both strategies on all the formulas in the dataset we can see that q_1 solves all three formulas while q_2 solves only two (f_1 and f_2). However, while q_1 solves f_3 in 100 seconds, the same formula can be solved by q_2 in only 20 seconds. Since q_1 is faster on all the remaining formulas, the best possible program in this example would be – select q_1 for f_1 and f_2, but select q_2 for f_3. To achieve this, we evaluate a number of formula measure that can serve as predicates in the synthesized program. In Figure 5.6 (top right), we can see an example of a measure

Algorithm 7: A procedure for selecting a subset of the candidate strategies.

Input: Set of formulas $\mathcal{F}$, Strategies $Q = \{q_i\}_{i=1}^{m}$, weight $\lambda \in [0,1]$, $K \in \mathbb{Z}^+$
Output: K selected strategies

```
1  Q_best ← ∅;   A ← F;   B ← ∅
2  while |Q_best| < K do
3      q_best ← ε;   A_best ← ∅;   B_best ← ∅;   score_best ← 0
4      for q_i ∈ Q do
5          if q_i ∉ Q_best then
6              A_i ← subset of formulas in A which strategy q_i can solve
7              B_i ← subset of formulas in B which strategy q_i can solve
8              score_i ← λ|A_i| + (1 − λ)|B_i|
9              if score_i > score_best then
10                 q_best ← q_i;   A_best ← A_i;   B_best ← B_i;   score_best ← score_i
11     Q_best ← Q_best ∪ q_best;   A ← A \ A_best;   B ← B ∪ A_best
12 return Q_best
```

that computes the number of expressions in each formula. Using this measure we can express our desired program as `if num_expr > 100 then a2 else a4`.

Additionally, Figure 5.6 (bottom) shows the branch score computed for this program according to Equation 5.4. Given that the branch above is optimal, it should hold that the score for the other branch is worse (in this case higher). We can check that this is the case by comparing the score to a branch that uses strategy q_1 for all the formulas `if true then a2 else _`. Indeed, by computing the scores for both programs we can see that the optimal program achieves score 0.2 which is better than 0.276 for the program that always uses strategy q_1.

SCALING THE SYNTHESIS TO LARGE DATASETS As the set of candidate strategies can be large, especially when the dataset contains a large number of formulas, we perform synthesis only on a subset set of strategies. Intuitively, these strategies should be: strong individually (i.e. each strategy should be able to solve a large number of formulas alone) and strong together (i.e. the number of formulas solvable by at least one of the strategies should be large). In order to trade-off these two conditions we use a greedy procedure shown in Algorithm 7.

We proceed in an iterative manner, choosing one new strategy at every step. In every iteration, each strategy receives a score for every formula it solves. The score is equal to λ for every formula which was previously unsolved, and $1 - \lambda$ for each formula that was already solved (by another previously selected strategy). One can notice that if $\lambda = 1$, the algorithm will greedily maximize the number of formulas that the strategies can solve in the union. If $\lambda = 1/2$, the algorithm will simply select k strategies that solve the most formulas individually. In our experiments, we treat λ as a hyperparameter and optimize it on a validation set of formulas.

5.5 EVALUATION

We implemented our method in a system called FastSMT and evaluated its effectiveness by learning to solve formulas of varying complexity across 3 logics QF_NRA, QF_BV and QF_NIA and 5 benchmarks. The main results of our experiments are that:

- Our approach successfully learns a neural policy that solves 27% more formulas than the handcrafted heuristics included in the state-of-the-art SMT solver Z3 [66]. This is even though our training has not prior knowledge – the solution of all training formulas was unknown and existing Z3 heuristics were not used to bootstrap the learning.
- We synthesized a program in the Strategy language that mimics the policy decision making and solves 9% more formulas and up to three orders of magnitude faster than Z3. In contrast to the non-interpretable neural policy, this program is directly incorporated into Z3 without any modifications required.
- We show that our learned models generalize well to significantly harder formulas that those seen during training. Concretely, a model trained with only 10 second time limit generalizes well to formulas that require up to 10 minutes to solve and successfully solves 8.6% more formulas than Z3.

To train our models, we use 10 iterations of Algorithm 5 with exponentially decaying exploration rate to choose between policy and random action. We train the bilinear model using FastText [213] for 5 epochs with learning rate 0.1 and 20 hidden units. We implemented the neural network model in PyTorch [191] and trained it using a learning rate of 10^{-4}, mini batch size 32, Adam optimizer [196] with Xavier initialization [216] and early stopping. Further, we train all our models using time limit of 10 seconds allocated for solving each formula. Unless specified otherwise, we also use 10 second time limit when evaluating the models. We make all the models, datasets and the code available at:

https://github.com/eth-sri/fastsmt

SMT BENCHMARKS We evaluated the effectiveness of FastSMT on 5 different datasets – AProVE [217, 218], hycomp [219], core [220], leipzig [221] and Sage2 [222, 223]. These contain formulas of varying complexity across 3 logics QF_NRA, QF_BV and QF_NIA. The formulas have on average 336, 35, 228, 153, 345 assertions, 929, 10 690, 1 672, 886, 1 887 expressions and 118, 49, 46, 20, 79 variables for leipzig, core, hycomp, AProVE and Sage2 benchmarks, respectively and require up to 60 MB for each formula to be stored in the SMT2-lib format. All datasets are part of the official SMT-LIB benchmarks [224] and encode formulas from both academic and industrial tools (i.e. Sage and AProVE). We split the formulas in training, validation and test sets in predetermined ratios for all datasets.

	Formulas solved				Speedup percentile		
Benchmark	Both	Only Z3	Only FastSMT	None	P_{90}	P_{50}	P_{10}
Best learned strategy (Section 5.3)							
leipzig	57	0	3	8	5.8×	60.7×	191.5×
Sage2	2 531	0	2 402	1 511	1.2×	2.5×	22.0×
core	270	0	0	0	1.2×	1.3×	1.9×
hycomp	1 633	1	210	138	1.0×	2.0×	4.3×
AProVE	1 397	3	221	91	4.0×	89.6×	988.7×
Total	56.2%	0.1%	27.0%	16.7%	2.6×	31.2×	241.7×
Synthesized program (Section 5.4)							
leipzig	55	5	1	7	1.4×	9.9×	21.7×
Sage2	2 488	200	705	3 051	0.8×	1.2×	3.1×
core	270	0	0	0	1.2×	1.3×	1.8×
hycomp	1 547	93	112	230	0.4×	1.1×	2.3×
AProVE	1 365	76	112	159	3.2×	65.1×	860.8×
Total	54.6%	3.6%	8.9%	32.9%	1.4×	15.7×	178.0×

Table 5.2: Comparison of the quality of the strategies found by our work against Z3.

5.5.1 *Comparison to the State-of-the-art SMT Solver* Z3

We evaluate the effectiveness of the learned strategies compared to the hand-crafted strategies in Z3 4.6.2 on two metrics – number of solved formulas and speedup. We compute the speedup by counting the amount of executed basic operations (using `rlimit` counter in Z3) as a deterministic and machine independent measure of the work required to solve the formula.

Table 5.2 *(top)* shows the number of formulas solved by Z3 compared to the best strategy found by any of our methods. We also measure the speedups of our strategies over Z3 on all the formulas which were solved by both methods. For example, the 90th percentile (P_{90}) in the AProVE benchmark denotes that for 90% of the formulas the speedup is at least 4.0×. The learned strategies significantly outperform Z3 across all benchmarks – solving 27% more formulas, with up to 3 orders of magnitude speedups and with only 4 formulas solved by Z3 but not by any of our learned strategies. This shows that, for all our benchmarks, the strategies found during training generalize well to other formulas from the same dataset.

Table 5.2 *(bottom)* shows the performance of the single combined strategy synthesized as described in Section 5.4. Here, the result of synthesis is a program in the `Strategy` language that is used as input to Z3 together with the formula to solve. Naturally, as the `Strategy` language has limited expressiveness (i.e., restrict-

	Formulas solved				Speedup percentile		
Benchmark	Both	Only Z3	Only FastSMT	None	P_{90}	P_{50}	P_{10}
10 second time limit							
leipzig	57	3	0	8	5.8×	60.7×	191.5×
Sage2	332	2	246	220	1.2×	2.7×	35.5×
core	270	0	0	0	1.2×	1.3×	1.9×
hycomp	273	0	34	18	1.0×	1.8×	4.0×
AProVE	283	0	18	14	3.9×	87.8×	1 314.0×
Total	68.3%	0.3%	16.8%	14.6%	2.6×	30.9×	309.4×
10 minutes time limit							
leipzig	63	0	1	4	3.5×	43.9×	183.2×
Sage2	630	0	138	32	1.3×	6.5×	199.6×
core	270	0	0	0	1.2×	1.3×	1.9×
hycomp	298	0	10	17	1.0×	2.0×	40.1×
AProVE	306	0	3	6	3.9×	89.3×	1301.5×
Total	88.1%	0.00%	8.6%	3.3%	2.2×	28.6×	345.3×

Table 5.3: Comparison of the best strategy found by any of our models against Z3 with 10 seconds *(top)* and 10 minutes *(bottom)* time limit for testing. Note that all models are still trained only with 10 seconds time limit.

ing the kind of expressions that can be used as branch predicates) the performance improvement is smaller compared to the best strategy found by any of our methods for each formula as shown in Table 5.2 *(top)*. However, the improvement over the default Z3 strategy is still significant – not only our synthesized program solves formulas faster, it also solves 8.9% more formulas in total.

generalization to harder to solve formulas The 10 seconds time limit in our experiments was selected for practical purposes - it is large enough to solve the majority of the formulas and to learn the strategies in a reasonable time. To check how well our strategies generalize to higher time limits we kept the 10 seconds time limit for training, but used 10 minutes for evaluation. We show the results from this experiment in Table 5.3. With the 10 minutes time limit, 88.1% of the formulas are solved by both methods. Crucially, our strategies are still able to solve 8.6% more formulas than Z3. Overall, with the increased time limit, our method can solve 97.7% of the formulas (up from 85.1% with the 10 seconds time limit). Further, the speedups over Z3 are comparable (and even slightly higher) to the speedups achieved with the 10 seconds time limit. Note since the experiments take significantly longer to run, we evaluated them only on a subset of the dataset.

	Speedup - number of operations				Speedup - wall clock time			
Benchmark	P_{90}	P_{50}	P_{10}	Mean	P_{90}	P_{50}	P_{10}	Mean
leipzig	3.5×	43.9×	183.2×	71.6×	0.3×	2.3×	7.3×	3.5×
Sage2	1.3×	6.5×	199.6×	62.7×	1.2×	4.8×	72.5×	37.8×
core	1.2×	1.3×	1.9×	1.4×	0.5×	0.8×	1.3×	0.9×
hycomp	1.0×	2.0×	40.1×	51.9×	0.9×	1.4×	65.7×	25.0×
AProVE	3.9×	89.3×	1 301.5×	519.9×	0.9×	6.4×	120.8×	45.8×
Total	2.2×	28.6×	345.3×	141.5×	0.8×	3.1×	53.5×	22.6×

Table 5.4: Comparison of speedup in number of operations and wall clock time.

NUMBER OF BASIC OPERATIONS AND RUNTIME So far, all our experiments used the number of basic operations as a deterministic measure of the amount of work required to solve a formula. In Table 5.4, we show a comparison between the number of operations and the wall clock time for the experiments in Table 5.3. We can see that on average (P_{50}), the wall clock runtime of the learned strategies is 3.1× faster compared to the Z3 solver. However, this speedup is significantly smaller than the corresponding speedup of the basic operations. The reasons for the smaller speedup of the wall clock time are two fold: *(i)* for formulas that can be solved very fast, wall clock time mostly accounts for the initialization of the solver and the overhead of the system, rather than the actual time it takes to solve the formula, and *(ii)* while the number of operations reported by Z3 do correlate with the runtime, they are only an approximation which in our case, seems to overapproximate the actual speedup.

5.5.2 *Effectivness of the Learned Models*

To compare bilinear model and the neural network based policy defined in Section 5.3, we trained both models and used them to obtain the 100 most likely strategies for each formula in our test dataset. The results are shown in Figure 5.7 and additionally include three baseline models that perform random search, breadth-first search and search using an evolutionary algorithm. The x-axis shows the number of most likely strategies sampled from each model and the y-axis their runtime proportional to the runtime of the best known strategy (i.e., best strategy from the top 100 strategies across all models). Here, score one denotes that the best known strategy was found, whereas score zero denotes that no strategy was found that solves the formula. Even though the baselines perform poorly, they are already able to find simpler strategies that can solve some of the formulas. Note that in our experiment, the evolutionary algorithm performed similarly to a random model, as it got easily stuck in a local minima without enough exploration.

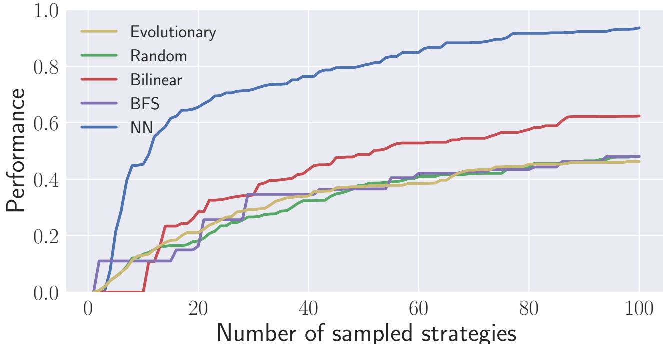


Figure 5.7: Comparison of our models (bilinear and NN) and baselines for learning SMT solver strategies. Each line (higher is better) denotes the quality of the best learned strategy among top x most likely strategies found by a given model (x axis) proportional to the best known strategy overall (y axis).

Overall, the best model is the neural network policy (NN), which is also most complex and considers the richest set of features.

THE EFFECT OF STATE REPRESENTATION In Figure 5.8 we evaluate the effect of instantiating the neural network model with a different set of input features capturing the current state. For our task, the representation at the right level of complexity is bag-of-words – if the formula representation is flattened using pre-trained embeddings it loses the relevant information and with more complex AST features the model suffers due to data sparsity. Further, we note that for our task the most important features are those capturing the sequences of tactics applied so far, which is illustrated by the strong performance of `Strategy` only model.

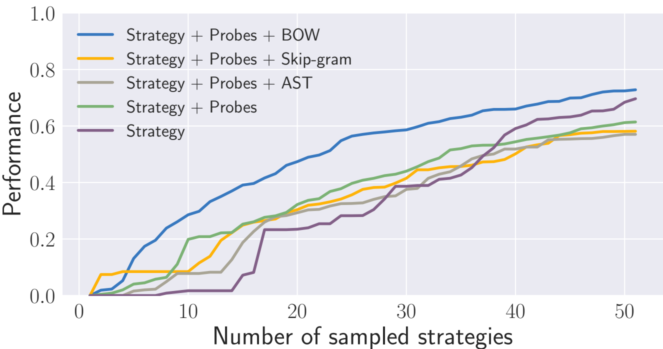


Figure 5.8: The effect of different state representation used to train the neural policy.

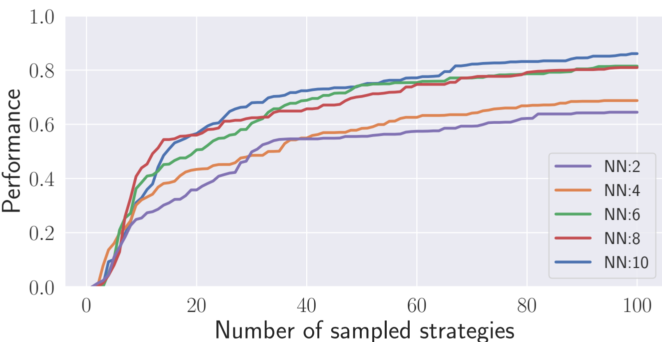


FIGURE 5.9: Performance of the neural policy after 2, 4, 6, 8 and 10 steps of Algorithm 5.

THE EFFECT OF ITERATIVE TRAINING Finally, in Figure 5.9 we show the improvement of our neural network policy as it is continuously retrained using the `DAgger` method [214]. We perform a total of 10 stages of `DAgger`, retraining the model after every stage. In every stage, the current model is used to search for the best strategies, as described in Algorithm 6. For the purpose of this experiment, we save the models after every 2 stages and then load each model again and use it to search for the best strategies on the unseen formulas from the test set. We run all of the models for 100 iterations without retraining (which means that each model predicts 100 best strategies for every formula). One can notice that later models tend to outperform earlier models, thus justifying the increased number of training stages. The only exception in this case are the models trained after 6 and 8 stages, where an earlier model performs better by a small margin. This can be explained by the stochastic nature of the training procedure.

5.6 RELATED WORK

Given the importance and wide range of SMT solver applications, a number of approaches have been developed to improve both their runtime, as well as the range of formulas they can solve.

PORTFOLIO BASED APPROACHES The most common approach of tools such as `SATzilla` [225], `CPhydra` [226], `SUNNY` [227], `Proteus` [228], `ISAC` [229] is a portfolio based method. The key idea is that different SMT solvers use different heuristics and hence work well for different types of formulas. Then, given several SMT solvers and a dataset of formulas, the goal is to learn a classifier that uses features extracted from the given formula to select the right SMT solver (or alternatively defines the order in which to run the solvers). In comparison, we address a harder problem - we learn how to instantiate an SMT solver with a strategy that efficiently solves the given dataset of formulas.

EVOLUTIONARY SEARCH A closely related work to ours is `StratEVO` [230] which also studies the task of generating SMT strategies. However, `StratEVO` has several limitations – it performs the search using an evolutionary algorithm, which does not incorporate any form of learning, the search does not depend on the actual formula, and local mutations tend to get stuck in local minima (as we show in our experiments in Section 5.5). As a result, such approach reduces to random search for many tasks where a suitable strategy cannot be trivially found. Instead, we leverage models that learn from previously explored strategies as well as the current formula. As we show, this enables us to discover complex strategies that are out of reach for approaches not based on learning.

LEARNING BRANCHING HEURISTICS AND PREMISE SELECTION Recently, several learning techniques have been applied for improving the performance of SAT solvers [231, 232], constraint programming solvers [233], solving quantified boolean formulas [234], solving mixed integer programming problems [235] as well as premise selection in interactive theorem provers [236–238]. At a high level, these are complementary to us – we learn to search across many tactics and combine them into high level strategies, while they optimize a single tactic (e.g., by learning which variable to branch on in SAT). Our work also supports formulas from multiple theories (as long as there is a corresponding tactic language) where selecting a suitable high level strategy leads to higher speedups compared to optimizing a single tactic in isolation. However, there are also common challenges, such as defining a suitable formula representation. This representation can range from a set of hand-crafted features [231, 239], to recursive and convolutional neural networks [236, 238], to graph embeddings [237]. We extend this line of work by considering fast to compute representations based on bag of words, syntactic features, and features extracted from a graph representation of the formula.

PARAMETER TUNING Finally, a number of approaches exist for finding good parameter configurations from a vast space of both discrete and continuous parameters, including `ParamILS` [240], `GGA` [241], `TB − SPO` [242] or `SMAC` [243]. An interesting application of such approaches to the domain of SAT/SMT solvers is proposed by `SATenstein` [244], which first designs a highly parameterized solver framework to be subsequently optimized by off-the shelf tools. Although such tools are not applicable for the task of searching for strategy programs (that include loops and conditionals) considered in our work, they can be used to fine-tune the strategy parameters once a candidate strategy is found.

5.7 CONCLUSION

We presented a new approach for improving the performance of state-of-the-art SMT solvers based on a combination of training a policy that learns to discover strategies that solve formulas efficiently and a synthesizer that produces interpretable strategies based on this model. The synthesized strategies are represented

```
(declare-const x Int)
(declare-const y Int)
...
(assert (...))
(assert (...))

(check-sat)
```

(a) Formula solved by the default Z3 strategy

```
(declare-const x Int)
(declare-const y Int)
...
(assert (...))
(assert (...))

(check-sat-using (fastsmt-strategy))
```

(b) Formula solved by our learned strategy

FIGURE 5.10: Example which shows that integrating our learned strategy with Z3 corresponds to changing a single line of code.

as programs with branches that can be directly used by the developers, without requiring any changes to the underlying solver. Concretely, to solve a formula with our learned strategy, it is sufficient to change a single line of code in the formula definition as illustrated in Figure 5.10. Here, we replace the line `(check-sat)`, which corresponds to solving the formula using the default Z3 strategy, with `(check-sat-using (fastsmt-strategy))`, which solves the formula using our learned strategy. This avoids the need to evaluate the learned policy at inference time and ensures that all the SMT solver functionality is still supported (e.g., retrieving a model or computing the unsat core).

We demonstrate the practical usefulness of our approach by learning strategies that consistently improve over the Z3 SMT solver on formulas from 5 official SMT-LIB benchmarks of varying complexity. Concretely, when selecting the best strategy per formula, we solve 27% more formulas and achieve on average $\approx 31\times$ runtime improvements over Z3. When synthesizing the best strategy for the whole benchmark, we solve 8.9% more formulas with a speedup of $\approx 16\times$ on average. An interesting future work would be to close this gap between best strategies learned using a neural policy and the strategies that can be represented in the Strategy language supported by Z3. For example, one could try to learn new probes and formula measures that can be used as predicates in the Strategy language, thus increasing its expressiveness. This also partially addresses the main limitation of our approach, the fact that we are limited by the fragment of the SMT theories for which suitable tactics have to be predefined by the authors of Z3. Another future direction would be to automatically learn the transformations (i.e., tactics) that preserve satisfiability of formulas but improve the solver performance. Such transformations have been used by a subsequent work for the complementary problem of checking the correctness of SMT solvers [245, 246]. These could include rewriting parts of the formulas using an API function that exists but the user was not familiar with (e.g, PbEq), as well as learning a probabilistic model that selectively restricts the formula to make it easier for the solver to prove[3].

3 In the next chapter, we will discuss a handcrafted instantiation of this idea as one of the techniques that makes our approach scalable (Section 6.5.1)

6 INFERUI: FROM IMAGES TO LAYOUTS

So far, our focus was on building new techniques and tools that predict program properties. Concretely, given a program x as input, we learned a number of probabilistic models of code (Chapter 2), robust models that perform type inference (Chapter 4), compute points-to or allocations site analysis (Chapter 3), or learn strategies to solve SMT formulas (Chapter 5). To achieve this, we trained our models using two perspectives: *(i)* a machine learning perspective that *learns over programs*, where the model is parametrized by a set of weights θ (typically corresponding to the weights of a deep learning model), and *(ii)* a programming languages perspective, where we *learn programs* which are either used directly to compute the results (e.g., as done when learning a static analyzer in Chapter 3) or are used to *parametrize* a machine learning model (as in Chapter 2).

PROGRAM SYNTHESIS As our next step, we explore the direction of *learning programs* further, but this time from the perspective of program synthesis. Even though the output of both program synthesis and *learning programs* is the same, a program, they differ in what guarantees they provide and in which settings they are used. In particular, the most common application of program synthesis is to repeatedly synthesize programs with *provable guarantees* from a user provided specification $\mathcal{I}$, such as a *small set of input-output examples*. For example, given the input-output pairs `(01,10)` $\rightarrow$ `11` and `(00,10)` $\rightarrow$ `10`, a bitvector synthesizer might return a program that computes bitwise `or` of the input arguments `(x, y)` $\rightarrow$ `x | y`. More formally, the problem statement in program synthesis is typically phrased as a satisfiability query that searches for a program which produces the correct result when executed on all the input-output examples in the specifications $\mathcal{I}$, i.e., $\exists p \in \mathcal{L}\ \forall (x, y) \in \mathcal{I}.\ p(x) = y$. In contrast, learning from programs is usually invoked only once on a *massive datasets* and applied to tasks that produce *probabilistic solutions*, without any formal guarantees. As a result, the problem formulation is phrased as learning a program that maximizes the probability (or minimizes the loss) of the correct answer $\arg\max_{p \in \mathcal{L}} \mathbb{E}_{(x,y)\sim\mathbb{D}}\ Pr(y \mid x; p)$.

In Figure 6.1 we illustrate the formulation of the techniques presented so far and their relation to program synthesis. We can see that in Chapter 3, we have in fact already showed how program synthesis techniques can be combined with probabilistic learning over large datasets to provide some guarantees – in this case that the learned analysis soundly over-approximates the correct results. Further, for completeness, we also differentiate between program synthesis using *symbolic* and *statistical* search (e.g., [247]). While in this chapter we will focus on using symbolic search, we note that our recent work on guiding program synthesizers [7] is a general technique applicable to both (though beyond the scope of this thesis).

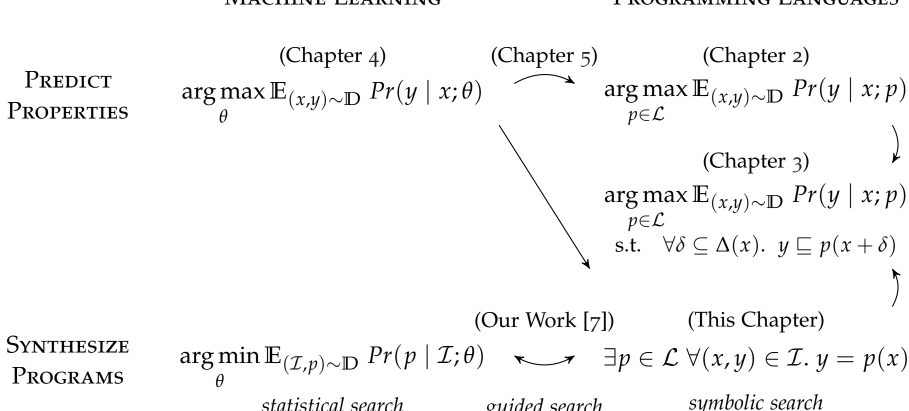


Figure 6.1: Overview of the techniques described so far and their high-level formulation.

this chapter In this chapter, we present a novel approach for synthesizing robust relational layouts from examples. Given an application design consisting of a set of views and their location on the screen, we synthesize a relational layout that when rendered, places the components at that same location. To achieve this, we first develop a symbolic synthesizer that produces relational layouts for Android that generalize across multiple screen sizes and resolutions. Then, we show how machine learning techniques can be used to improve the synthesizer by making it scale to real-world layouts, generalize better and automate parts of the design process domain experts need to perform in order to build such synthesizer.

synthesis layouts from images The design and creation of the user interface layouts are core parts of the application development for both desktop and mobile applications. Creating a user interface typically involves a collaboration between a designer, who draws an image of how the user interface should look like, and a developer that implements the design on the desired platform such as Android, iOS, Web or desktop. Concretely, the goal of the developer is to write an implementation of the user interface, referred to as layout, which when rendered on the device places all the views (e.g., buttons, text views, images, etc.) at the same location on the screen as specified by the designer, as shown in Figure 6.2. For a developer, the task of writing code to generate a given user interface layout is challenging due to the large space of potential designs – many candidate programs can produce visually identical user interface layouts yet some of these programs may fail to generalize well (e.g., when a screen is resized) or have poor performance. Designing robust layouts is a critical factor, especially in domains in which layouts are expected to be used across a large number of different contexts. For example, each Android application can be potentially installed on more than 15 000 devices with varying screen resolutions and physical dimensions, all of which need to be considered by the developer during layout implementation.

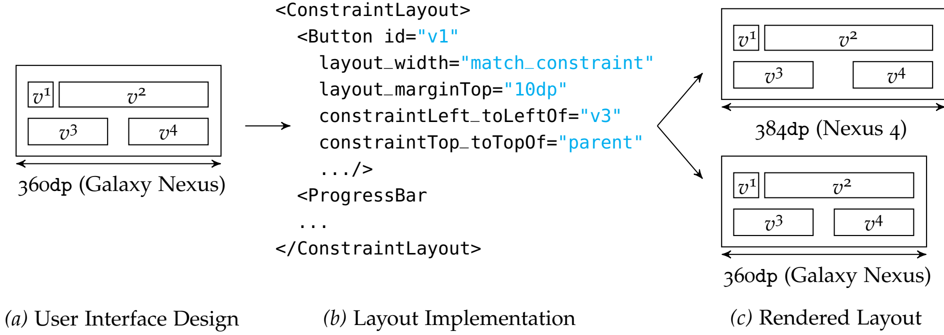


FIGURE 6.2: Main steps of user interface design: *(a)* the designer draws an image containing four views $v^1, \ldots, v^4$ (e.g., buttons, text views, etc.) how the application user interface should look like, *(b)* the developer implements the design for a given platform (e.g., Android), *(c)* the implementation is rendered on a range of devices with different physical dimensions.

COMMON LAYOUT ERRORS To illustrate common layout generalization errors, consider the layouts shown in Figure 6.3. Given the input design from Figure 6.2 the developer can create multiple layouts that all produce the expected results when rendered on a Galaxy Nexus device but for various reasons do not generalize well to devices with slightly different screen size such as Nexus 4 or P4 Pro. The leftmost example shows that keeping the absolute view position and size unchanged on a smaller device often results in drawing some of the views outside the screen. Shifting views to the left or right to adjust for the smaller screen size is also not a solution, as it can result in overlaying views on top of each other. On a larger device, the issue is reversed and not adjusting the views to the screen size leads to visual errors due to resulting blank areas on the screen. The layout that generalizes well should adjust the position of views v^2 and v^4 while also resizing v^2. Note that deciding which views to adjust and how is a hard problem that is context dependent – it depends on other components and their location on the screen. This is where the developer experience is crucial, as it allows to manually create a layout that generalizes to a large number of devices often using only a single example provided by the designer.

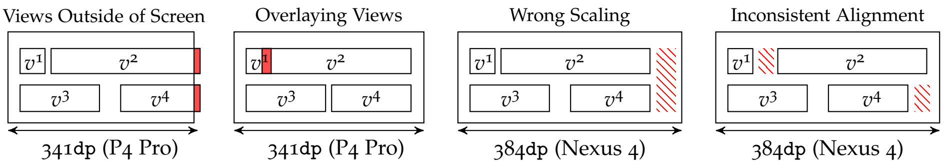


FIGURE 6.3: Common layout generalization errors, shown in red, on devices with different physical dimensions than the device for which the layout was designed.

EXISTING APPROACHES ARE INSUFFICIENT To address the gap between visual design and concrete layout implementation, several approaches have been proposed. Given the complexity and time requirements of this task, a common method is to outsource it to a company that manually creates layouts from images for a high fee [248–251]. Another approach is to try and automate parts of the process using better tool support [252, 253], generating user interface sketches from real images [254] or hand drawings [255], sketch based code search for similar layouts [256] or generation of layouts for different screen orientations [257].

Other approaches try to address the problem by generating layout code directly from images [258–261]. Their main focus is on the computer vision task required to process raw images and not on the actual layout synthesis. Concretely, *(i)* they lack a language for expressing layouts [258, 259] or the language is simple and fails to express many layouts [260], *(ii)* they either do not define a synthesis algorithm or it is implemented as part of a neural network which is non-interpretable, lacks any formal guarantees and can even produce layouts that are syntactically incorrect, *(iii)* they return any layout, without considering the issue of generalization to other contexts (i.e., multiple devices), and *(iv)* they do not provide a way for a developer to give feedback in cases where the layout should be adjusted. The ability to incorporate user feedback is especially important, since the synthesizer is expected to make mistakes as the input specification is severely underspecified – it contains only a single example, yet the produced layout is expected to generalize to a large number of different devices and screen sizes.

OUR WORK In this work, we propose a system called InferUI that addresses the above limitations in a principled way. The main idea is to phrase the problem of generating layouts as a program synthesis from examples. The examples considered in our work consist of a set of absolute view positions, allowing for a natural way to express the desired design. Then, given a set of views and their absolute positions, InferUI synthesizes a *relational layout* that renders each view according to the absolute view positions. Crucially, we also consider the harder task of synthesizing a relational layout that generalizes well across *multiple* devices from an input specification consisting of only a *single* device. As there can be a large space of possible solutions, we introduce three additional mechanisms in order to guide the synthesizer towards the desired goal: we present a set of robustness properties that a layout should satisfy (these also prevent common layout generalization errors), we introduce a probabilistic model of constraints learned from existing layouts written by developers (thus, giving preference to more natural layouts), and we additionally improve the generalization by learning a probabilistic model of how the rendered real-world applications look like (i.e., learning a probabilistic model how program outputs look like). Conditioning on the program outputs is useful not only as an additional source of learning signal but also because it allows us to obtain significantly larger datasets than what would have been possible otherwise – for example, for Android it is sufficient to browse applications in the marketplace without the need of obtaining their source code.

MAIN CONTRIBUTIONS Our main contributions are:

- A formal specification of a set of relational constraints from the most expressive and efficient Android `ConstrainLayout` (version 1.0.2).
- A new algorithm for synthesizing relational layouts on a *single* device. It succeeds in 100% of cases, bridging the gap between design and layout implementation for one device.
- A new algorithm that synthesizes relational layouts and generalizes well to *multiple* devices. Even when the given specification consists of a *single* device, the layouts correctly generalize in 92% of the cases for a real-world dataset of Google Play applications.
- A probabilistic model of constraints learned from a large set of layouts written by developers. The model is used to guide the layout synthesis and enables solving complex real-world layouts in less than 3 seconds. Further, it allows synthesizing constraints that developers prefer – it correctly predicts the exact constraints written by a developer in 62% of cases.
- A set of robustness properties that capture good design practices and prevent common visual errors caused by incorrect layouts. We incorporate these properties as part of the synthesis problem, ensuring they are always satisfied by the produced layouts. Further, we use the robustness properties to discover several visual errors in existing Android applications.
- A technique that reduces the problem of selecting which program generalizes well to the simpler task of deciding which output is correct. This allows us to learn a probabilistic model over program outputs that guides the synthesis towards layouts that generalize well. While the full details of this technique are beyond the scope of this thesis, we do provide a brief overview in Section 6.5.2. For a complete description, please refer to our work [7].

OUTLINE We organize this chapter as follows. In Section 6.1 we discuss the motivation behind our work and the practical benefits of the layout synthesis. In Section 6.2 we formally define the semantics of the Android's `ConstrainLayout` and show how the Android renderer engine encodes the semantics as a set of linear equations; the solution of which specifies the absolute position of each view. In Section 6.3 we introduce an algorithm that synthesizes layouts for a single device. In Section 6.4 we extend the algorithm to synthesize layouts that generalize across multiple devices, a necessary extension for making our tool practically usable. In Section 6.5 we introduce a probabilistic model used to guide the synthesis that allows scaling to real-world layouts. In Section 6.6 we provide an experimental evaluation of our approach, which also incudes a synthetic user study where the synthesizer refines the layout based on user feedback. Finally, we describe the related work in Section 6.7 and provide summary and discussion in Section 6.8.

6.1 PRACTICAL BENEFITS OF OUR APPROACH

Our work carries a number of practical benefits when it comes to writing, maintaining and ensuring correctness of relation layouts for Android, including: automation of end-user design, discovering and fixing layout errors, as well as porting existing layouts to improve performance. We briefly elaborate on each of these next, and discuss at a high level how our approach achieves these goals.

AUTOMATING END-USER DESIGN Traditionally, the process of creating any type of application (web, desktop, mobile, etc.) or content (text documents, images) consists of a design phase and an implementation phase. In the design phase the user decides what the result should be, whereas the implementation puts the design into effect. In some domains, it is possible to lift the implementation to a level that is natural for the user to operate on and hides all (or majority) of the internal complexity. For example, consider the task of drawing an image using a stylus pen compared to writing the corresponding vector image directly in SVG format. Even though both approaches represent the result in an SVG format, using the stylus pen is natural, orders of magnitude faster and does not require any technical knowledge about how SVG is implemented.

Unfortunately, even though there are currently almost 3 million Android applications, implementing Android layouts still resembles drawing images by writing SVG format by hand. In particular, Android layouts are represented using XML format where the user needs to know the semantics of several layouts containers (e.g., `RelativeLayout`, `LinearLayout`, `FrameLayout`, etc.), all of their attributes, and how they affect rendering the views on screen. Instead, in our work, we hide the implementation complexity of Android layouts from the user and synthesize layouts from a specification that is natural for the user to write – by giving examples of how the views should be positioned on screen.

AVOIDING AND FINDING LAYOUT ERRORS A key challenge in layout synthesis and a potential cause of errors is failing to produce layouts that generalize well to a large set of different devices and screen sizes. To address this challenge we developed and formalized a set of robustness properties that good layouts should satisfy. Our synthesis algorithm ensures that all of these properties hold for the generated layouts. Further, these robustness properties are useful beyond synthesis to find errors in existing layouts, as demonstrated in our evaluation.

PORTING LAYOUTS FOR BETTER PERFORMANCE A major reason why relational layouts were introduced in Android is their rendering performance. Concretely, implementing a given layout using `ConstraintLayout` can result in up to 20% faster rendering speed compared to previous layout implementations (e.g., `LinearLayout` or `RelativeLayout`). However, as `ConstraintLayout` was only recently released, more than 99% of existing layouts in Google Play Store applications are still written using the old layout system. To benefit from this improved

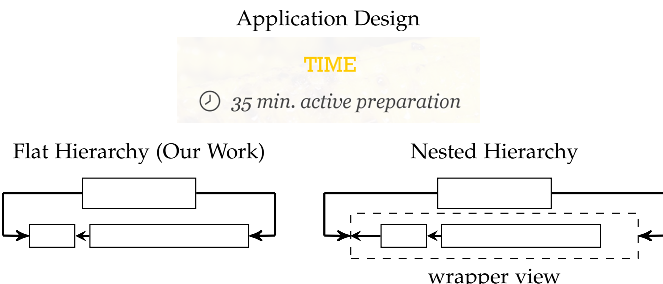


FIGURE 6.4: Same set of views expressed as flat vs nested layout hierarchy. The nested hierarchy uses extra views used to group other views without being rendered. In contrast, the flat hierarchy uses more expressive layout constraints that removed the need of wrapper views and results in faster layout rendering.

performance, developers have no other choice but to manually rewrite their existing layouts. This not only requires considerable amount of time but can introduce errors and visual artefacts. Instead, using our approach, developers can automatically synthesize `ConstraintLayout` from their existing layouts. Crucially, the layouts are synthesized not on a "best effort" basic but instead with provable guarantees that the result will visually look the same on all of the supported devices.

To illustrate the reason behind the performance gains, consider the layout shown in Figure 6.4 consisting of three views. Here, the designer would like the upper `TextView` to be centered with the `Image` and `TextView` below it. The standard way to implement this on Android is to wrap the bottom two views in a `LinearLayout` which positions them next to each other. Then the `LinearLayout` can be centered with the `TextView` above. This results in deeply-nested layout hierarchies that are slower to render. Instead, the `ConstraintLayout` is more expressive and enables centering a view, in our case the upper `TextView`, in between two other views – the left edge of bottom `Image` and right edge of the bottom `TextView`. This makes the view hierarchy smaller and faster to render by removing the unnecessary `LinearLayout`. Note that the added expressivity also means that `ConstraintLayouts` are harder to synthesize.

PROBABILISTIC MODEL OF CONSTRAINTS Finally, our probabilistic model of constraints is a useful component that can be incorporated in other applications. For example, a common issue with a number of approaches that identify user interface components in images [258, 260, 261] is that they produce noisy output. A possible approach to reduce this issue would be to incorporate the probabilistic model as an additional term in the loss function, effectively allowing the computer vision model to adjust the views into positions that produce likely layouts.

6.2 CAPTURING RELATIONAL CONSTRAINTS

In this section, we define a set of relational constraints that capture the semantics of Android's `ConstraintLayout`.

VIEWS, HANDLE POINTS AND RELATIONS We define a *view* as a tuple of five handle points corresponding to left, right, top, bottom edges and text baseline (the vertical position of an imaginary line upon which a line of text is placed), illustrated in Figure 6.5. Note that the text baseline is defined only for views containing text. For a given view A, we denote these handle points as $A.x_L$, $A.x_R$, $A.y_T$, $A.y_B$ and $A.y_{baseline}$. The handle points allow us to specify relational constraints such as: left edge of view A should be aligned to right edge of view B. That is, we can relate views via their handle points.

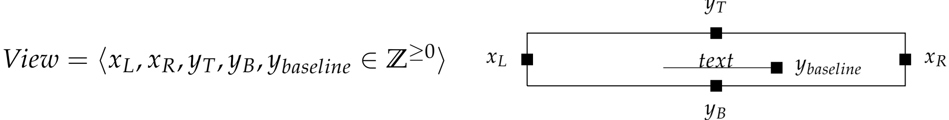


FIGURE 6.5: View definition consisting of five handle points used to relate views to each other *(left)* and illustration of the handle points on a rendered view *(right)*.

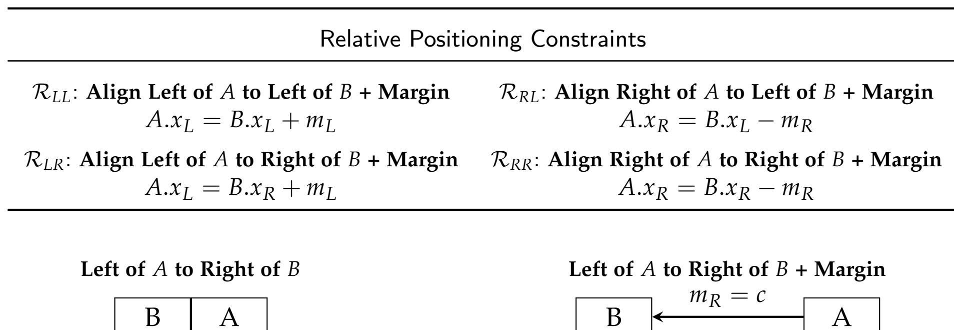
RELATIONAL CONSTRAINTS To relate views via their handle points, the following three classes of constraints exist – constraints for relative, centering and circular positioning. We define a *constraint* to specify the horizontal position of a view as the following tuple:

$$Constraint = \langle t_h \in \mathcal{C},\ A, B, C \in View, m_L, m_R \in \mathbb{Z}^{\geq 0}, b_h \in \mathbb{R}^{[0,1]}, \alpha \in \mathbb{Z}^{[0,360)}, r \in \mathbb{R} \rangle$$

where t_h is the type of the constraint, A, B and C are views related to each other and m_L, m_R, b_h, α and r are constants corresponding to the left margin, right margin, bias, angle in degrees and distance, respectively. Note that the constraints typically use only a subset of the constants and views, depending on the constraint type (which also specifies the given constant semantics). The constraints that specify vertical position are defined analogously, except that they use vertical margins and bias (m_T, m_B, b_v) instead of horizontal margin and bias (m_L, m_R, b_h). In what follows, we formally define 26 different types of constraints $\mathcal{C}$ that can be applied to a given view to specify its location, as supported by `ConstraintLayout` v1.0.2.

RELATIVE POSITIONING CONSTRAINTS A core component in relational layout is constraining the position of a given view relative to another. This can be done either horizontally (e.g., view A is to the right of B) or vertically (e.g., view A is above B). We provide semantics of horizontal relational layout constraints in Figure 6.6. For example, the constraint $\mathcal{R}_{LR}$ specifies that the left edge of view

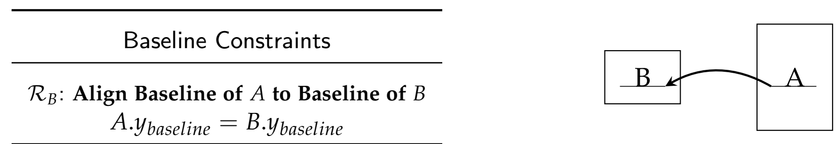
A should be aligned to the right edge of view B, which results in the constraint $A.x_L = B.x_R + m_L$. When $m_L = 0$, then the views would be rendered right next to each other as shown in Figure 6.6 *(bottom left)*. If $m_L > 0$, then view A will be positioned with the offset m_L from view B. Vertical relative constraints are defined analogously to horizontal constraints.



Relative Positioning Constraints	
$\mathcal{R}_{LL}$: **Align Left of** A **to Left of** B **+ Margin** $A.x_L = B.x_L + m_L$	$\mathcal{R}_{RL}$: **Align Right of** A **to Left of** B **+ Margin** $A.x_R = B.x_L - m_R$
$\mathcal{R}_{LR}$: **Align Left of** A **to Right of** B **+ Margin** $A.x_L = B.x_R + m_L$	$\mathcal{R}_{RR}$: **Align Right of** A **to Right of** B **+ Margin** $A.x_R = B.x_R - m_R$

FIGURE 6.6: Relative positioning constraints that specify the horizontal position (left and right edges) of view A relative to other views (B and/or C). Vertical constraints for top and bottom edges (not shown) are defined analogously.

BASELINE CONSTRAINTS Baseline constraints $\mathcal{R}_B$ provide additional flexibility to constraint the vertical position of two views by aligning their baseline handle points. This is illustrated in Figure 6.7, where we can see that regardless of the view size, their text baseline will be positioned at the same vertical position.



Baseline Constraints
$\mathcal{R}_B$: **Align Baseline of** A **to Baseline of** B $A.y_{baseline} = B.y_{baseline}$

FIGURE 6.7: Baseline position constraints that align two views vertically such that their text is at the same vertical position.

FIXED VIEW SIZE CENTERING CONSTRAINTS In addition to relating pairs of views using relative position constraints, it is also possible to relate view triples. A typical example is horizontally centering a view on the screen, as illustrated in Figure 6.8 *(bottom left)*. Here, we would like to express that view A should be horizontally centered in between views B and C instead of relating it only to the left (or right) view using a margin. For this purpose, we define fixed view size centering constraints, as shown in the Figure 6.8 *(top)*. Without *margins* and with the default *bias* $b = 0.5$, view A is always centered in between the corresponding handle points of views B and C (note that B and C can refer to the same view).

The *margins* have the same semantics as in relative positioning constraints, only that now there are both left and right margins (or top and bottom).

The *bias* attribute $b \in \mathbb{R}^{[0,1]}$ controls the preference for positioning in between views B and C. The intuition is that the bias specifies where to position the view on the line segment between both handle points. For instance, for $b = 0.35$, the view will be positioned at 35% of the line segment length, as illustrated in Figure 6.8 *(bottom right)*. If the bias is set to the minimum (i.e., $b = 0$) the view A is positioned directly to the right of B. Note that even if the bias is set to minimum (or maximum) the margins still apply. That is, the centering constraint with $b = 0$ and $m_L = 10$ will position view A at distance 10 from the right edge of view B.

Finally, in order to accurately model the semantics of Android's layout system implementation we strengthened the constraints. In particular, for constraints $\mathcal{F}_{LL}$ and $\mathcal{F}_{RR}$ the margins are ignored if view A is centered with a single handle point (i.e., if $B = C$). Further, for constraints $\mathcal{F}_{LR}$ and $\mathcal{F}_{RL}$ we discovered a bug in the Android layout solver that results in rendering view A incorrectly (its position is shifted by one pixel). This happens in case view A is related to the content frame (a view representing the available device screen size) and the margins are larger than its distance from the content frame.

Fixed View Size Centering Constraints

$\mathcal{F}_{LR}$: **Center A between Left of B and Right of C + Margin + Bias**

$$(1-b) \cdot A.x_L + b \cdot A.x_R = (1-b) \cdot (B.x_L + m_L) + b \cdot (C.x_R - m_R) \wedge$$
$$(A = B \wedge A = \text{ContentFrame}) \Rightarrow (m_L \leq A.x_L - B.x_L \wedge m_R \leq A.x_R - B.x_R)$$

$\mathcal{F}_{RL}$: **Center A between Right of B and Left of C + Margin + Bias**

$$(1-b) \cdot A.x_L + b \cdot A.x_R = (1-b) \cdot (B.x_R + m_L) + b \cdot (C.x_L - m_R) \wedge$$
$$(A = B \wedge A = \text{ContentFrame}) \Rightarrow (m_L \leq A.x_L - B.x_L \wedge m_R \leq A.x_R - B.x_R)$$

$\mathcal{F}_{LL}$: **Center A between Left of B and Left of C + Margin + Bias**

$$(1-b) \cdot A.x_L + b \cdot A.x_R = (1-b) \cdot (B.x_L + m_L) + b \cdot (C.x_L - m_R) \quad \text{if } B \neq C$$
$$(1-b) \cdot A.x_L + b \cdot A.x_R = (1-b) \cdot B.x_L + b \cdot C.x_L \wedge m_L = 0 \wedge m_R = 0 \quad \text{if } B = C$$

$\mathcal{F}_{RR}$: **Center A between Right of B and Right of C + Margin + Bias**

$$(1-b) \cdot A.x_L + b \cdot A.x_R = (1-b) \cdot (B.x_R + m_L) + b \cdot (C.x_R - m_R) \quad \text{if } B \neq C$$
$$(1-b) \cdot A.x_L + b \cdot A.x_R = (1-b) \cdot B.x_R + b \cdot C.x_R \wedge m_L = 0 \wedge m_R = 0 \quad \text{if } B = C$$

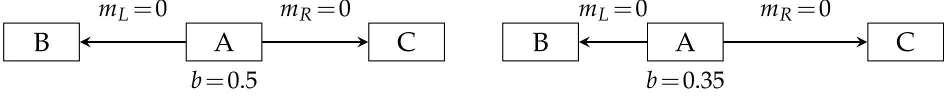


FIGURE 6.8: Fixed view size centering constraints that specify the horizontal center of view A to be along the line connecting the corresponding handle points of view B and C. Note that the constraints are *fixed view size* because they specify only the center of view A, not its width or height. Vertical constraints (not shown) are defined analogously.

CIRCULAR CONSTRAINTS Circular position constraints, shown in Figure 6.9, allow relating the center of two views at an angle α and a distance r. Using circular constraints, we can express that two views are related at an angle $\alpha = 45°$ without having to compute the corresponding margins manually.

Circular Constraints
$\mathcal{R}_C$: **Align Center of A to Center of B at an Angle + Distance** $A.x_L + A.x_R = 2r \cdot \sin(\alpha) + (B.x_L + B.x_R)$

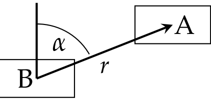


FIGURE 6.9: Circular constraints that position the center of view A at an angle α with distance r from the center of view B.

DYNAMIC VIEW SIZE CENTERING CONSTRAINTS So far all the constraints assumed that the width and height of the view are known. This is because in order to render the view on the screen we need to compute the position of all of its edges and not only one of them or its center. To support dynamic view sizes, the constraint has to relate both horizontal (or vertical) handle points, as defined in Figure 6.10. For example, the effect of constraint $\mathcal{D}_{RL}$: $A.x_L = B.x_R + m_L \wedge A.x_R = C.x_L + m_R$ with $m_L = 10, m_R = 10$ is that view A is rendered between the right and left edges of views B and C, respectively, while spanning the whole space in between them (except for margins), as illustrated in Figure 6.10 *(bottom)*.

Dynamic View Size Centering Constraints
$\mathcal{D}_{LL}$: **Center A between Left of B and Left of C + Margin** $A.x_L = B.x_L + m_L \wedge A.x_R = C.x_L + m_R$
$\mathcal{D}_{LR}$: **Center A between Left of B and Right of C + Margin** $A.x_L = B.x_L + m_L \wedge A.x_R = C.x_R + m_R$
$\mathcal{D}_{RL}$: **Center A between Right of B and Left of C + Margin** $A.x_L = B.x_R + m_L \wedge A.x_R = C.x_L + m_R$
$\mathcal{D}_{RR}$: **Center A between Right of B and Right of C + Margin** $A.x_L = B.x_R + m_L \wedge A.x_R = C.x_R + m_R$

Center A between Right of B and Left of C + Margin

FIGURE 6.10: Dynamic view size centering constraints that specify the position of both view edges. As a result, specifying the position of both edges also determines the view size. Vertical constraints (not shown) are defined analogously.

VIEW SIZE The view size needs to be specified in addition to relative, circular and fixed size centering constraints in order to compute the absolute positions of all view handle points. The view size is defined as follows:

$$Size = \langle t_h, t_v \in \{\texttt{Fixed}, \texttt{MatchConstraint}\}, width, height \in \mathbb{Z}^{\geq 0} \rangle$$

that is, the view size is either fixed (denoted as `Fixed`) in which case the view size is a constant or dynamically computed, as specified by the constraints (denoted as `MatchConstraint`). Note that it is allowed for a view to have one dimension fixed while the other dimension is computed dynamically.

6.2.1 *Layout Constraint Solving*

Having formalized the relational constraints and view sizes we now describe how they are used to compute absolute view positions. This will be a necessary component for synthesizing layouts that generalize to multiple devices presented in Section 6.4. Given a set of views with associated constraints and view sizes, we compute the absolute view positions in two steps:

1. we encode view constraints and view sizes as a set of linear equations where free variables correspond to view handle points (as formalized in Figures 6.6, 6.7, 6.8, 6.9 and 6.10), and

2. we find the satisfying assignment to the free variables by solving the resulting linear equations.

In our case, this assignment captures the absolute positions of each view's handle points corresponding to the left, right, top and bottom edges. However, because all the constraints are relational (i.e., they only specify the position of a view relative to other views), one may obtain many different satisfying assignments. This is undesirable as it results in non-deterministic view position. Because of this, a so called content frame (typically spanning the available screen size) is included as an additional view with fixed absolute coordinates. Then, if all views are transitively related to the content frame and the relations do not contain cycles, the resulting system of linear equations has exactly one satisfying assignment. Throughout the paper, when indexing views we use ρ or index zero (e.g., v^0) to refer to the content frame and indices 1 and above when referring to other views. Further, for all of our algorithms we simplify the presentation and remove clutter by encoding only the horizontal constraints (the vertical constraints are defined analogously).

ENCODING RELATIONAL CONSTRAINTS Figure 6.11 defines how to encode relational constraints as a conjunction of linear equations. We have three kinds of equations: $\phi_{position}$, ϕ_{size} and $\phi_{constraints}$. Here, $\phi_{position}$ specifies the absolute position of the content frame. As this position is known, we simply assign the concrete values to the content view handle points. Equation ϕ_{size} restricts the view size. If the size is fixed it is enforced using $x_L^i + \boldsymbol{s}^i.width = x_R^i$, which specifies that the

$$\psi_{layout}(\rho \in View, \boldsymbol{c} \subset Constraint, \boldsymbol{s} \subset Size) = \phi_{position} \wedge \phi_{size} \wedge \phi_{constraints}$$
$$\phi_{position} \stackrel{\text{def}}{=} \left(x_L^0 = \rho.x_L\right) \wedge \left(x_R^0 = \rho.x_R\right)$$
$$\phi_{constraints} \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{c}|} [\![\boldsymbol{c}^i]\!] \qquad \phi_{size} \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{s}|} \begin{cases} x_L^i + \boldsymbol{s}^i.width = x_R^i & \text{if } \boldsymbol{s}^i.t_h^i = \texttt{Fixed} \\ x_R^i - x_L^i \geq 0 & \text{otherwise} \end{cases}$$

FIGURE 6.11: Function ψ_{layout} that specifies absolute view positions from relational layout constraints $\boldsymbol{c}$ and view sizes $\boldsymbol{s}$ (where $|\boldsymbol{c}| = |\boldsymbol{s}|$) by encoding this problem as a set of linear equations.

distance between left and right handle points is equal to the width of the view. If the size is computed dynamically we only enforce that it is non negative. Finally, $\phi_{constraints}$ encodes the actual constraints over the view handle points. We use $[\![\boldsymbol{c}^i]\!]$ to denote the evaluation of the constraint $\boldsymbol{c}^i$, which returns the logical formula associated with $\boldsymbol{c}^i$ based on its definition from Figures 6.6, 6.7, 6.8, 6.9 and 6.10.

EXAMPLE Figure 6.12 illustrates the encoding of the relational layout constraints according to Figure 6.11 on a simple example consisting of two views. The constraint $\boldsymbol{c}_h^1$ specifies that the left edge of the first view is to the right of the left edge of the content frame with margin 10. The constraint $\boldsymbol{c}_h^2$ specifies that the second view should be positioned between the right edges of the first view and the content frame. The width of the first view is fixed, while the width of the second view is computed dynamically allowing it to span all of the remaining available space on the screen. By solving the resulting formula we can compute the absolute positions of all views, as shown at the bottom of Figure 6.12.

INPUT: CONSTRAINTS & SIZE

$$\boldsymbol{c}_h^1 = \langle t_h = \mathcal{R}_{LL}, A = v^1, B = v^0, m_L = 10\rangle \qquad \boldsymbol{s}_h^1 = \langle t_h = \texttt{Fixed}, width = 260\rangle$$
$$\boldsymbol{c}_h^2 = \langle t_h = \mathcal{D}_{LR}, A = v^2, B = v^1, C = v^0, m_L = m_R = 10\rangle \qquad \boldsymbol{s}_h^2 = \langle t_h = \texttt{MatchConstraint}\rangle$$

CONSTRAINT SYSTEM

$$\psi_{layout}((0,720),(\boldsymbol{c}_h^1,\boldsymbol{c}_h^2),(\boldsymbol{s}_h^1,\boldsymbol{s}_h^2)) = \phi_{position} \wedge \phi_{size} \wedge \phi_{constraints}$$
$$\phi_{constraints} \stackrel{\text{def}}{=} x_L^1 = x_L^0 + 10 \wedge x_L^2 = x_R^1 + 10 \wedge x_R^2 = x_R^0 - 10$$
$$\phi_{position} \stackrel{\text{def}}{=} x_L^0 = 0 \wedge x_R^0 = 720 \qquad \phi_{size} \stackrel{\text{def}}{=} x_L^1 + 260 = x_R^1 \wedge x_R^2 - x_L^2 \geq 0$$

SOLUTION

A satifying assigment to $x_L^1, x_R^1, x_L^2, x_R^2$ in ψ_{layout}

$x_L^1 = 10, x_R^1 = 270$ [v^1 | v^2] $x_L^2 = 280, x_R^2 = 710$

FIGURE 6.12: An example of how the relational constraints are encoded using ψ_{layout}. The solution of the formula specifies the absolute view positions on the screen.

6.3 SINGLE DEVICE RELATIONAL LAYOUT SYNTHESIS

So far we defined how to compute absolute view positions given relational constraints (Section 6.2.1). We now discuss the inverse problem of generating these constraints for a single device given the absolute view positions and sizes (as provided by a user). In particular, for each view, we are interested in generating one constraint that controls its horizontal position and one constraint for its vertical position. We need at least one constraint (for either axis) as otherwise the view position is unconstrained and cannot be computed. Moreover, exactly one constraint is also sufficient because if multiple constraints are specified then they are either redundant or unsatisfiable.

PROBLEM STATEMENT The layout synthesis problem is defined as follows:

Input: A set $\boldsymbol{v} \subset \mathit{View}$ of N views with specified absolute positions on the screen defining where the views should be rendered. A content frame $\rho \in \mathit{View}$ defining the screen size.

Output: A set of N view sizes $\boldsymbol{s} \subset \mathit{Size}$ and N horizontal and vertical constraints $\boldsymbol{c}_h, \boldsymbol{c}_v \subset \mathit{Constraint}$ (one for each input view) where $\boldsymbol{v} \models \psi_{layout}(\rho, \boldsymbol{c}_h, \boldsymbol{c}_v, \boldsymbol{s})$.

That is, we need to define a synthesizer that for a given screen size ρ and absolute positions of all views, finds constraints and view sizes whose solution matches the input (as specified by ψ_{layout}). Such a synthesizer can then automate the manual process of writing constraints and view sizes by hand.

RELATIONAL LAYOUT SYNTHESIS We encode the problem as a logical formula ψ ranging over boolean, integer and real valued variables, as shown in Figure 6.13. A model (i.e., a satisfying assignment to all the free variables) of ψ determines which constraints and view sizes should be applied, such that we solve the problem statement above. We divide the formula into four parts $\phi_{position}$, ϕ_{valid}, $\phi_{acyclic}$ and $\phi_{constraints}$ that are described next.

Here, $\phi_{positions}$ encodes the input specification by setting the handle points based on the input views $\boldsymbol{v}$ and content frame ρ. Then, ϕ_{valid} defines the domain of constants which are allowed to be used by the (to be synthesized) relational constraints. In particular, the margins m_L, m_R and the distance r have to be non-negative, the bias b_h has to be between zero and one, and the angle is a valid degree. The formula $\phi_{acyclic}$ encodes that constraint relations are acyclic. Acyclic relations are required, as otherwise the constraints do not specify a unique solution when solving for absolute view positions (since views depend transitively on themselves). To encode acyclic relations we assign an integer variable $\boldsymbol{v}^i.d > 0$ to each view $\boldsymbol{v}^i$. The intuition behind each $\boldsymbol{v}^i.d$ is that it captures distance from the

$$\psi(\rho \in View, \boldsymbol{v} \subset View) = \phi_{position} \wedge \phi_{valid} \wedge \phi_{constraints} \wedge \phi_{acyclic}$$

$$\phi_{position} \stackrel{\text{def}}{=} \left(x_L^0 = \rho.x_L\right) \wedge \left(x_R^0 = \rho.x_R\right) \wedge \left(\bigwedge_{i=1}^{|\boldsymbol{v}|} x_L^i = v^i.x_L \wedge x_R^i = v^i.x_R\right)$$

$$\phi_{valid} \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}|} m_L^i \geq 0 \wedge m_R^i \geq 0 \wedge 0 \leq b_h^i \leq 1 \wedge 0 \leq \alpha^i < 360 \wedge r^i \geq 0$$

$$\phi_{constraints} \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}|} \left(\bigwedge_{k=0}^{|\mathcal{C}(\boldsymbol{v}^i, \boldsymbol{v}, \rho)|} \boldsymbol{g}_k^i \Rightarrow [\![\boldsymbol{c}_k^i]\!] \right) \wedge \boldsymbol{g}_0^i + \cdots + \boldsymbol{g}_{|\mathcal{C}(\boldsymbol{v}^i, \boldsymbol{v}, \rho)|}^i = 1$$

$$\phi_{acyclic} \stackrel{\text{def}}{=} (\rho.d = 0) \wedge \left(\bigwedge_{i=1}^{|\boldsymbol{v}|} \boldsymbol{v}^i.d > 0 \wedge \bigwedge_{k=0}^{|\mathcal{C}(\boldsymbol{v}^i, \boldsymbol{v}, \rho)|} \boldsymbol{g}_k^i \Rightarrow \boldsymbol{v}^i.d = \begin{cases} [\![\boldsymbol{c}_k^i.B]\!].d + 1 & \text{if } t_h^i \in \mathcal{R} \\ [\![\boldsymbol{c}_k^i.B]\!].d + [\![\boldsymbol{c}_k^i.C]\!].d + 1 & \text{otherwise} \end{cases} \right)$$

FIGURE 6.13: Synthesis algorithm that given a set of absolute view positions $\boldsymbol{v}$ and a content frame ρ computes suitable constraints and view sizes that render the views at the same absolute positions as specified by $\boldsymbol{v}$.

content frame, where the content frame has a distance of zero (i.e., $\rho.d = 0$). Then, the distance of a view is defined as 1 plus the sum of distances of the views it relates to (where $[\![\boldsymbol{c}_k^i.B]\!]$ is used to denote view B associated with constraint $\boldsymbol{c}_k^i$). Such encoding efficiently disallows cycles, since introducing a cycle will result in an unsatisfiable assignment to distance variables.

The $\phi_{constraints}$ encodes the constraints for each view. Let $\mathcal{C}(\boldsymbol{v}^i, \boldsymbol{v}, \rho)$ denote the set of all admissible constraints for a given view $\boldsymbol{v}^i$. This set is obtained by instantiating each constraint type with view $\boldsymbol{v}^i$ as source (i.e., $A = \boldsymbol{v}^i$) and all the other views including content frame as possible targets (i.e., $B, C \in \{\rho \cup \boldsymbol{v} \setminus \{v^i\}\}$). We then associate a boolean variable $\boldsymbol{g}_k^i$ with each admissible constraint $\boldsymbol{c}_k^i$ for that view and add the implication $\boldsymbol{g}_k^i \Rightarrow [\![\boldsymbol{c}_k^i]\!]$. The interpretation of $\boldsymbol{g}_k^i$ is such that if it is true, the constraint $\boldsymbol{c}_k^i$ is activated, and will be returned as part of the solution. Finally, we enforce that for each view exactly one horizontal constraint is synthesized[1], by restricting the sum of all $\boldsymbol{g}_k^i$ for a given view $\boldsymbol{v}^i$ to be equal to 1.

VIEW SIZE Note that the synthesis algorithm ψ does not depend on the view size. Instead, the view size can be determined *after* a satisfying assignment for the synthesis formula is found. Concretely, if the synthesized constraint is of type $\mathcal{D}_{LL}, \mathcal{D}_{LR}, \mathcal{D}_{RL}$ or $\mathcal{D}_{RR}$ then the view size is of type `MatchConstraint` and of type `Fixed` otherwise. This is possible because during synthesis, the view size is known, as the input $\boldsymbol{v}$ already specifies all handle points.

1 In our implementation we encode this constraint using *PbEq* function, which is an optimized implementation of pseudo-boolean relations of type $k_1 * p_1 + \cdots + k_n * p_n = k$ provided by Z3 Solver. In our case $k_1, \ldots, k_n = 1$ and $k = 1$.

6.4 GENERALIZING LAYOUTS TO MULTIPLE DEVICES

In this section, we build upon the synthesis algorithm ψ and show how to extend it to synthesize constraints that generalize across multiple devices. This functionality is extremely useful as in practice developers have to consider a large number of devices with different screen sizes and resolutions on which an application might be rendered. For example, there are more than 15 000 device models with almost 100 different screen sizes (even after an adjustment using density independent pixels) that one needs to consider when developing an Android application.

Compared to ψ, which takes only a single device ρ as input, the algorithm to support multiple devices takes a list of devices $\boldsymbol{d} \subset View$ to be considered (or alternatively, a maximum allowed resize ratio of the device ρ). However, note that the input specification still consists of absolute view positions $\boldsymbol{v}$ only for a single device ρ and not for all the devices $\boldsymbol{d}$. As a result, the synthesis problem is severely underspecified, as we do not have any specification for the additional devices. We address the under specification issue using two techniques: *(i)* by designing a set of properties that "good layouts" should satisfy (this section), and *(ii)* by learning from a large set of layouts already written by developers (Section 6.5).

Our synthesis algorithm $\psi_{\square}$ supports multiple devices and is shown in Figure 6.14. It consists of three parts. First, using ψ, we compute formulas for $\boldsymbol{c}$ and $\boldsymbol{s}$ that satisfy the input specification $\boldsymbol{v}$ on device ρ. Second, using $\boldsymbol{c}$ and $\boldsymbol{s}$ from the first step we produce a view layout $\boldsymbol{v}_d$ on each device d, that is, $\boldsymbol{v}_d \models \psi_{layout_syn}(d, \boldsymbol{c}, \boldsymbol{s})$. Finally, for each device we check that the views $\boldsymbol{v}_d$ satisfy a set of robustness properties. All three steps are encoded as a single logical formula, the solution of which specifies the desired constraints and view sizes.

$$\psi_{\square}(\rho \in View, \boldsymbol{d} \subset View, \boldsymbol{v} \subset View) = \psi(\rho, \boldsymbol{v}) \rightarrow \langle \boldsymbol{c}, \boldsymbol{s} \rangle \wedge \bigwedge_{k=1}^{|\boldsymbol{d}|} \psi_{gen}(\boldsymbol{d}_k, \boldsymbol{v}, \boldsymbol{c}, \boldsymbol{s})$$

$$\begin{aligned}\psi_{gen}(d, \boldsymbol{v}, \boldsymbol{c}, \boldsymbol{s}) \stackrel{\text{def}}{=} \; & \boldsymbol{v}_d \models \psi_{layout_syn}(d, \boldsymbol{v}, \boldsymbol{c}, \boldsymbol{s}) \wedge \\ & \phi_{inside_screen}(d, \boldsymbol{v}_d) \wedge \phi_{pixel_perfect}(\boldsymbol{v}_d) \wedge \phi_{preserve_aspect_ratio}(\boldsymbol{v}, \boldsymbol{v}_d) \wedge \\ & \phi_{preserve_order}(\boldsymbol{v}, \boldsymbol{v}_d) \wedge \phi_{preserve_centering}(\boldsymbol{v}, \boldsymbol{v}_d) \wedge \phi_{preserve_margins}(\boldsymbol{v}, \boldsymbol{v}_d)\end{aligned}$$

$$\psi_{layout_syn}(d \in View, \boldsymbol{v} \subset View, \boldsymbol{c} \subset Constraint, \boldsymbol{s} \subset Size) = \phi_{position} \wedge \phi_{size} \wedge \phi_{constraints}$$

$$\phi_{position} \stackrel{\text{def}}{=} \left(\boldsymbol{v}_d^0.x_L = d.x_L\right) \wedge \left(\boldsymbol{v}_d^0.x_R = d.x_R\right)$$

$$\phi_{size} \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}|} \left(\bigwedge_{k=0}^{|\mathcal{C}(\boldsymbol{v}^i, \boldsymbol{v}, \rho)|} \boldsymbol{g}_k^i \Rightarrow \begin{cases} \boldsymbol{v}^i.x_R - \boldsymbol{v}^i.x_L = \boldsymbol{v}_d^i.x_R - \boldsymbol{v}_d^i.x_L & \text{if } \boldsymbol{c}^i.t_h^i \notin \mathcal{D} \\ true & \text{otherwise} \end{cases} \right)$$

$$\phi_{constraints} \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}|} \left(\bigwedge_{k=0}^{|\mathcal{C}(\boldsymbol{v}^i, \boldsymbol{v}, \rho)|} \boldsymbol{g}_k^i \Rightarrow [\![\boldsymbol{c}_k^i]\!]^d \right)$$

FIGURE 6.14: Algorithm for synthesizing constraints that generalize to a set of devices $\boldsymbol{d}$.

Before formalizing the robustness properties we first briefly describe the encoding of ψ_{layout_syn}. The idea behind ψ_{layout_syn} is similar to ψ_{layout} defined in Figure 6.11 except that now our goal is to encode both the layout and the synthesis within a single logical formula. For this purpose ψ_{layout_syn} introduces a fresh set of free variables $\boldsymbol{v}_d$ denoting views rendered on a device d. Then, it reuses the boolean variables $\boldsymbol{g}_k^i$ defined in ψ to restrict the view size and apply constraints over $\boldsymbol{v}_d$ (here $[\![\boldsymbol{c}_k^i]\!]^d$ denotes evaluating the constraint $\boldsymbol{c}_k^i$ over a set of views $\boldsymbol{v}_d$).

6.4.1 *Robustness Properties*

To prevent common errors made by developers, we designed a set of general properties that good layouts should satisfy. Encoding these properties as part of the synthesis formula allows us to synthesize layouts that generalize well to multiple devices even though their input specification is not available. As an example, Figure 6.15 shows four existing applications rendered on a device they were designed for (LG Nexus 4), as well as on other devices. Using the applications on a device with smaller height leads to overlaying two views containing text and image (Amigo), rendering a button partially out of the screen (Candid), as well as visual artefacts caused by stretching common margins used between views (FBook). Similarly, using the application on a device with smaller/larger width can lead to errors as the one found in the Asobimasu application. In the remainder of this section we formalize our generalization properties designed to avoid such common layout errors and encode those in $\psi_{\square}$.

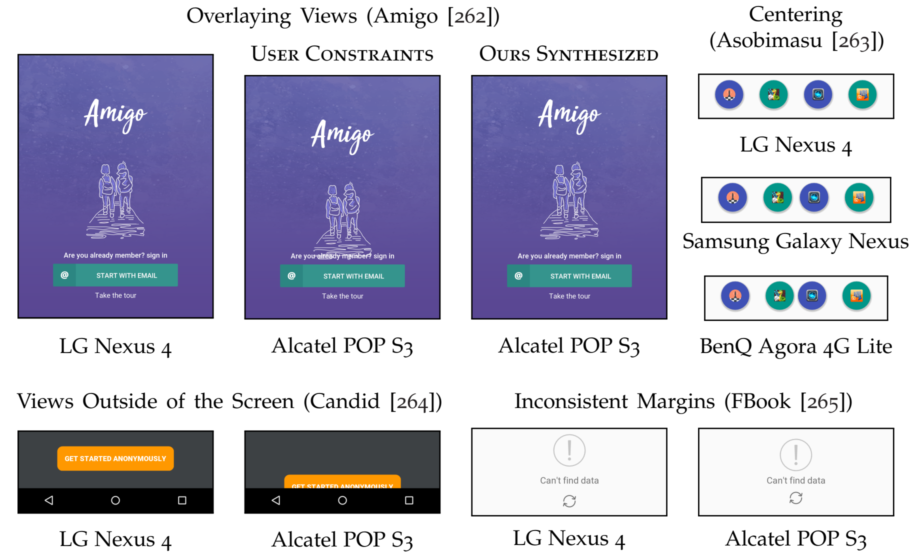


Figure 6.15: Examples of visual errors that arise from using applications on different physical devices that designed for (reference device is LG Nexus 4).

PRESERVE ORDER The $\phi_{preserve_order}$ property ensures that when views $\boldsymbol{v}$ are rendered on a different device, their relative order stays the same. That is, if view A is to the left of view B on a device ρ, we expect the same to hold for all devices. Concretely, we add constraints for each pair of view handle points that ensure their ordering is preserved. This property enforces that a layout producing the visual error in Amigo is not allowed. Instead, the synthesized constraints will prevent overlaying the views and move the application logo and name upwards instead of downwards on a smaller screen (shown in Figure 6.15).

$$\phi_{preserve_order}(\boldsymbol{v}, \boldsymbol{v}_d) \stackrel{\text{def}}{=} \bigwedge \left\{ \begin{array}{l} aligned_{LL}(\boldsymbol{v}^i, \boldsymbol{v}^j, \boldsymbol{v}_d^i, \boldsymbol{v}_d^j) \wedge aligned_{LR}(\boldsymbol{v}^i, \boldsymbol{v}^j, \boldsymbol{v}_d^i, \boldsymbol{v}_d^j) \\ aligned_{RL}(\boldsymbol{v}^i, \boldsymbol{v}^j, \boldsymbol{v}_d^i, \boldsymbol{v}_d^j) \wedge aligned_{RR}(\boldsymbol{v}^i, \boldsymbol{v}^j, \boldsymbol{v}_d^i, \boldsymbol{v}_d^j) \end{array} \middle| \begin{array}{l} \forall i, j \in \mathbb{N}. \\ 0 \le j < i < |\boldsymbol{v}| \end{array} \right\}$$

$$aligned_{LR}(\boldsymbol{v}^i, \boldsymbol{v}^j, \boldsymbol{v}_d^i, \boldsymbol{v}_d^j) \stackrel{\text{def}}{=} \begin{cases} \boldsymbol{v}_d^i.x_L = \boldsymbol{v}_d^i.x_R & \text{if } \boldsymbol{v}^i.x_L = \boldsymbol{v}^i.x_R \\ \boldsymbol{v}_d^i.x_L < \boldsymbol{v}_d^i.x_R & \text{if } \boldsymbol{v}^i.x_L < \boldsymbol{v}^i.x_R \\ \boldsymbol{v}_d^i.x_L > \boldsymbol{v}_d^i.x_R & \text{if } \boldsymbol{v}^i.x_L > \boldsymbol{v}^i.x_R \end{cases}$$

PRESERVE MARGINS A standard design technique is to position two views within a certain margin of each other. For example, views are aligned to the screen border typically with a margin of 16 pixels and the spacing between two views is often 8 pixels (or a multiple of 8). The $\phi_{preserve_margins}$ property ensures that such margins are preserved across multiple devices. The margins we preserve, denoted $\mathcal{M}$, correspond to the most commonly used values by developers.

$$\phi_{preserve_margins}(\boldsymbol{v}, \boldsymbol{v}_d) \stackrel{\text{def}}{=} \bigwedge \left\{ \begin{array}{ll} \boldsymbol{v}^i.x_L - \boldsymbol{v}^j.x_L = \boldsymbol{v}_d^i.x_L - \boldsymbol{v}_d^j.x_L & \text{if } \left|\boldsymbol{v}^i.x_L - \boldsymbol{v}^j.x_L\right| \in \mathcal{M} \\ \boldsymbol{v}^i.x_R - \boldsymbol{v}^j.x_L = \boldsymbol{v}_d^i.x_R - \boldsymbol{v}_d^j.x_L & \text{if } \left|\boldsymbol{v}^i.x_R - \boldsymbol{v}^j.x_L\right| \in \mathcal{M} \\ \boldsymbol{v}^i.x_L - \boldsymbol{v}^j.x_R = \boldsymbol{v}_d^i.x_L - \boldsymbol{v}_d^j.x_R & \text{if } \left|\boldsymbol{v}^i.x_L - \boldsymbol{v}^j.x_R\right| \in \mathcal{M} \\ \boldsymbol{v}^i.x_R - \boldsymbol{v}^j.x_R = \boldsymbol{v}_d^i.x_R - \boldsymbol{v}_d^j.x_R & \text{if } \left|\boldsymbol{v}^i.x_R - \boldsymbol{v}^j.x_R\right| \in \mathcal{M} \end{array} \middle| \begin{array}{l} \forall i, j \in \mathbb{N}. \\ j \neq i \\ 1 \le i < |\boldsymbol{v}| \\ 0 \le j < |\boldsymbol{v}| \end{array} \right\}$$

PRESERVE CENTERING The $\phi_{preserve_centering}$ property ensures that when the views $\boldsymbol{v}$ are centered on device ρ, they will also be centered on all the other devices in $\boldsymbol{d}$. Note that for this property we consider all view triples that can be centered.

$$\phi_{preserve_centering}(\boldsymbol{v}, \boldsymbol{v}_d) \stackrel{\text{def}}{=} \bigwedge \left\{ \begin{array}{ll} centered_{LL}(\boldsymbol{v}_d^i, \boldsymbol{v}_d^j, \boldsymbol{v}_d^k) & \text{if } centered_{LL}(\boldsymbol{v}^i, \boldsymbol{v}^j, \boldsymbol{v}^k) \\ centered_{LR}(\boldsymbol{v}_d^i, \boldsymbol{v}_d^j, \boldsymbol{v}_d^k) & \text{if } centered_{LR}(\boldsymbol{v}_i, \boldsymbol{v}_j, \boldsymbol{v}_k) \\ centered_{RL}(\boldsymbol{v}_d^i, \boldsymbol{v}_d^j, \boldsymbol{v}_d^k) & \text{if } centered_{RL}(\boldsymbol{v}_i, \boldsymbol{v}_j, \boldsymbol{v}_k) \\ centered_{RR}(\boldsymbol{v}_d^i, \boldsymbol{v}_d^j, \boldsymbol{v}_d^k) & \text{if } centered_{RR}(\boldsymbol{v}_i, \boldsymbol{v}_j, \boldsymbol{v}_k) \end{array} \middle| \begin{array}{l} \forall i, j, k \in \mathbb{N}. \\ j \neq i \\ k \neq i \\ 1 \le i < |\boldsymbol{v}| \\ 0 \le j, k < |\boldsymbol{v}| \end{array} \right\}$$

$$centered_{LR}(v, v^i, v^j) \stackrel{\text{def}}{=} \Big((v.x_L + v.x_R)/2 = (v^i.x_L + v^j.x_R)/2\Big)$$

PRESERVE ASPECT RATIO The $\phi_{preserve_aspect_ratio}$ property ensures that the ratio of width to height is preserved across all devices. However, this property applies only to views with one of the standard aspect ratios as defined below:

$$\phi_{preserve_aspect_ratio}(\boldsymbol{v}, \boldsymbol{v}_d) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}|} ar(\boldsymbol{v}^i) = ar(\boldsymbol{v}_d^i) \text{ if } ar(\boldsymbol{v}^i) \in \{\frac{16}{9}, \frac{3}{2}, \frac{4}{3}, \frac{1}{1}, \frac{3}{4}, \frac{2}{3}\}$$

$$ar(v) \stackrel{\text{def}}{=} \frac{v.x_R - v.x_L}{v.x_B - v.x_T}$$

PIXEL PERFECT We ensure that the synthesized constraints take into account the physical limitations of the device – the fact that device screens consist of a discrete number of pixels. To prevent rounding (which can introduce visual artefacts) we ensure that only solutions which do not require rounding are produced by restricting the handle points representation to be non-negative integers.

$$\phi_{pixel_perfect}(\boldsymbol{v}_d) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}_d|} \boldsymbol{v}_d^i.x_L \in \mathbb{Z}^{\geq 0} \wedge \boldsymbol{v}_d^i.x_R \in \mathbb{Z}^{\geq 0}$$

INSIDE SCREEN Finally, the ϕ_{inside_screen} property ensures that all views are rendered fully inside the device screen.

$$\phi_{inside_screen}(d, \boldsymbol{v}_d) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}_d|} \left(d.x_L \leq \boldsymbol{v}_d.x_L^i\right) \wedge \left(\boldsymbol{v}_d.x_R^i \leq d.x_R\right)$$

6.4.2 *Incorporating User Feedback*

As the synthesis problem is under specified, it is possible that even after satisfying all robustness properties, the synthesized layout is not the one the designer had in mind. It is therefore important that a designer can provide feedback and re-run the layout synthesis. As our synthesis algorithm is encoded as a logical formula, it naturally allows specifying a wide range of additional properties to be satisfied. However, instead of requiring the designer to write logical formulas, we can simply render the layouts and allow the user to modify the absolute position and size of the rendered views. Then, the views which were changed are added as an additional input specification.

SUMMARY In this section, we presented a synthesis algorithm that produces layouts that generalize well on multiple devices while requiring an input specification only for a single device. To achieve this, the key idea is to design a set of properties satisfied by layouts that generalize well and encode them as part of the synthesis task. Further, the properties are encoded in a modular way, as logical formulas that allow the user to easily add more constraints if necessary.

6.5 SCALING SYNTHESIS WITH A PROBABILISTIC CONSTRAINTS MODEL

In this section, we extend $\psi_{\square}$ to make it: *(i)* scale to synthesizing real-world layouts in less than a second, and *(ii)* improve the generalization across multiple devices by taking into account constraints that are likely to be written by developers.

KEY CHALLENGE I: SCALABILITY As we will show in our evaluation, state-of-the-art SMT engines do not scale to solving the synthesis formula $\psi_{\square}$ directly. This is because the formula size is cubic in the number of views $\boldsymbol{v}$ and quickly becomes intractable to solve for all but very small sizes. The main scalability bottleneck is that both synthesis algorithms ψ and $\psi_{\square}$ consider all admissible constraints for each view, out of which only two are synthesized – one horizontal and one vertical constraint. The key idea that enables us to make synthesis scale is that instead of considering all admissible constraints, we only consider those that are likely to make the synthesis formula satisfiable. That is, we are interested in learning a model $F\colon Constraint \rightarrow Constraint$ which takes as input all admissible constraints and returns only a subset of them. In the extreme case, F returns exactly the two constraints for each view that make the synthesis formula satisfiable, effectively solving the synthesis problem. Although creating such a perfect model is too expensive, we will focus on learning a model that is precise and fast to compute.

KEY CHALLENGE II: NATURAL CONSTRAINTS In Section 6.4 we defined a set of general properties which every layout should satisfy. However, for many designs, multiple options may be available and a designer has to select one of these based on their intentions. Although our model supports supplying additional constraints (Section 6.4.2), our goal is to reduce the amount of feedback the user needs to provide in order to synthesize the desired layout. For this purpose, we extend the synthesis algorithm such that if multiple constraints exist that satisfy the input specification, it produces those that are more likely to be written by a developer.

6.5.1 *Guiding the Synthesis via Probabilistic Model of Constraints*

The key idea that addresses both issues, scalability and natural layouts, is to guide the synthesis with a probabilistic model of constraints that assigns probability to each constraint $P\colon Constraint \rightarrow \mathcal{R}^{[0,1]}$. Then, F is defined to simply return K most likely constraints according to P. Our final synthesis algorithm $\psi_{\square + \square}$, shown in Figure 6.16, extends $\psi_{\square}$ by: *(i)* considering only a subset of all admissible constraints ($F(\mathcal{C}(\boldsymbol{v}^i, \boldsymbol{v}, \rho))$), and *(ii)* out of those constraints $\boldsymbol{c}$ that satisfy the formula, it selects the ones which are most likely according to the probabilistic model P.

We now define a probabilistic model that assigns probabilities to constraints. A key challenge here is that the probability of a constraint depends on the context in which it is used. That is, the same constraints can have different probability depending on where the views they relate to are located on the screen. To solve

$$\psi_{\square+\text{📊}}(\rho \in View, \boldsymbol{d} \subset View, \boldsymbol{v} \subset View) = \max_{\sum_{i=1}^{|\boldsymbol{v}|} score^i} \psi_{\square}(\rho, \boldsymbol{d}, \boldsymbol{v})$$

$$\phi_{constraints} \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}|} \left(\bigwedge_{k=0}^{|F(\mathcal{C}(\boldsymbol{v}^i,\boldsymbol{v},\rho))|} g_k^i \Rightarrow \left([\![\boldsymbol{c}_k^i]\!] \wedge score^i = P(\boldsymbol{c}_k^i, \boldsymbol{v}) \right) \right) \wedge g_0^i + \cdots + g^i_{|F(\mathcal{C}(\boldsymbol{v}^i,\boldsymbol{v},\rho))|} = 1$$

FIGURE 6.16: Synthesis algorithm $\psi_{\square+\text{📊}}$ guided by a probabilistic model of constraints.

this issue, we learn the probabilistic model over a large dataset of layouts that capture the context in which constraints are written by developers.

PROBABILISTIC MODEL DEFINITION Let $c \in Constraint$ be a relational constraint, $\boldsymbol{v} \subset View$ be a set of views and $\rho \in View$ be a content frame, as defined in Section 6.2. We define the probability of a constraint as:

$$P(c, \rho, \boldsymbol{v}) = \frac{1}{Z(\rho, \boldsymbol{v})} \prod_{k=1}^{K} P_{f_k}(c \mid f_k(c, \boldsymbol{v}))^{w_k} \tag{6.1}$$

where $\{P_{f_k}\}_{k=1}^{K}$ is a set of probability distributions with associated weights $w_k \in \mathbb{R}^k$, $\{f_k\}_{k=1}^{K}$ is a set of feature functions and Z is a normalization function that ensures P is a valid probability distribution (where $\mathcal{C}(\boldsymbol{v}, \rho)$ denotes all admissible constraints defined over $\boldsymbol{v}$ and ρ):

$$Z(\rho, \boldsymbol{v}) = \sum_{c \in \mathcal{C}(\boldsymbol{v}, \rho)} \prod_{k=1}^{K} P_{f_k}(c \mid f_k(c, \boldsymbol{v}))^{w_k} \tag{6.2}$$

The intuition behind our definition is that the complex probability distribution of a constraint can be decomposed into a product of simpler probability distributions P_{f_k} over the set of views $\boldsymbol{v}$. The goal of each P_{f_k} is to capture one relevant aspect that helps in deciding whether the constraint c is likely or not. Although P_{f_k} and its feature function f_k can be complex, we show that even a small set of well designed simple functions is sufficient to capture the intuition behind good constraints. We note that even though the feature functions can depend on the context captured by the position of the other views, they cannot condition on other constraints. This is important, as it allows us to pre-compute all constraint probabilities before solving the synthesis formula.

To estimate the distributions P_{f_k} we use a training dataset that reflects developer preferences instead of manually designing fixed distributions. For a given feature function f_k we estimate the constraint probability using maximum likelihood estimation (via counting):

$$P_{f_k}(c \mid f_k(c, \boldsymbol{v})) = \frac{1 + Count(f_k(c, \boldsymbol{v}))}{1 + |T| + \sum_{t \in T} Count(t)} \tag{6.3}$$

where $Count(f_k(c, \boldsymbol{v}))$ denotes the number of times the value computed by the function f_k was seen in the training data, T denotes the range of f_k when evaluated on the training dataset (i.e., the set of all returned values) and $\sum_{t \in T} Count(t)$ is the number of training examples. To avoid returning zero probability for values not seen during training we use an approach called additive smoothing [266, 267] which adds one to the numerator and $|T|$ to denominator.

FEATURE FUNCTIONS The feature functions used in our work are formally defined in Table 6.1. Each feature function returns a tuple of values that are used to compute the constraint probability as defined by Equation 6.3. We use *orientation* and *class* helper functions to return the constraint orientation (horizontal or vertical) and class (relational, centering or circular), respectively. We use the constraint type (or its orientation/class) as part of the return value as a shorthand for defining a specialized probability distribution learned only from constraints of that type. Further, to prefer simpler constraints we use a regularization feature function which returns the number of unique constants and views used by a constraint c.

Feature Functions: $f_k(c, \boldsymbol{v})$	Description
Margins	
$f_m \stackrel{\text{def}}{=} \langle orientation(c.t_h), c.m_L, c.m_R \rangle$	Returns the margins associated with the constraint c. Defines one model per orientation.
Bias	
$f_b \stackrel{\text{def}}{=} \langle orientation(c.t_h), c.b_h \rangle$	Returns the bias associated with the constraint c. Defines one model per orientation.
Distance	
$f_d \stackrel{\text{def}}{=} \langle class(c.t_h), \lVert LineSeg(c.t_h, c.A, c.B) \rVert \rangle$	Returns the shortest euclidean distance between A and B. Defines one model per constraint class.
Size	
$f_s \stackrel{\text{def}}{=} \langle orientation(c.t_h), s.t_h, \lfloor width/16 \rfloor \rangle$	Returns a tuple consisting of the view size type and the view width rounded to 16 pixels.
Orientation	
$f_o \stackrel{\text{def}}{=} \langle c.t_h, \texttt{arctan2}(LineSeg(c.t_h, c.A, c.B)) \rangle$	Returns the angle of the shortest line segment from A to B. Defines one model per constraint type.
Type	
$f_t \stackrel{\text{def}}{=} \langle c.t_h \rangle$	Returns the constraint type.
Intersection	
$f_i \stackrel{\text{def}}{=} \lvert \{ v \mid \forall v \in \boldsymbol{v}.\ \texttt{intersects}(v, seg) \} \rvert$ where $seg = LineSeg(c.t_h, c.A, c.B)$	Returns the number of other views intersected by a line segment between h_s and h_t.

TABLE 6.1: Feature functions used to define a probabilistic model of constraints. We use $LineSeg(c.t_h, c.A, c.B)$ to denote the shortest line segment connecting handle points of views A and B related by constraint c.

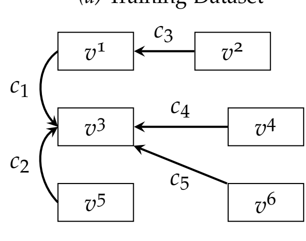
Constraint	Orientation		Intersection		Margins	
	$f_o(c, \boldsymbol{v})$	$P(c \mid f_o)$	$f_i(c, \boldsymbol{v})$	$P(c \mid f_i)$	$f_m(c, \boldsymbol{v})$	$P(c \mid f_m)$
A $\xrightarrow{c_a}$ B	$\langle \mathcal{R}_{RL}, 0° \rangle$	0.34	0	0.77	$\langle H, 10 \rangle$	0.06
A $\xleftarrow{c_b}$ B	$\langle \mathcal{R}_{LR}, 180° \rangle$	0.38	0	0.77	$\langle H, 10 \rangle$	0.06
A $\xrightarrow{c_c}$ B	$\langle \mathcal{R}_{RR}, 0° \rangle$	0.06	1	0.07	$\langle H, 60 \rangle$	0.005

TABLE 6.2: Values and probabilities computed by feature functions of different constraints relating two views.

EXAMPLE: QUERYING THE MODEL Consider two views A and B positioned next to each other and related with each other using a constraint, as illustrated in Table 6.2. For each constraint, Table 6.2 shows the values computed by the orientation, intersection and margins feature functions, as well as their probabilities as computed by our model. The most likely constraint is c_b since the orientation 180° (i.e., right to left) is slightly more likely than 0° (i.e., left to right). This is due to the fact that the screen content tends to be written from left to right which results in right to left constraints (since we want to anchor the view to the left side). Note that the computed orientation probability for constraint c_c is significantly smaller than for c_a even though they both have angle value 0°. This is because for orientation we learn a separate model for each constraint type. As a result, we learn that aligning views with their right edges, as done by constraint c_c, is much more likely when the views are either above or below each other instead of side by side as in the above example. On the other hand, for margins we learn only two models, one for horizontal and one for vertical constraints. Therefore, since both constraints c_a and c_b in Table 6.2 are horizontal and have the same margin, their probabilities according to the margin model are also the same.

EXAMPLE: TRAINING THE MODEL In Figure 6.17 we show the training of the probabilistic model of constraints. Figure 6.17 *(a)* contains example of five constraints written by a developer that are used as a training dataset. For example, the constraint c_1 specifies that the left edge of view v^1 is aligned to the left edge of view v^3 with zero margin. For each of the constraints we first extract all the features, such as the orientation, margin or the type of the constraint, as shown in *(b)*. Based on the extracted feature we then define the probabilistic models using maximum likelihood estimation which first counts the number of times each feature appears in the training dataset and then uses Equation 6.3 to compute the constraint probability as shown in Figure 6.17 *(c)*. For example, consider the probability of constraint c_3 according to the orientation model $P(c_3 \mid f_o(c_3, \boldsymbol{v})) = \frac{1+Count(f_o(c_3,\boldsymbol{v}))}{1+|T|+\sum_{t \in T} Count(t)}$. Here, $Count(f_o(c_3, \boldsymbol{v})) = Count(\langle \mathcal{R}_{LR}, 180° \rangle) = 2$ is the number of times $\langle \mathcal{R}_{LR},$ $180° \rangle$ appears in the dataset which is twice (for constraints c_3 and c_4). $|T| = |\{\langle \mathcal{R}_{LR},$

(a) Training Dataset



(b) Features Extracted from the Training Dataset in *(a)*

Constraint	$f_o(c, \boldsymbol{v})$	$f_m(c, \boldsymbol{v})$	$f_t(c, \boldsymbol{v})$
c_1	$\langle \mathcal{R}_{LL}, 270° \rangle$	$\langle H, 0 \rangle$	$\mathcal{R}_{LL}$
c_2	$\langle \mathcal{R}_{LL}, 90° \rangle$	$\langle H, 0 \rangle$	$\mathcal{R}_{LL}$
c_3	$\langle \mathcal{R}_{LR}, 180° \rangle$	$\langle H, 10 \rangle$	$\mathcal{R}_{LR}$
c_4	$\langle \mathcal{R}_{LR}, 180° \rangle$	$\langle H, 16 \rangle$	$\mathcal{R}_{LR}$
c_5	$\langle \mathcal{R}_{LR}, 160° \rangle$	$\langle H, 16 \rangle$	$\mathcal{R}_{LR}$

(c) Probabilistic Constraint Models of the Training Dataset in *(a)*

Orientation Model for $\mathcal{R}_{LL}$ Constraints

f_o	$Count(f_o)$	$P(c \mid f_o)$
$\langle \mathcal{R}_{LL}, 90° \rangle$	1	$\frac{1+1}{1+2+2} = 0.4$
$\langle \mathcal{R}_{LL}, 270° \rangle$	1	$\frac{1+1}{1+2+2} = 0.4$
Unseen	0	$\frac{1+0}{1+2+2} = 0.2$

Orientation Model for $\mathcal{R}_{LR}$ Constraints

f_o	$Count(f_o)$	$P(c \mid f_o)$
$\langle \mathcal{R}_{LR}, 180° \rangle$	2	$\frac{1+2}{1+2+3} = 0.5$
$\langle \mathcal{R}_{LR}, 160° \rangle$	1	$\frac{1+1}{1+2+3} = 0.33$
Unseen	0	$\frac{1+0}{1+2+3} = 0.16$

Margin Model for Horizontal Constraints

f_m	$Count(f_m)$	$P(c \mid f_m)$
$\langle H, 0 \rangle$	2	$\frac{1+2}{1+3+5} = 0.33$
$\langle H, 16 \rangle$	2	$\frac{1+2}{1+3+5} = 0.33$
$\langle H, 10 \rangle$	1	$\frac{1+1}{1+3+5} = 0.22$
Unseen	0	$\frac{1+0}{1+3+5} = 0.1\bar{1}$

Type Model

f_t	$Count(f_t)$	$P(c \mid f_t)$
$\mathcal{R}_{LR}$	3	$\frac{1+3}{1+2+5} = 0.5$
$\mathcal{R}_{LL}$	2	$\frac{1+2}{1+2+5} = 0.375$
Unseen	0	$\frac{1+0}{1+2+5} = 0.125$

Figure 6.17: Example of training a probabilistic model of constraints. For each constraint in a training dataset *(a)* the features are extracted *(b)* and used to train the probabilistic model *(c)*.

$180° \rangle, \langle \mathcal{R}_{LR}, 160° \rangle \}| = 2$ is the number of unique features seen during training and $\sum_{t \in T} Count(t) = 3$ is the number of constraints in the model. Note that using the constraint type as part of the feature denotes a specialized probability distribution learned only from constraints of that type. Therefore, in Figure 6.17 *(c)*, separate orientation models are defined for constraint types $\mathcal{R}_{LL}$ and $\mathcal{R}_{LR}$.

constraint generation Note that the probabilistic model can condition the constraint probability on the values of the margins, bias or views it relates to. This is crucial for the model precision yet these are the values we are interested in synthesizing. To address this issue, we take advantage of the fact that constraints with one unknown variable can be resolved *locally* given the input specification $\boldsymbol{v}$ and the content frame ρ. For this purpose we instantiate all relative and circular positioning constraints for each pair of views $A, B \in \{v \cup \rho\}$ and solve for the margins and distance, respectively. Further, we instantiate the centering constraints

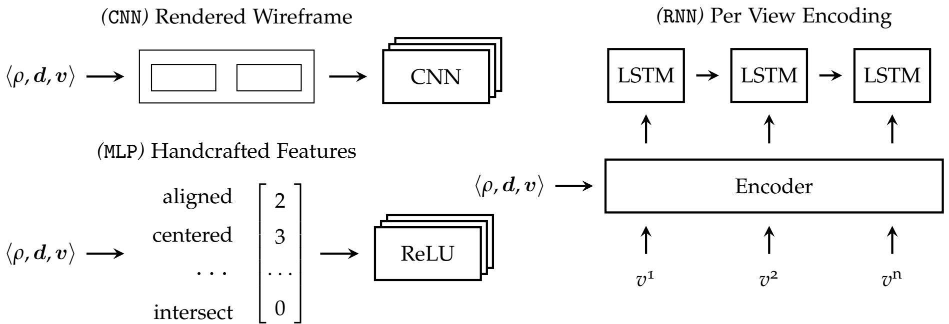


FIGURE 6.18: Illustration of three different models used to learn a neural model over the synthesis outputs. For a full description, please refer to our work [7].

for each triple of views $A, B, C \in \{v \cup \rho\}$, by fixing the bias to be from a fixed set of values and restricting that either one of the margins is zero or they are both equal to each other. Finally, the logical formulas specified by the constraints are also simplified (e.g., by evaluating $\sin(\alpha)$ in circular constraints for known α). Using this approach allows us to score the constraints with the probabilistic model *before* solving the synthesis formula.

6.5.2 *Guiding the Synthesis via Probabilistic Model of Outputs*

Finally, we briefly discuss an alternative way of building probabilistic models that guide the synthesis. In particular, instead of learning a model directly over programs, which are often difficult to obtain at scale, we can learn a probabilistic model over the program outputs. In our domain, this corresponds to learning a probabilistic model of how an application layout looks like. This allows us to effectively reduce the task of selecting which program generalizes well to a simpler task of deciding which output is correct (or more likely).

To achieve this, we learn a number of neural models over different output representations as illustrated in Figure 6.18. For *(*`CNN`*)* the representation corresponds to the rendered image of the layout where each view is drawn as a rectangle with a 1px black border on a white background. For *(*`MLP`*)* the representation corresponds to a set of handcrafted features similar to those defined in Table 6.1. Finally, in *(*`RNN`*)* the representation is obtained by computing an embedding of each view via a shared encoder. The embeddings of each view are then combined with a LSTM layer to compute the representation of the whole output. All of these probabilistic models can then be used as additional signal that guides the synthesis towards better solutions. For a full technical description of how this is implemented in practice, please refer to our work [7].

SUMMARY We have presented our approach to synthesizing relational constraints from examples. We started by introducing a new synthesis algorithm that solves

the problem for a single device in Section 6.3. In Section 6.4 we extend the synthesis algorithm such that the synthesized layouts satisfy a set of robustness properties allowing them to generalize across multiple devices. Finally, in this section we show how to scale both algorithms to complex real-world layouts using a probabilistic model of constraints to guide the synthesis towards satisfiable solutions.

6.6 EVALUATION

In this section, we provide a thorough evaluation of our proposed approach implemented in a tool called InferUI that synthesizes Android layouts. We demonstrate the key benefits of our approach by showing that:

- *Scalability.* The synthesis algorithm scales to real-world applications and synthesizes even the most complex layouts for a single device in ≈ 3 seconds.
- *Precision & Robustness.* The synthesized layouts based on a specification from only a *single* device generalize well to multiple devices with 92% of views being rendered at the screen location intended by the user.
- *Naturalness.* The synthesis algorithm creates natural constraints for developers: 62% of constraints it synthesizes match those written manually by developers.

We performed all experiments on a typical developer laptop with 2.40 GHz Intel(R) Core(TM) i7-7560U CPU, running Ubuntu 16.04.

DATASET OF ANDROID APPLICATIONS For the purposes of evaluation we collected two datasets of real-world layouts: *(i)* `PlayStore` dataset consisting of top 500 ranked applications on Google Play Store, and *(ii)* a `GitHub` dataset consisting of top 500 public repositories with the highest number of watchers on GitHub that contain `ConstraintLayout`. In order to compare to ground truth layouts written manually by developers, we consider only layouts that use `ConstraintLayout`. However, note that this does not limit the applicability of our approach since `ConstraintLayout` is the latest and most expressive layout available on Android. Further, we preprocess the dataset by removing incomplete layouts (e.g., the position of some views is not constrained), layouts with invalid constraints (e.g., relating to a non-existing view) and layouts with circular constraints. Although the Android layout solver implementation is robust enough to render even such invalid layouts, we remove them as they are typically of low quality. Finally, we consider only layouts with at least two views.

TRAINING A PROBABILISTIC MODEL OF CONSTRAINTS We trained our probabilistic model of constraints by extracting all user defined constraints from our datasets and evaluating them using the feature functions from Section 6.5. When evaluating applications from the `PlayStore` dataset we use the model trained on

the `GitHub` dataset and conversely we use the model trained on the `GitHub` dataset when evaluating layouts from the `PlayStore` dataset. Because our model estimates the constraint probability using maximum likelihood estimation (via counting) it is extremely fast to train and query – it takes less than a second to train models for both of our datasets.

EVALUATED ALGORITHMS To evaluate the effectiveness of our approach we evaluate four algorithms:

- ψ described in Section 6.3 that synthesizes layouts for a single device. At a high level, this algorithm can be summarized as solving $p \models \mathcal{I}$, where p is the synthesized layout and $\mathcal{I}$ encodes the input specification (in this case, the set of views for a single device).
- $\psi_{\boxplus}$ described in Section 6.4 that synthesizes layouts that generalize to multiple devices. This can be summarized as $p \models \mathcal{I} \wedge \phi_{robust}(\boldsymbol{d})$, where $\phi_{robust}(\boldsymbol{d})$ corresponds to robustness properties that should be satisfied across multiple devices $\boldsymbol{d}$.
- $\psi_{\boxplus + \sqcup}$ described in Section 6.5 which is an extension of $\psi_{\boxplus}$ that guides the synthesis with a learned probabilistic model of constraints. This can be summarized as $\arg max_{p \models \mathcal{I} \wedge \phi_{robust}(\boldsymbol{d})} P(p \mid \mathcal{I})$, where $P(p \mid \mathcal{I})$ is a learned probabilistic model of constraints.
- $\psi_{\sqcup}$ which is a guided extension of the ψ algorithm. This can be summarized as $\arg max_{p \models \mathcal{I}} P(p \mid \mathcal{I})$.

For all algorithms, we use the Z3 SMT solver version 4.6.0 [66] and set the timeout to one minute. Further, we guide the search for the $\psi_{\sqcup}$ and $\psi_{\boxplus + \sqcup}$ algorithms by selecting top 5 most likely constraints for each view. In case the problem is unsatisfiable we increase the number of selected constraints by 10 for each view in the unsat core. We repeat this process until a satisfiable solution is found or the time limit is reached. In our experiments, top 5 constraints are sufficient in 69% of cases. They need to be expanded once, twice and three or more times in 21%, 7% and 3% of the cases, respectively.

6.6.1 *Scalability & Runtime*

To evaluate the scalability of our algorithms, we synthesized layouts of increasing complexity. We consider layouts containing up to 20 views which includes 99.9% of the layouts in our dataset. The average runtime of successfully synthesized layouts is shown in Table 6.3. For a single device, the synthesis runtime is in milliseconds and even the most complex layouts are synthesized in a little over half a second. When synthesizing constraints for multiple devices, the runtimes are naturally higher but still very fast and less than 3 seconds. All runtimes reported are end-to-end, that is, including the time spent generating and scoring constraints using a probabilistic model.

Algorithm	Number of Views [2, 4)	[4, 8)	[8, 12)	[12, 16)	[16, 20)
Single Device					
ψ	29 ms	94 ms	490 ms	1.8 s	15 s
$\psi_{\text{ılıl}}$	37 ms	59 ms	129 ms	238 ms	519 ms
Multi Device					
$\psi_{⧉}$	49 ms	580 ms	19 s	–	–
$\psi_{⧉ + \text{ılıl}}$	44 ms	95 ms	314 ms	3 s	3 s

Table 6.3: Average runtime of different synthesis algorithms proposed in our work.

In addition to runtime, we also evaluate the percentage of successfully synthesized layouts. The results are shown in Table 6.4 together with a breakdown of why the synthesis was unsuccessful. The algorithm ψ scales to layouts of size ≈ 10 after which it timeouts in 87.2% of the cases. This is because the problem complexity growth is cubic in the number of views and quickly becomes intractable as more views are added. In contrast, the $\psi_{\text{ılıl}}$ succeeds for *all* layouts in our dataset. The problem of synthesizing layouts that generalize to multiple devices is much harder and can be solved directly by $\psi_{⧉}$ only for the smallest layouts containing less than 4 views. Guiding the synthesis using a probabilistic model significantly improves the scalability and allows us to synthesize up to three times larger layouts.

When synthesizing layouts for multiple devices, in addition to timeout, the problem can also be unsatisfiable. For example, depending on the application design it is not always possible to fit all views into a smaller screen. Instead, the designer needs to create an alternative design that removes or restructures some of the views. In some cases, however, the robustness properties are too restrictive and disallow valid layouts. For example, although preserving margins or centering is typically correct, some views might be centered simply by chance.

Algorithm	Number of Views [2, 4)	[4, 8)	[8, 12)	[12, 16)	[16, 20)	Synthesis Result `SAT`	`UNSAT`	`TIMEOUT`
Single Device								
ψ	99.0%	82.9%	38.3%	26.5%	16.6%	845	0	238
$\psi_{\text{ılıl}}$	100%	100%	100%	100%	100%	1083	0	0
Multi Device								
$\psi_{⧉}$	61.3%	23.5%	4.3%	0%	0%	327	9	747
$\psi_{⧉ + \text{ılıl}}$	98.7%	92.6%	73.0%	58.8%	41.7%	963	97	23

Table 6.4: Percentage of successfully synthesized layouts of increasing complexity.

Metric	Percentage of Views that Generalize to Multiple Devices		
	ψ	$\psi_{\text{[chart]}}$	$\psi_{\text{[devices]}+\text{[chart]}}$
GitHub Dataset			
Horizontal Match	14.7%	78.3%	**89.1%**
Vertical Match	73.7%	88.3%	**96.5%**
Full Match	12.6%	69.4%	**86.5%**
PlayStore Dataset			
Horizontal Match	13.7%	85.8%	**93.4%**
Vertical Match	81.7%	92.4%	**98.9%**
Full Match	12.9%	75.5%	**92.3%**

TABLE 6.5: Percentage of views that generalize to multiple devices. A given view generalizes if its synthesized position is the same as the one specified by the user.

6.6.2 *Precision & Robustness*

To evaluate the precision of our approach we compare the absolute view positions computed using our synthesized layout with the ground truth provided by the user (obtained by rendering layouts written by developers). Recall that the input to all of our synthesis algorithms is a set of absolute view positions on a single device. As a result, for a single device synthesis the precision is always 100%, as we are guaranteed to satisfy the input specification. To evaluate the precision on multiple devices we synthesize layouts based on the specification for devices with screen size 360`dp` $\times$ 640`dp` (e.g., Galaxy Nexus) and evaluate on devices with screen size in the range 341`dp` $\times$ 518`dp` to 384`dp` $\times$ 640`dp`. The results for both our datasets are shown in Table 6.5. We can see that ψ generalizes poorly to only 12.6% and 12.9% of all views for the `GitHub` and `PlayStore` datasets, respectively. This is expected, as the synthesis only considers a single device for which the layout is synthesized. The $\psi_{\text{[chart]}}$ improves the generalization significantly by more that 55% for both datasets. Here, even though the synthesis algorithm still considers only a single device, the probabilistic model enables the synthesis to select constraints that are likely to generalize instead of any constraints that satisfy the input specification. Finally, we can see that $\psi_{\text{[devices]}+\text{[chart]}}$ leads to additional $\approx$ 15% generalization improvement by using *both* the probabilistic model of constraints, as well as considering multiple devices during synthesis.

FINDING LAYOUT BUGS IN EXISTING APPLICATIONS The robustness properties defined in Section 6.4 can also be used to find layout bugs in existing applications. The results of evaluating robustness properties on both existing and synthesized layouts are shown in Table 6.6. We can see that although $\psi_{\text{[chart]}}$ significantly improves over ψ it still violates at least one property in more than half of

	Percentage of Layouts that Violate Robustness Properties				
Property	ψ	$\psi_{\text{[chart]}}$	$\psi_{\text{[layout]}+\text{[chart]}}$	**User Defined (GitHub + PlayStore)**	
ϕ_{inside_screen}	86%	52.3%	0%	(4%)	21
$\phi_{pixel_perfect}$	40.3%	22.7%	0%	(8.5%)	41
$\phi_{preserve_aspect_ratio}$	1.2%	0.6%	0%	(0.8%)	4
$\phi_{preserve_order}$	16.8%	24.5%	0%	(4%)	19
$\phi_{preserve_margins}$	85.0%	59.0%	0%	(5%)	24
$\phi_{preserve_centering}$	89.0%	55.2%	0%	(3.3%)	16

TABLE 6.6: Percentage of cases where layouts violates robustness properties. For user defined layouts we also provide the total number of violations found.

the analyzed layouts. On the other hand, the $\psi_{\text{[layout]}+\text{[chart]}}$ guarantees by design that all the properties are satisfied if the synthesis succeeds. Further, we have also found several property violations in existing applications. The most common violation is for $\phi_{pixel_perfect}$ property which can cause small "off by one pixel" visual artefacts. More importantly, we also discovered serious issues that result in views being rendered outside of the screen or overlapping with each other. For concrete examples of bugs we found please refer to Figure 6.15.

6.6.3 *Synthesising Natural Layouts*

We now evaluate the similarity of our synthesized constraints compared to those written manually by the users. It is a useful metric to optimize even though ideally the user never has to modify the constraints and in fact does not even need to be aware that they exist (especially if the user is not a developer but a designer). This is because, as illustrated in Table 6.5, synthesizing constraints that a user would write is an important indicator that the layout generalizes well. For our datasets, we synthesize constraints that relate the same views as the user (centering constraints are considered to match only if both target views are the same) in 62% of the cases using the $\psi_{\text{[layout]}+\text{[chart]}}$ algorithm. Note that this percentage of constraints is significantly lower than the percentage of views that generalize well. This is because multiple constraints typically exist that all result in the same absolute position of the view. The ones that are finally selected depend on the preference of the developer, such as constraints that relate views from left to right.

THE IMPORTANCE OF PROBABILISTIC MODEL OF CONSTRAINTS We have already shown that guiding the synthesis using a probabilistic model of constraints is crucial for achieving scalability. For completeness, we also evaluate the effect of returning the most likely constraints that satisfy the synthesis formula instead of returning any satisfying assignment. For $\psi_{\text{[chart]}}$, this leads to 35% improvement

in view generalization (Table 6.5) and 20% improvement in returning a constraint a user would write. For ψ, although we did not observe an improvement in view generalization (it is already very high at 92%), the improvement in returning a constraint user would write is still 10%.

6.6.4 *Incorporating User Feedback*

So far, all our experiments synthesized layouts from a single device input specification. However, as discussed in Section 6.4.2, our approach also supports refining the synthesized layout by extending the input specification with the user feedback. To evaluate such an interactive setting, we performed a synthetic user study as follows: *(i)* synthesize the layout using a single device input specification, *(ii)* render the layout on a set of devices,*(iii)* ask the user to randomly select a single view not rendered according to their design preferences. If such a view is found, we add it as part of the input specification and repeat the process from step *(i)*. Otherwise, the synthesis terminates successfully. Note that this experiment is synthetic as we emulate the user using the constraints the developers wrote in our dataset of `GitHub` and `PlayStore` applications. Overall, using ψ the user needs to provide 0, 1, 2 or 3 or more feedbacks in 63%, 25%, 8% and 4% of the cases, respectively.

6.7 RELATED WORK

We next discuss some of the work that is most closely related to ours.

LAYOUT GENERATION FROM IMAGES A number of recent works such as REMAUI [259], UI2Code [261] and pix2code [260] aim to generate layouts from images. Pix2code and UI2Code both use a language model based on deep neural networks which first encodes the input image using a convolutional neural network and then uses a recurrent neural network to generates a sequence of view names (e.g., `TextView`, `Button`, etc.) that are present on the screen. In pix2code, the output sequence is represented in a domain-specific language which encodes the simplest layout supported in Android (`LinearLayout`) and arranges all components either horizontally in a single column or vertically in a single row (or their combination). In UI2Code the output is a deeply-nested hierarchical structure with the names of all views present on the screen. This means that UI2Code does not in fact generate layouts but rather a sequence of components detected on the screen. As a result, a developer still has to write the layout manually and the output is only used to help in deciding which views are used in the screenshot. REMAUI uses optical character recognition to identify text within an application screen. Identified words, detected edges and hand-engineered heuristics are used to segment the screenshot into user interface components, which are then exported into a layout. What layouts are supported, as well as how the export is performed (the synthesis algorithm) is however not explained by the authors.

As it can be seen, pix2code, UI2Code as REMAUI are all focused on the vision problem of identifying which views are present in the input image rather than on the layout synthesis problem. In comparison, our work is the *first* to solve the complementary problem to view detection – once the views and their location are known, we synthesize a layout that *both* renders them on the screen according to this specification and generalizes to multiple devices.

VISUAL ERRORS DETECTION In recent years, several techniques have been developed to detect visual errors that arise from cross-browser incompatibilities in web applications [268–271], as well as in mobile applications [272]. For this purpose, a given application is typically first rendered on a set of devices (or web browsers) and searched for visual errors. If an error is found, an effort is made to localize its precise location (e.g., CSS property causing the error) which is then reported to the developer. In this line of work, `Cassius` [271] formalizes a set of core components of CSS 2.1 standard and then verifies that a given web page conforms to 14 accessibility guidelines. In comparison, in our work we defined a set of robustness properties that can be verified in a similar way by formalizing the Android `ConstraintLayout`. More importantly, we developed a scalable synthesis algorithm that encodes the robustness properties as part of the specification, thus avoiding visual errors by construction.

PROGRAM SYNTHESIS Combinations of program synthesis with images have been recently used to synthesize graphic programs from simple hand-drawn images [273], as well as to infer program updates from how objects are manipulated on a SVG canvas [274]. In both of these approaches, the image is abstracted to a set of traces performed to draw the image. Although the visual output of these works and ours is similar (a set of rectangles drawn on a canvas) their internal representation is very different, which also requires developing different techniques to solve the task. In our case, the representation is declarative and based on a set of relational constraints compared to representing images as programs containing conditionals or even loops. Furthermore, although our implementation currently returns a single most likely layout it might be useful to return multiple layouts amongst which the user can choose. To achieve this we could incorporate the techniques proposed by Ellis, Solar-Lezama & Tenenbaum [275] which allows efficient sampling of programs that satisfy given specification.

MACHINE LEARNING FOR PROGRAM SYNTHESIS Several approaches have been developed to accelerate program synthesizers by guiding the search towards a solution using a learned probabilistic model. Log-linear models were trained over a set of hand crafted features to guide synthesis of text processing tasks [276] and automatic patch generation [67]. In [247] a neural network is trained to predict which predefined transformations are likely to be applied such that an input-output example is satisfied. In [236], a recurrent neural network is trained on a dataset of existing proof traces and used to improve the proof search of the the-

orem prover. Neural networks are also used by Kalyan *et al.* [277] for synthesis of text processing tasks which improves over prior work by combining statistical and symbolic search approaches and by not requiring hand crafted features. Finally, the work of Lee *et al.* [278] uses probabilistic higher order grammar [6] that learns to guide A* search to speed-up the synthesis in various domains including bitvectors, circuits and text processing tasks.

Compared to prior work, our work differs in three aspects – the domain over which the probabilistic model is learned, how the model is used to guide the synthesis and the type of the model. We introduce learning techniques to the domain of relational layout synthesis. The inputs over which our features are learned are a set of views positioned on a screen, instead of strings, numbers or proof traces. Next, our work considers the search procedure that solves the synthesis task to be a black-box – in our case an SMT solver. As a result, we guide the synthesis by restricting its search space, concretely, by selecting a subset of constraints that are extended if the synthesis fails. In contrast, prior works [67, 236, 247, 276–278] keep the search space unchanged and instead modify the search procedure used to find the solution. This is because while modifying the search procedure of A^* or breadth-first search considered in prior works is straightforward, it is challenging to modify the search procedure of state-of-the-art SMT solvers. However, it would be interesting to apply our techniques presented in Chapter 5 and to learn SMT strategies that speed-up the solver on this particular dataset of formulas. Finally, the probabilistic model used in our work is maximum likelihood estimation that is extremely fast to both train and query. Here, the model precision can be further improved by using more complex models such as log-linear model, neural networks or probabilistic higher order grammar.

6.8 CONCLUSION

We presented a new approach for synthesizing relational layout constraints from examples and implemented it in a system called InferUI targeting Android applications. Our approach is based on a combination of techniques, enabling it to scale to complex real-world layouts that generalize across multiple devices. Concretely, our algorithm synthesizes layouts with provable guarantees that satisfy both the input specification for a single device as well as a set of robustness properties (which aid in generalization). We showed that InferUI works well in practice and successfully scales to synthesizing complex layouts from top 500 ranked applications in the Google Play Store as well as top 500 most watched application on `GitHub`. Crucially, we achieved this without compromising on applicability – we support the latest `ConstraintLayout` used in Android and directly generate the corresponding source code to be used by the developer.

7 CONCLUSION AND FUTURE WORK

In this dissertation, we developed a number of techniques and practical tools that learn from large codebases consisting of mainly program source code, but also program traces, program outputs, input-output examples or even natural language text from Wikipedia. The programs over which we learned correspond to both general purpose programs written in Python and JavaScript, as well as specialized programs designed for a particular domain, such as writing relational layouts in Android or writing satisfiability modulo theories (SMT) formulas.

At a high-level, our work and the models we learn can be summarized into two main dimensions – the types of goals that we are solving and the techniques we used to solve them. In terms of goals, we focused on: *(i) predicting program properties*, such as those produced by a type inference or a code completion model, and *(ii) synthesizing programs*, where the scope is to synthesize a program which when executed, produces the desired result (e.g., an Android layout or a SMT solver strategy that solves a given formula). In terms of the techniques, we focused on two different perspectives: *(i)* a machine learning perspective, $Pr(y \mid x; \theta)$, that *learns over programs*, where the program is just a different input data modality and we are learning the model's parameters θ that optimize a given metric (such as the likelihood of the correct label), and *(ii)* the programming languages perspective,

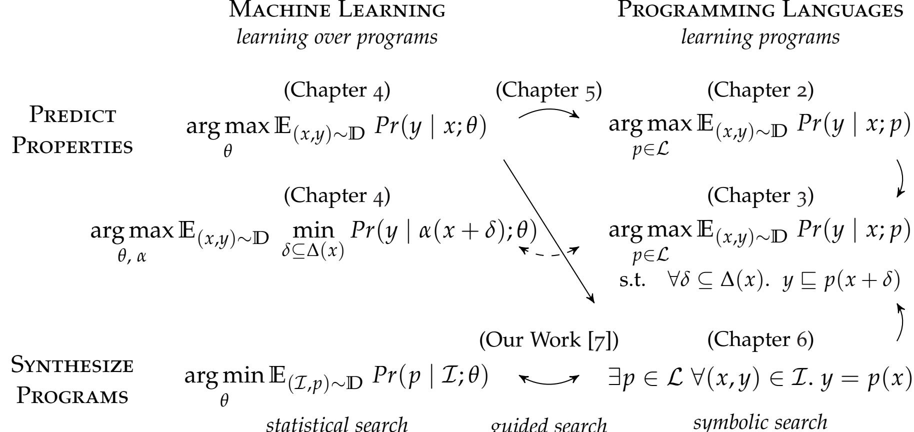


Figure 7.1: Overview two learning goals (predicting properties and synthesizing programs), two perspectives used to address them (machine learning and programming languages) and the high-level level formulation used in this dissertation to address them. The solid arrows (↗) denote that the techniques from one chapter are used or extended in another chapter. The dashed arrows (⇢) denote that the same task is solved, but using different techniques.

$Pr(y \mid x; p)$, which *learns programs* that are either used to parametrize a probabilistic model (e.g., as done in our probabilistic model called PHOG) or are executed to obtain the desired result (as in program synthesis).

We have shown how the same task can be solved independently using both the machine learning and the programming language techniques. For example, both Chapters 3 and 4 address the task of learning robust models of code that generalize beyond the samples included in the original training dataset. However, while in Chapter 3 this problem is phrased as a hard constraint that ensures the analysis soundness $\forall \delta \subseteq \Delta(x).\ \ y \sqsubseteq p(x + \delta)$, in Chapter 4 the same property is phrased as part of the optimization objective of an end-to-end differentiable deep learning model $\min_{\delta \subseteq \Delta(x)} Pr(y \mid \alpha(x + \delta); \theta)$.

At the same time, we have explored different ways of combining machine learning and program synthesis. Concretely, in Chapter 5 we first use machine learning to train a neural policy that is efficient at solving SMT formulas, and then extract the policy decisions as an interpretable program in a domain-specific language supported by the Z3 SMT solver. Here, the actions learned by the neural policy (i.e., equisatisfiable transformations of a given SMT formula) are so complex that it is infeasible to encode them formally and use program synthesis techniques directly. Instead, the neural policy is used to search in this highly complex search space and its decisions are observed by executing the trained policy on a dataset of formulas. This significantly reduces the problem program synthesis has to consider, since now it is sufficient to synthesize a loop-free program with branches that is best at combining the observed program traces.

Further, in Chapter 6 we have also explored combining machine learning and program synthesis in a domain where it is possible to encode the semantics of the target language formally and where traditional program synthesis can be applied. In this case, we use machine learning to address two key challenges of program synthesis: *(i)* scalability of the symbolic search, for which the search space grows exponentially with the program size, and *(ii)* the severe problem underspecification, in which the synthesis is expected to produce correct programs from a single input-output example. We address both of these challenges by learning a number of probabilistic models that are used to guide the search, rank the synthesized programs, as well as to generate new input-output examples automatically [7].

Overall, combining machine learning and program synthesis is an exciting and active research area, especially given the recent advances in deep learning. We believe that exploring ways of using the best of each method is important – not only it can lead to state-of-the-art models, but also because some of the techniques are fundamentally very similar, yet often overlooked by either of the research communities (e.g., as shown in Chapters 3 and 4). For example, a common deep learning view is that ultimately, deep learning will be able to solve the task given enough data and compute power. While certainly ambitious, this can also lead to systems that in practice boil down to a hyperparameter optimization exercise. More importantly, the assumptions of enough data and compute power are often misaligned with typical program synthesis tasks that are designed specifically for as few as

one input-output example. Instead, a more practical view might be that given the success of machine learning and deep learning in particular, deep learning techniques are going to be involved in most systems in some form. However, that form would vary from task to task and does not necessarily have to correspond to a model that solves the task end-to-end. Instead, the machine learning model can improve the effectiveness of individual parts or even automate their design.

7.1 FUTURE WORK

Unfortunately, we did not explore all the challenges and questions we wish to see answered. In the remainder of this section, we summarize the main directions that can be explored in the future.

PARAMETRIZING MODELS BY PROGRAMS In Chapter 2 we introduced a new probabilistic model with the key idea of parametrizing the model with a learned program. Even though our formulation allows instantiating our approach with any model in the decision tree leafs, we only experimented with the simplest possible model – a maximum likelihood estimation via counting. While this model was already powerful enough to achieve state-of-the-art results, a natural next step would be to use more powerful models such as neural networks. This can be done by incorporating them as leaf nodes, as well as for the predicates inside of the tree.

Another interesting direction in this line of work is incorporating the programs used to parametrize the model as prior over which neural based approaches can learn. For example, this idea has been recently explored by [279] where the authors: *(i)* define a domain-specific language that navigates over graphs and allows adding edges, and *(ii)* develop a new differentiable Graph Finite-State Automaton Layer that learns to optimize over the weighted combination of such programs. As another example, consider the task of program translation where several recent works phrase the problem as a token level sequence-to-sequence translation [48] or tree-to-tree translation [47]. However, even when not considering adversarial robustness, such approaches can fail in trivial cases that are slightly different than those seen during training. This is even though the corresponding program written by a domain expert to perform such translation would be the same for both cases (e.g., to rewrite ternary expressions from CoffeeScript to JavaScript).

ROBUST MODELS OF CODE While our work makes a number of steps in addressing the task of robust models of code, many challenges remain still open. For example, in our work we focused on tasks that can be solved without any ambiguity and for which it is possible to write a sound static analyzer. This results in a controlled setting where the model performance and decisions can be analyzed and reasoned about. For example, we can inspect the model manually and determine whether: *(i)* the features learned by the model (i.e., set of program locations) are indeed relevant for the prediction (i.e., they are used by the static analysis), or whether *(ii)* the features are non-robust and even though the model is making the

right prediction, it is for the wrong reasons. However, in the future we would like to also explore the effectiveness of our approach for other tasks over code, beyond tasks with no ambiguity.

Further, we optimize for the worst case adversarial robustness, which corresponds to learning a robust model for all valid programs. An interesting future work is to optimize only with respect to those modifications that are common among developers, especially if it is not possible to be robust for all of them. Additionally, while we checked the robustness for a wide range of program modifications, these are still far from exhaustive and more work is needed in defining new ones. An interesting direction to explore here is learning program modifications directly from the data, such as, by looking at the git commits.

At the same time, as the number of possible program modifications can be further increased, an interesting research direction is exploring how they can be searched and combined more efficiently. In our work, the side-effect of learning to refine the program representation and learning the conditioning context is that the search space of possible program modifications is reduced (since parts of the program become independent of each other). However, more work in this area is needed, both to explore how structural modifications can be discovered efficiently (current gradient-based approaches support only token renamings), as well as how different transformations can be composed efficiently.

Finally, in our work, we considered the task of empirical robustness, which is only an under-approximation of the true robustness and does not give formal guarantees. It is however possible to adapt and extend the recently proposed certified robustness techniques, such as those based on Interval Bound Propagation [189, 190], to verify robustness with respect to any valid word renaming and word substitution modifications.

LEARNING STRATEGIES TO SOLVE SMT FORMULAS In our work, we showed that it is possible to speed-up the state-of-the-art SMT solver `Z3` by up to 100× by learning a neural policy that predicts how the formula should be transformed next. However, we assumed a setting where the set of possible transformations (i.e., tactics), as well as the language to express them were fixed (`Z3 Strategy` language). This was possible only because the authors of `Z3` already spent a significant manual effort to refactor the heuristics used internally by the solver and expose them via a high-level interface.

A natural next step is to remove this limitation. To start with, one could learn new predicates that extend the existing `Strategy` language. This would allow us to reduce the gap between strategies that are learned by the neural policy (which can solve 27% more formulas than `Z3`) and strategies that we can synthesize and express in the `Strategy` language (which solve 8% more formulas than `Z3`). As a next step, a significantly more challenging task would be to apply our approach to other solvers, which do not yet define an explicit language to control the heuristics. Even though they do not provide such interface explicitly, they still do contain the same (or similar) heuristics internally. The main challenge here would be identify-

ing such heuristics automatically (e.g., the corresponding branches in the source code) and replacing them with the learned heuristics.

combining machine learning and program synthesis Lastly, there are various directions in which our work on combining machine learning and synthesis can be extended. Firstly, to instantiate the relational layout synthesis presented in Chapter 6, one of our contributions was to formalize the semantics of the Android `Constraint` layout. While necessary, this step is currently done manually and requires a lot of time and tedious work to correctly model all the corner cases. Even worse, to build a practical tool, one would have to continuously update the formalization to match the latest version implemented in Android. The situation becomes even more problematic when considering similar layout systems for web, where each browser comes with its own implementation. To address this issue, in the future we would like to learn the specification itself. To achieve this, the high-level problem statement is similar to the one in Chapter 3, where the goal was to learn a sound static analysis from a dataset of program traces.

The second interesting future work is combining the object detection step, which identifies user interface components and their location in the image, with the synthesis step which synthesizes relational layouts that when rendered, position the components at that location. The challenging part here is that the object detection model often produces noisy outputs, which our current synthesis does not support. At the same time, whether a given layout can be efficiently synthesized (including how likely the synthesized program is), would be useful information to refine the predictions of the object detection model.

BIBLIOGRAPHY

1. Bielik, P. & Vechev, M. *Adversarial Robustness for Code* in *Proceedings of The 37rd International Conference on Machine Learning* (2020).
2. Balunović, M., Bielik, P. & Vechev, M. *Learning to Solve SMT Formulas* in *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Curran Associates Inc., Montréal, Canada, 2018), 10338.
3. Bielik, P., Fischer, M. & Vechev, M. *Robust Relational Layout Synthesis from Examples for Android* in. **2** (Association for Computing Machinery, New York, NY, USA, 2018).
4. Bielik, P., Raychev, V. & Vechev, M. *Program Synthesis for Character Level Language Modeling* in *International Conference on Learning Representations* (2017).
5. Bielik, P., Raychev, V. & Vechev, M. T. *Learning a Static Analyzer from Data* in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I* **10426** (Springer, 2017), 233.
6. Bielik, P., Raychev, V. & Vechev, M. *PHOG: Probabilistic Model for Code* in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48* (JMLR.org, New York, NY, USA, 2016), 2933.
7. Laich, L., Bielik, P. & Vechev, M. *Guiding Program Synthesis by Learning to Generate Examples* in *International Conference on Learning Representations* (2020).
8. Raychev, V., Bielik, P. & Vechev, M. *Probabilistic Model for Code with Decision Trees* in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Association for Computing Machinery, Amsterdam, Netherlands, 2016), 731.
9. Raychev, V., Bielik, P., Vechev, M. & Krause, A. *Learning Programs from Noisy Data* in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Association for Computing Machinery, St. Petersburg, FL, USA, 2016), 761.
10. Yao, C., Bielik, P., Tsankov, P. & Vechev, M. T. Automated Discovery of Adaptive Attacks on Adversarial Defenses. *CoRR* **abs/2102.11860** (2021).
11. Mirman, M., Hägele, A., Gehr, T., Bielik, P. & Vechev, M. *Robustness Certification with Generative Models* in *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Association for Computing Machinery, New York, NY, USA, 2021).
12. Dang-Nhu, R., Singh, G., Bielik, P. & Vechev, M. *Adversarial Attacks on Probabilistic Autoregressive Forecasting Models* in *Proceedings of The 37rd International Conference on Machine Learning* (2020).
13. Schlattner, P., Bielik, P. & Vechev, M. *Learning to Infer User Interface Attributes from Images* 2019.

14. El-Hassany, A., Miserez, J., Bielik, P., Vanbever, L. & Vechev, M. *SDNRacer: Concurrency Analysis for Software-Defined Networks* in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Association for Computing Machinery, Santa Barbara, CA, USA, 2016), 402.
15. Bielik, P., Raychev, V. & Vechev, M. *Scalable Race Detection for Android Applications* in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Association for Computing Machinery, 2015), 332.
16. Miserez, J., Bielik, P., El-Hassany, A., Vanbever, L. & Vechev, M. *SDNRacer: Detecting Concurrency Violations in Software-Defined Networks* in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (Association for Computing Machinery, Santa Clara, California, 2015).
17. Bielik, P., Raychev, V. & Vechev, M. *Programming with" big code": Lessons, techniques and applications* in (2015), 41.
18. Kanade, A., Maniatis, P., Balakrishnan, G. & Shi, K. *Learning and Evaluating Contextual Embedding of Source Code* in (Vienna, Austria, 2020).
19. Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C. B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S. & Liu, S. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* **abs/2102.04664** (2021).
20. Koh, P. W., Sagawa, S., Marklund, H., Xie, S. M., Zhang, M., Balsubramani, A., Hu, W., Yasunaga, M., Phillips, R. L., Gao, I., Lee, T., David, E., Stavness, I., Guo, W., Earnshaw, B. A., Haque, I. S., Beery, S., Leskovec, J., Kundaje, A., Pierson, E., Levine, S., Finn, C. & Liang, P. WILDS: A Benchmark of in-the-Wild Distribution Shifts. *arxiv* (2021).
21. *GitHub* `https://github.blog/2018-11-08-100m-repos/`. 2018.
22. Raychev, V., Vechev, M. & Krause, A. Predicting Program Properties from "Big Code". *SIGPLAN Not.* **50**, 111 (2015).
23. Allamanis, M., Barr, E. T., Devanbu, P. & Sutton, C. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* **51**, 1 (2018).
24. Bader, J., Scott, A., Pradel, M. & Chandra, S. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.* **3**, 1 (2019).
25. Pradel, M., Gousios, G., Liu, J. & Chandra, S. *TypeWriter: Neural Type Prediction with Search-based Validation* in *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Association for Computing Machinery, New York, NY, USA, 2020).
26. Brockschmidt, M., Allamanis, M., Gaunt, A. L. & Polozov, O. Generative Code Modeling with Graphs (2018).
27. Li, J., Wang, Y., Lyu, M. R. & King, I. *Code completion with neural attention and pointer networks* in *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (AAAI Press, Stockholm, Sweden, 2018), 4159.
28. Alon, U., Brody, S., Levy, O. & Yahav, E. code2seq: Generating Sequences from Structured Representations of Code (2018).

29. Kim, S., Zhao, J., Tian, Y. & Chandra, S. Code Prediction by Feeding Trees to Transformers (2020).

30. Hellendoorn, V. J., Bird, C., Barr, E. T. & Allamanis, M. *Deep learning type inference* in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Association for Computing Machinery, Lake Buena Vista, FL, USA, 2018), 152.

31. Malik, R. S., Patra, J. & Pradel, M. *NL2Type: Inferring JavaScript Function Types from Natural Language Information* in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), 304.

32. Allamanis, M., Barr, E. T., Ducousso, S. & Gao, Z. Typilus: Neural Type Hints (2020).

33. Wei, J., Goyal, M., Durrett, G. & Dillig, I. *LambdaNet: Probabilistic Type Inference using Graph Neural Networks* in *International Conference on Learning Representations* (2019).

34. Alon, U., Zilberstein, M., Levy, O. & Yahav, E. *code2vec: learning distributed representations of code* 2019.

35. Allamanis, M., Peng, H. & Sutton, C. *A Convolutional Attention Network for Extreme Summarization of Source Code* in *International Conference on Machine Learning* (PMLR, 2016), 2091.

36. Fernandes, P., Allamanis, M. & Brockschmidt, M. Structured Neural Summarization (2018).

37. Iyer, S., Konstas, I., Cheung, A. & Zettlemoyer, L. *Summarizing Source Code using a Neural Attention Model* in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Association for Computational Linguistics, Berlin, Germany, 2016), 2073.

38. Mou, L., Li, G., Zhang, L., Wang, T. & Jin, Z. Convolutional Neural Networks over Tree Structures for Programming Language Processing (2014).

39. Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K. & Liu, X. *A Novel Neural Source Code Representation Based on Abstract Syntax Tree* in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), 783.

40. Dinella, E., Dai, H., Li, Z., Naik, M., Song, L. & Wang, K. *HOPPITY: LEARNING GRAPH TRANSFORMATIONS TO DETECT AND FIX BUGS IN PROGRAMS* 2019.

41. Pradel, M. & Sen, K. *DeepBugs: a learning approach to name-based bug detection* 2018.

42. Li, Y., Wang, S., Nguyen, T. N. & Van Nguyen, S. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang.* **3**, 1 (2019).

43. Allamanis, M., Brockschmidt, M. & Khademi, M. Learning to Represent Programs with Graphs (2017).

44. Ryan, G., Wong, J., Yao, J., Gu, R. & Jana, S. CLN2INV: Learning Loop Invariants with Continuous Logic Networks (2019).

45. Si, X., Dai, H., Raghothaman, M., Naik, M. & Song, L. Learning loop invariants for program verification. *Adv. Neural Inf. Process. Syst.* **31**, 7751 (2018).

46. Sharif, M., Lucas, K., Bauer, L., Reiter, M. K. & Shintre, S. Optimization-Guided Binary Diversification to Mislead Neural Networks for Malware Detection (2019).

47. Chen, X., Liu, C. & Song, D. Tree-to-tree neural networks for program translation. *Adv. Neural Inf. Process. Syst.* **31**, 2547 (2018).
48. Lachaux, M.-A., Roziere, B., Chanussot, L. & Lample, G. Unsupervised Translation of Programming Languages (2020).
49. David, Y., Alon, U. & Yahav, E. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proc. ACM Program. Lang.* **4**, 1 (2020).
50. Cambronero, J., Li, H., Kim, S., Sen, K. & Chandra, S. *When deep learning met code search* in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Association for Computing Machinery, Tallinn, Estonia, 2019), 964.
51. Gu, X., Zhang, H. & Kim, S. *Deep Code Search* in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), 933.
52. Sachdev, S., Li, H., Luan, S., Kim, S., Sen, K. & Chandra, S. *Retrieval on source code: a neural code search* in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Association for Computing Machinery, Philadelphia, PA, USA, 2018), 31.
53. Goodfellow, I. J., Shlens, J. & Szegedy, C. *Explaining and Harnessing Adversarial Examples* in *3rd International Conference on Learning Representations* (2015).
54. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. & Fergus, R. *Intriguing properties of neural networks* in *International Conference on Learning Representations* (2014).
55. Papernot, N., McDaniel, P., Swami, A. & Harang, R. *Crafting adversarial input sequences for recurrent neural networks* in *MILCOM 2016 - 2016 IEEE Military Communications Conference* (2016), 49.
56. Adadi, A. & Berrada, M. Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI). *IEEE Access* **6**, 52138 (2018).
57. Vilone, G. & Longo, L. Explainable Artificial Intelligence: a Systematic Review (2020).
58. Belle, V. & Papantonis, I. Principles and Practice of Explainable Machine Learning (2020).
59. Liu, C., Arnon, T., Lazarus, C., Barrett, C. & Kochenderfer, M. J. Algorithms for Verifying Deep Neural Networks (2019).
60. Iyer, A., Jonnalagedda, M., Parthasarathy, S., Radhakrishna, A. & Rajamani, S. K. *Synthesis and machine learning for heterogeneous extraction* in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Association for Computing Machinery, Phoenix, AZ, USA, 2019), 301.
61. Hindle, A., Barr, E. T., Su, Z., Gabel, M. & Devanbu, P. *On the naturalness of software* in *Proceedings of the 34th International Conference on Software Engineering* (IEEE Press, Zurich, Switzerland, 2012), 837.
62. Nguyen, A. T. & Nguyen, T. N. *Graph-Based Statistical Language Model for Code* in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* **1** (2015), 858.
63. Nguyen, T. T., Nguyen, A. T., Nguyen, H. A. & Nguyen, T. N. *A statistical semantic language model for source code* in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Association for Computing Machinery, Saint Petersburg, Russia, 2013), 532.

64. Raychev, V., Vechev, M. & Yahav, E. Code completion with statistical language models. *SIGPLAN Not.* **49**, 419 (2014).

65. Facebook. *Facebook Flow: Static Typechecker for JavaScript* `https://github.com/facebook/flow`. 2016.

66. De Moura, L. & Bjørner, N. *Z3: An Efficient SMT Solver* in *Tools and Algorithms for the Construction and Analysis of Systems* (Springer Berlin Heidelberg, 2008), 337.

67. Long, F. & Rinard, M. Automatic patch generation by learning correct code. *SIGPLAN Not.* **51**, 298 (2016).

68. Allamanis, M. & Sutton, C. *Mining source code repositories at massive scale using language modeling* in *2013 10th Working Conference on Mining Software Repositories (MSR)* (2013), 207.

69. Karampatsis, R.-M., Babii, H., Robbes, R., Sutton, C. & Janes, A. *Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code* in *2020 IEEE/ACM 42st International Conference on Software Engineering (ICSE)* (2020).

70. Bojanowski, P., Grave, É., Joulin, A. & Mikolov, T. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics* **5**, 135 (2017).

71. Mikolov, T., Sutskever, I., Deoras, A., Son, L. H., Kombrink, Š. & Černocký, J. *Subword Language Modeling With Neural Networks* 2012.

72. Luong, M.-T., Socher, R. & Manning, C. D. *Better Word Representations with Recursive Neural Networks for Morphology* in (2013), 104.

73. Vinyals, O., Fortunato, M. & Jaitly, N. in *Advances in Neural Information Processing Systems 28* (eds Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M. & Garnett, R.) 2692 (Curran Associates, Inc., 2015).

74. Mnih, V., Heess, N., Graves, A. & Kavukcuoglu, K. *Recurrent models of visual attention* in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2* (MIT Press, Montreal, Canada, 2014), 2204.

75. Bahdanau, D., Cho, K. & Bengio, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

76. Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhudinov, R., Zemel, R. & Bengio, Y. *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention* in (eds Bach, F. & Blei, D.) **37** (PMLR, Lille, France, 2015), 2048.

77. Luong, M.-T., Pham, H. & Manning, C. D. *Effective Approaches to Attention-based Neural Machine Translation* in (2015), 1412.

78. Mnih, V., Heess, N., Graves, A. & Kavukcuoglu, K. in *Advances in Neural Information Processing Systems 27* (eds Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D. & Weinberger, K. Q.) 2204 (Curran Associates, Inc., 2014).

79. Ribeiro, M. T., Singh, S. & Guestrin, C. *"Why Should I Trust You?": Explaining the Predictions of Any Classifier* in (2016), 97.

80. Augasta, M. G. & Kathirvalavakumar, T. Reverse Engineering the Neural Networks for Rule Extraction in Classification Problems. *Neural Process. Letters* **35**, 131 (2012).

81. Zilke, J. R., Mencia, E. L. & Janssen, F. *Deepred–rule extraction from deep neural networks* in *International Conference on Discovery Science* (2016), 457.

82. Lakkaraju, H., Kamar, E., Caruana, R. & Leskovec, J. Interpretable & Explorable Approximations of Black Box Models (2017).

83. Chen, J., Song, L., Wainwright, M. J. & Jordan, M. I. Learning to Explain: An Information-Theoretic Perspective on Model Interpretation (2018).

84. Shrikumar, A., Greenside, P. & Kundaje, A. Learning Important Features Through Propagating Activation Differences (2017).

85. Sundararajan, M., Taly, A. & Yan, Q. Axiomatic Attribution for Deep Networks (2017).

86. Ying, R., Bourgeois, D., You, J., Zitnik, M. & Leskovec, J. GNNExplainer: Generating Explanations for Graph Neural Networks. *Adv. Neural Inf. Process. Syst.* **32**, 9240 (2019).

87. Lundberg, S. M. & Lee, S.-I. in *Advances in Neural Information Processing Systems 30* (eds Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S. & Garnett, R.) 4765 (Curran Associates, Inc., 2017).

88. Erhan, D., Bengio, Y., Courville, A. & Vincent, P. Visualizing higher-layer features of a deep network. *University of Montreal* **1341**, 1 (2009).

89. Koh, P. W. & Liang, P. Understanding Black-box Predictions via Influence Functions (2017).

90. Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., *et al.* A survey of methods for explaining black box models. *ACM computing* (2018).

91. Xie, T. & Grossman, J. C. Crystal Graph Convolutional Neural Networks for an Accurate and Interpretable Prediction of Material Properties. *Phys. Rev. Lett.* **120**, 145301 (2018).

92. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P. & Bengio, Y. Graph Attention Networks (2017).

93. Neil, D., Briody, J., Lacoste, A., Sim, A., Creed, P. & Saffari, A. Interpretable Graph Convolutional Neural Networks for Inference on Noisy Knowledge Graphs (2018).

94. Odena, A., Olsson, C., Andersen, D. & Goodfellow, I. *Tensorfuzz: Debugging neural networks with coverage-guided fuzzing* in *International Conference on Machine Learning* (2019), 4901.

95. Hochreiter, S. & Schmidhuber, J. Long short-term memory. *Neural Comput.* **9**, 1735 (1997).

96. Karpathy, A. *The Unreasonable Effectiveness of Recurrent Neural Networks*

97. Madry, A., Makelov, A., Schmidt, L., Tsipras, D. & Vladu, A. Towards Deep Learning Models Resistant to Adversarial Attacks (2017).

98. Wong, E. & Kolter, Z. *Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope* in (eds Dy, J. & Krause, A.) **80** (PMLR, Stockholmsmässan, Stockholm Sweden, 2018), 5286.

99. Sinha, A., Namkoong, H. & Duchi, J. *Certifiable Distributional Robustness with Principled Adversarial Training* in *International Conference on Learning Representations* (2018).

100. Raghunathan, A., Steinhardt, J. & Liang, P. *Certified Defenses against Adversarial Examples* in *International Conference on Learning Representations* (2018).

101. Miyato, T., Dai, A. M. & Goodfellow, I. *Adversarial Training Methods for Semi-supervised Text Classification* in *International Conference on Learning Representations* (2017).

102. Belinkov, Y. & Bisk, Y. Synthetic and Natural Noise Both Break Neural Machine Translation (2017).

103. Ebrahimi, J., Rao, A., Lowd, D. & Dou, D. *HotFlip: White-Box Adversarial Examples for Text Classification* in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)* (Association for Computational Linguistics, 2018), 31.

104. Gao, J., Lanchantin, J., Soffa, M. L. & Qi, Y. *Black-Box Generation of Adversarial Text Sequences to Evade Deep Learning Classifiers* in *2018 IEEE Security and Privacy Workshops (SPW)* (2018), 50.

105. Liang, B., Li, H., Su, M., Bian, P., Li, X. & Shi, W. *Deep Text Classification Can Be Fooled* in *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (AAAI Press, Stockholm, Sweden, 2018), 4208.

106. Yefet, N., Alon, U. & Yahav, E. Adversarial Examples for Models of Code (2019).

107. Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A. & Tinelli, C. *CVC4* in *Computer Aided Verification* (Springer Berlin Heidelberg, 2011), 171.

108. Redmon, J., Divvala, S., Girshick, R. & Farhadi, A. *You only look once: Unified, real-time object detection* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), 779.

109. Huang, J., Rathod, V., Sun, C., Zhu, M., Korattikara, A., Fathi, A., Fischer, I., Wojna, Z., Song, Y., Guadarrama, S., *et al. Speed/accuracy trade-offs for modern convolutional object detectors* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017), 7310.

110. Ren, S., He, K., Girshick, R. & Sun, J. in *Advances in Neural Information Processing Systems 28* (eds Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M. & Garnett, R.) 91 (Curran Associates, Inc., 2015).

111. Chen, J., Xie, M., Xing, Z., Chen, C., Xu, X., Zhu, L. & Li, G. *Object detection for graphical user interface: old fashioned or deep learning or a combination?* in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Association for Computing Machinery, Virtual Event, USA, 2020), 1202.

112. Gvero, T. & Kuncak, V. *Synthesizing Java expressions from free-form queries* in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Association for Computing Machinery, Pittsburgh, PA, USA, 2015), 416.

113. Karaivanov, S., Raychev, V. & Vechev, M. *Phrase-Based Statistical Translation of Programming Languages* in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Association for Computing Machinery, Portland, Oregon, USA, 2014), 173.

114. Gvero, T., Kuncak, V., Kuraj, I. & Piskac, R. *Complete completion using types and weights* in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Association for Computing Machinery, Seattle, Washington, USA, 2013), 27.

115. Tai, K. S., Socher, R. & Manning, C. D. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks (2015).

116. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. U. & Polosukhin, I. in *Advances in Neural Information Processing Systems 30* (eds Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S. & Garnett, R.) 5998 (Curran Associates, Inc., 2017).

117. Mitchell, T. M. *Machine Learning* 1st ed. (McGraw-Hill, Inc., New York, NY, USA, 1997).

118. Quinlan, J. R. Induction of Decision Trees. *Mach. Learn.* **1**, 81 (1986).

119. Chen, S. F. & Goodman, J. An empirical study of smoothing techniques for language modeling. *Comput. Speech Lang.* **13**, 359 (1999).

120. Kneser, R. & Ney, H. *Improved backing-off for M-gram language modeling* in *1995 International Conference on Acoustics, Speech, and Signal Processing* **1** (1995), 181.

121. Witten, I. H. & Bell, T. C. The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression. *IEEE Trans. Inf. Theory* **37**, 1085 (1991).

122. Rosenfeld, R. Two decades of statistical language modeling: Where do we go from here? *Proc. IEEE* **88**, 1270 (2000).

123. Allamanis, M., Tarlow, D., Gordon, A. & Wei, Y. *Bimodal Modelling of Source Code and Natural Language* in *International Conference on Machine Learning* (PMLR, 2015), 2123.

124. Allamanis, M., Barr, E. T., Bird, C. & Sutton, C. *Suggesting accurate method and class names* in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Association for Computing Machinery, Bergamo, Italy, 2015), 38.

125. Liu, C., Wang, X., Shin, R., Gonzalez, J. E. & Song, D. *Neural Code Completion* 2016.

126. Karpathy, A., Johnson, J. & Fei-Fei, L. Visualizing and Understanding Recurrent Networks (2015).

127. Hutter, M. *The Human Knowledge Compression Contest* http://prize.hutter1.net/. 2012.

128. Dauphin, Y., de Vries, H. & Bengio, Y. *Equilibrated adaptive learning rates for non-convex optimization* in *Advances in Neural Information Processing Systems* (eds Cortes, C., Lawrence, N., Lee, D., Sugiyama, M. & Garnett, R.) **28** (Curran Associates, Inc., 2015), 1504.

129. Chung, J., Ahn, S. & Bengio, Y. Hierarchical Multiscale Recurrent Neural Networks (2016).

130. Kuhn, R. & De Mori, R. A cache-based natural language model for speech recognition. *IEEE Trans. Pattern Anal. Mach. Intell.* **12**, 570 (1990).

131. Graves, A. Generating Sequences With Recurrent Neural Networks (2013).

132. Sutskever, I., Martens, J. & Hinton, G. E. *Generating Text with Recurrent Neural Networks* 2011.

133. Wu, Y., Zhang, S., Zhang, Y., Bengio, Y. & Salakhutdinov, R. R. *On multiplicative integration with recurrent neural networks* in *Advances in neural information processing systems* (2016), 2856.

134. Teh, Y. W. *A hierarchical Bayesian language model based on Pitman-Yor processes* in *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics* (2006), 985.

135. Allamanis, M., Barr, E. T., Bird, C. & Sutton, C. *Learning natural coding conventions* in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Association for Computing Machinery, Hong Kong, China, 2014), 281.

136. Tu, Z., Su, Z. & Devanbu, P. *On the localness of software* in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), 269.

137. Collins, M. Head-Driven Statistical Models for Natural Language Parsing. *Comput. Linguist.* **29**, 589 (2003).

138. Allamanis, M. & Sutton, C. *Mining idioms from source code* in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Association for Computing Machinery, Hong Kong, China, 2014), 472.

139. Maddison, C. & Tarlow, D. *Structured Generative Models of Natural Source Code* in *International Conference on Machine Learning* (PMLR, 2014), 649.

140. Liang, P., Jordan, M. I. & Klein, D. *Learning programs: A hierarchical Bayesian approach* in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)* (2010), 639.

141. Ross Quinlan, J. *C4.5: Programs for Machine Learning* (Morgan Kaufmann, 1993).

142. Garg, P., Neider, D., Madhusudan, P. & Roth, D. Learning invariants using decision trees and implication counterexamples. *SIGPLAN Not.* **51**, 499 (2016).

143. Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E. & Udupa, A. *Syntax-guided synthesis* in *2013 Formal Methods in Computer-Aided Design* (2013), 1.

144. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S. & Saraswat, V. *Combinatorial sketching for finite programs* in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (Association for Computing Machinery, San Jose, California, USA, 2006), 404.

145. Solar-Lezama, A. Program sketching. *Int. J. Softw. Tools Technol. Trans.* **15**, 475 (2013).

146. Jha, S., Gulwani, S., Seshia, S. A. & Tiwari, A. *Oracle-guided component-based program synthesis* in *2010 ACM/IEEE 32nd International Conference on Software Engineering* **1** (2010), 215.

147. Hottelier, T. & Bodik, R. Synthesis of layout engines from relational constraints. *SIGPLAN Not.* **50**, 74 (2015).

148. Barman, S., Bodik, R., Chandra, S., Torlak, E., Bhattacharya, A. & Culler, D. *Toward tool support for interactive synthesis* in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (Association for Computing Machinery, Pittsburgh, PA, USA, 2015), 121.

149. Raza, M., Gulwani, S. & Milic-Frayling, N. *Compositional program synthesis from natural language and examples* in *Twenty-Fourth International Joint Conference on Artificial Intelligence* (2015).

150. Kneuss, E., Kuraj, I., Kuncak, V. & Suter, P. *Synthesis modulo recursive functions* in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications* (Association for Computing Machinery, Indianapolis, Indiana, USA, 2013), 407.

151. Mangal, R., Zhang, X., Nori, A. V. & Naik, M. *A user-guided approach to program analysis* in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Association for Computing Machinery, Bergamo, Italy, 2015), 462.

152. Luxburg, U. v. & Schölkopf, B. in *Handbook of the History of Logic* (eds Gabbay, D. M., Hartmann, S. & Woods, J.) 651 (North-Holland, 2011).

153. Jensen, S. H., Møller, A. & Thiemann, P. *Type Analysis for JavaScript* in *Static Analysis* (Springer Berlin Heidelberg, 2009), 238.

154. Cousot, P. & Cousot, R. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints* in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (Association for Computing Machinery, Los Angeles, California, 1977), 238.

155. Giacobazzi, R., Logozzo, F. & Ranzato, F. Analyzing Program Analyses. *SIGPLAN Not.* **50**, 261 (2015).

156. Smaragdakis, Y. & Balatsouras, G. Pointer Analysis. *Found. Trends Program. Lang.* **2**, 1 (2015).

157. Guarnieri, S. & Livshits, V. B. *GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code* in *USENIX Security Symposium* **10** (2009), 78.

158. Madsen, M., Livshits, B. & Fanning, M. *Practical static analysis of JavaScript applications in the presence of frameworks and libraries* in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Association for Computing Machinery, Saint Petersburg, Russia, 2013), 499.

159. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J. N., Chang, B.-Y. E., Guyer, S. Z., Khedker, U. P., Møller, A. & Vardoulakis, D. In defense of soundiness: a manifesto. *Commun. ACM* **58**, 44 (2015).

160. Jang, D. & Choe, K.-M. *Points-to analysis for JavaScript* in *Proceedings of the 2009 ACM symposium on Applied Computing* (Association for Computing Machinery, Honolulu, Hawaii, 2009), 1930.

161. Gulwani, S. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.* **46**, 317 (2011).

162. Feser, J. K., Chaudhuri, S. & Dillig, I. Synthesizing data structure transformations from input-output examples. *SIGPLAN Not.* **50**, 229 (2015).

163. Gehr, T., Dimitrov, D. & Vechev, M. *Learning Commutativity Specifications* in *Computer Aided Verification* (Springer International Publishing, 2015), 307.

164. Heule, S., Sridharan, M. & Chandra, S. *Mimic: computing models for opaque code* in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Association for Computing Machinery, Bergamo, Italy, 2015), 710.

165. Jeon, J., Qiu, X., Fetter-Degges, J., Foster, J. S. & Solar-Lezama, A. *Synthesizing Framework Models for Symbolic Execution* in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (2016), 156.

166. Katz, O., El-Yaniv, R. & Yahav, E. Estimating types in binaries using predictive modeling. *SIGPLAN Not.* **51**, 313 (2016).

167. Oh, H., Yang, H. & Yi, K. Learning a strategy for adapting a program analysis via bayesian optimisation. *SIGPLAN Not.* **50**, 572 (2015).

168. Heo, K., Oh, H. & Yang, H. *Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis* in *Static Analysis* (Springer Berlin Heidelberg, 2016), 237.

169. Cha, S., Jeong, S. & Oh, H. *Learning a Strategy for Choosing Widening Thresholds from a Large Codebase* in *Programming Languages and Systems* (Springer International Publishing, 2016), 25.

170. Si, X., Raghothaman, M., Heo, K. & Naik, M. Synthesizing Datalog Programs Using Numerical Relaxation (2019).

171. Muggleton, S. H., Lin, D. & Tamaddoni-Nezhad, A. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Mach. Learn.* **100**, 49 (2015).

172. Si, X., Lee, W., Zhang, R., Albarghouthi, A., Koutris, P. & Naik, M. *Syntax-guided synthesis of Datalog programs* in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Association for Computing Machinery, Lake Buena Vista, FL, USA, 2018), 515.

173. Albarghouthi, A., Koutris, P., Naik, M. & Smith, C. *Constraint-Based Synthesis of Datalog Programs* in *Principles and Practice of Constraint Programming* (Springer International Publishing, 2017), 689.

174. Sharma, R., Gupta, S., Hariharan, B., Aiken, A. & Nori, A. V. *Verification as Learning Geometric Concepts* in *Static Analysis* (Springer Berlin Heidelberg, 2013), 388.

175. Clarke, E., Grumberg, O., Jha, S., Lu, Y. & Veith, H. *Counterexample-Guided Abstraction Refinement* in *Computer Aided Verification* (Springer Berlin Heidelberg, 2000), 154.

176. Feldthaus, A., Schäfer, M., Sridharan, M., Dolby, J. & Tip, F. *Efficient construction of approximate call graphs for JavaScript IDE services* in *2013 35th International Conference on Software Engineering (ICSE)* (2013), 752.

177. Midtgaard, J. & Møller, A. *QuickChecking static analysis properties* 2017.

178. Bugariu, A., Wüstholz, V., Christakis, M. & Müller, P. *Automatically testing implementations of numerical abstract domains* in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Association for Computing Machinery, Montpellier, France, 2018), 768.

179. Schrouff, J., Wohlfahrt, K., Marnette, B. & Atkinson, L. Inferring Javascript types using Graph Neural Networks (2019).

180. Liu, Z., Wang, Z., Liang, P. P., Salakhutdinov, R. R., Morency, L.-P. & Ueda, M. *Deep Gamblers: Learning to Abstain with Portfolio Theory* in *Advances in Neural Information Processing Systems* (eds Wallach, H., Larochelle, H., Beygelzimer, A., Alché-Buc, F., Fox, E. & Garnett, R.) **32** (Curran Associates, Inc., 2019), 10623.

181. Laidlaw, C. & Feizi, S. Playing it Safe: Adversarial Robustness with an Abstain Option (2019).

182. Gal, Y. & Ghahramani, Z. *Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning* in *International Conference on Machine Learning* (PMLR, 2016), 1050.

183. Gal, Y. Uncertainty in deep learning. *University of Cambridge* **1** (2016).

184. Geifman, Y. & El-Yaniv, R. *Selective Classification for Deep Neural Networks* in *Advances in Neural Information Processing Systems* (eds Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S. & Garnett, R.) **30** (Curran Associates, Inc., 2017), 4878.

185. Geifman, Y. & El-Yaniv, R. SelectiveNet: A Deep Neural Network with an Integrated Reject Option (2019).

186. Kipf, T. N. & Welling, M. Semi-Supervised Classification with Graph Convolutional Networks (2016).

187. Wu, F., Zhang, T., de Souza Jr., A. H., Fifty, C., Yu, T. & Weinberger, K. Q. Simplifying Graph Convolutional Networks (2019).

188. Li, Y., Tarlow, D., Brockschmidt, M. & Zemel, R. Gated Graph Sequence Neural Networks (2015).

189. Huang, P.-S., Stanforth, R., Welbl, J., Dyer, C., Yogatama, D., Gowal, S., Dvijotham, K. & Kohli, P. Achieving Verified Robustness to Symbol Substitutions via Interval Bound Propagation (2019).

190. Jia, R., Raghunathan, A., Göksel, K. & Liang, P. Certified Robustness to Adversarial Word Substitutions (2019).

191. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. & Chintala, S. *PyTorch: An Imperative Style, High-Performance Deep Learning Library* in *Advances in Neural Information Processing Systems* (eds Wallach, H., Larochelle, H., Beygelzimer, A., Alché-Buc, F., Fox, E. & Garnett, R.) **32** (Curran Associates, Inc., 2019), 8026.

192. Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., Li, M., Zhou, J., Huang, Q., Ma, C., *et al.* Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909. 01315* (2019).

193. Gurobi Optimization, L. *Gurobi Optimizer Reference Manual* http://www.gurobi.com. 2020.

194. Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J. & Kaiser, Ł. Universal Transformers (2018).

195. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* **15**, 1929 (2014).

196. Kingma, D. P. & Ba, J. Adam: A Method for Stochastic Optimization (2014).

197. Graves, A. Adaptive Computation Time for Recurrent Neural Networks (2016).

198. Tsipras, D., Santurkar, S., Engstrom, L., Turner, A. & Madry, A. Robustness May Be at Odds with Accuracy (2018).

199. Dai, H., Li, H., Tian, T., Huang, X., Wang, L., Zhu, J. & Song, L. Adversarial Attack on Graph Structured Data (2018).

200. Zügner, D., Akbarnejad, A. & Günnemann, S. *Adversarial Attacks on Neural Networks for Graph Data* in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Association for Computing Machinery, London, United Kingdom, 2018), 2847.

201. Zügner, D. & Günnemann, S. *Adversarial Attacks on Graph Neural Networks via Meta Learning* in *International Conference on Learning Representations* (2018).

202. Zhang, H., Li, Z., Li, G., Ma, L., Liu, Y. & Jin, Z. *Generating Adversarial Examples for Holding Robustness of Source Code Processing Models* in *Proceedings of the AAAI Conference on Artificial Intelligence* **34** (2020), 1169.

203. Ramakrishnan, G., Henkel, J., Wang, Z., Albarghouthi, A., Jha, S. & Reps, T. Semantic Robustness of Models of Source Code (2020).

204. Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G. & Dean, J. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer (2017).

205. Zhang, Y., Albarghouthi, A. & D'Antoni, L. Robustness to Programmable String Transformations via Augmented Abstract Training (2020).

206. Katz, G., Barrett, C., Dill, D. L., Julian, K. & Kochenderfer, M. J. *Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks* in *Computer Aided Verification* (Springer International Publishing, 2017), 97.

207. De Moura, L. & Bjørner, N. Satisfiability modulo theories: introduction and applications. *Commun. ACM* **54**, 69 (2011).

208. Dutertre, B. *Yices 2.2* in *Computer Aided Verification* (Springer International Publishing, 2014), 737.

209. Cimatti, A., Griggio, A., Schaafsma, B. J. & Sebastiani, R. *The MathSAT5 SMT Solver* in *Tools and Algorithms for the Construction and Analysis of Systems* (Springer Berlin Heidelberg, 2013), 93.

210. Niemetz, A., Preiner, M. & Biere, A. Boolector 2.0. *J. Satisf. Boolean Model. Comput.* **9**, 53 (2014).

211. De Moura, L. & Passmore, G. O. in *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune* (eds Bonacina, M. P. & Stickel, M. E.) 15 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2013).

212. Pennington, J., Socher, R. & Manning, C. D. *Glove: Global vectors for word representation* in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (2014), 1532.

213. Joulin, A., Grave, E., Bojanowski, P. & Mikolov, T. Bag of Tricks for Efficient Text Classification (2016).

214. Ross, S., Gordon, G. & Bagnell, D. *A reduction of imitation learning and structured prediction to no-regret online learning* in *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (2011), 627.

215. Clare, A. & King, R. D. *Knowledge Discovery in Multi-label Phenotype Data* in *Principles of Data Mining and Knowledge Discovery* (Springer Berlin Heidelberg, 2001), 42.

216. Glorot, X. & Bengio, Y. *Understanding the difficulty of training deep feedforward neural networks* in *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (2010), 249.

217. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., *et al.* Analyzing program termination and complexity automatically with AProVE. *J. Automat. Reason.* **58**, 3 (2017).

218. Barrett, C., Fontaine, P. & Tinelli, C. *AProVE Benchmarks* `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NIA/tree/master/AProVE`. 2016.

219. Barrett, C., Fontaine, P. & Tinelli, C. *hycomp Benchmarks* `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NRA/tree/master/hycomp`. 2016.

220. Barrett, C., Fontaine, P. & Tinelli, C. *core Benchmarks* `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/tree/master/bruttomesso/core`. 2016.

221. Barrett, C., Fontaine, P. & Tinelli, C. *leipzig Benchmarks* `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NIA/tree/master/leipzig`. 2016.

222. Barrett, C., Fontaine, P. & Tinelli, C. *Sage2 Benchmarks* `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/Sage2`. 2016.

223. Godefroid, P., Levin, M. Y., Molnar, D. A., *et al. Automated Whitebox Fuzz Testing* in *NDSS* **8** (2008), 151.

224. Barrett, C., Fontaine, P. & Tinelli, C. *The Satisfiability Modulo Theories Library (SMT-LIB)* `www.SMT-LIB.org`. 2016.

225. Xu, L., Hutter, F., Hoos, H. H. & Leyton-Brown, K. SATzilla: Portfolio-based Algorithm Selection for SAT. *J. Artif. Intell. Res.* **32**, 565 (2008).

226. Bridge, D., O'Mahony, E. & O'Sullivan, B. in *Autonomous Search* (eds Hamadi, Y., Monfroy, E. & Saubion, F.) 73 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012).

227. Amadini, R., Gabbrielli, M. & Mauro, J. SUNNY: a Lazy Portfolio Approach for Constraint Solving (2013).

228. Hurley, B., Kotthoff, L., Malitsky, Y. & O'Sullivan, B. *Proteus: A Hierarchical Portfolio of Solvers and Transformations* in *Integration of AI and OR Techniques in Constraint Programming* (Springer International Publishing, 2014), 301.

229. Kadioglu, S., Malitsky, Y., Sellmann, M. & Tierney, K. *ISAC-Instance-Specific Algorithm Configuration* in *ECAI* **215** (2010), 751.

230. Ramirez, N. G., Hamadi, Y., Monfroy, E. & Saubion, F. *Evolving SMT Strategies* in *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)* (2016), 247.

231. Lagoudakis, M. G. & Littman, M. L. Learning to Select Branching Rules in the DPLL Procedure for Satisfiability. *Electronic Notes in Discrete Mathematics* **9**, 344 (2001).

232. Somol, P., Pudil, P. & Kittler, J. Fast branch & bound algorithms for optimal feature selection. *IEEE Trans. Pattern Anal. Mach. Intell.* **26**, 900 (2004).

233. Loth, M., Sebag, M., Hamadi, Y. & Schoenauer, M. *Bandit-Based Search for Constraint Programming* in *Principles and Practice of Constraint Programming* (Springer Berlin Heidelberg, 2013), 464.

234. Samulowitz, H. & Memisevic, R. *Learning to solve QBF* in *AAAI* **7** (2007), 255.

235. Khalil, E. B., Le Bodic, P., Song, L., Nemhauser, G. & Dilkina, B. *Learning to branch in mixed integer programming* in *Thirtieth AAAI Conference on Artificial Intelligence* (2016).

236. Irving, G., Szegedy, C., Alemi, A. A., Eén, N., Chollet, F. & Urban, J. Deepmath-deep sequence models for premise selection. *Adv. Neural Inf. Process. Syst.* **29**, 2235 (2016).

237. Wang, M., Tang, Y., Wang, J. & Deng, J. Premise selection for theorem proving by deep graph embedding. *Adv. Neural Inf. Process. Syst.* (2017).

238. Loos, S., Irving, G., Szegedy, C. & Kaliszyk, C. Deep network guided proof search. *arXiv preprint arXiv:1701.06972* (2017).

239. Nudelman, E., Leyton-Brown, K., Hoos, H. H., Devkar, A. & Shoham, Y. *Understanding Random SAT: Beyond the Clauses-to-Variables Ratio* in *Principles and Practice of Constraint Programming – CP 2004* (Springer Berlin Heidelberg, 2004), 438.

240. Hutter, F., Hoos, H. H., Leyton-Brown, K. & Stuetzle, T. ParamILS: An Automatic Algorithm Configuration Framework. *J. Artif. Intell. Res.* **36**, 267 (2009).

241. Ansótegui, C., Sellmann, M. & Tierney, K. A gender-based genetic algorithm for the automatic configuration of algorithms. *International Conference on* (2009).

242. Hutter, F., Hoos, H. H., Leyton-Brown, K. & Murphy, K. *Time-Bounded Sequential Parameter Optimization* in *Learning and Intelligent Optimization* (Springer Berlin Heidelberg, 2010), 281.

243. Hutter, F., Hoos, H. H. & Leyton-Brown, K. *Sequential Model-Based Optimization for General Algorithm Configuration* in *Learning and Intelligent Optimization* (Springer Berlin Heidelberg, 2011), 507.

244. KhudaBukhsh, A. R., Xu, L., Hoos, H. H. & Leyton-Brown, K. *SATenstein: automatically building local search SAT solvers from components* in *Proceedings of the 21st international jont conference on Artifical intelligence* (Morgan Kaufmann Publishers Inc., Pasadena, California, USA, 2009), 517.

245. Bugariu, A. & Müller, P. *Automatically Testing String Solvers* in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), 1459.

246. Winterer, D., Zhang, C. & Su, Z. *Validating SMT solvers via semantic fusion* in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Association for Computing Machinery, London, UK, 2020), 718.

247. Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S. & Tarlow, D. DeepCoder: Learning to Write Programs (2016).

248. psd2mobi. "https://www.psd2mobi.com/service/psd-to-android-ui". 2018.

249. replia. "http://www.replia.io/". 2018.

250. psd2android. "http://www.psd2androidxml.com/". 2018.

251. apptype. "https://www.apptype.io/". 2018.

252. Zeidler, C., Lutteroth, C., Stuerzlinger, W. & Weber, G. *Evaluating Direct Manipulation Operations for Constraint-Based Layout* in *Human-Computer Interaction – INTERACT 2013* (Springer Berlin Heidelberg, 2013), 513.

253. Zeidler, C., Lutteroth, C., Sturzlinger, W. & Weber, G. *The auckland layout editor: an improved GUI layout specification process* in *Proceedings of the 26th annual ACM symposium on User interface software and technology* (Association for Computing Machinery, St. Andrews, Scotland, United Kingdom, 2013), 343.

254. Swearngin, A., Dontcheva, M., Li, W., Brandt, J., Dixon, M. & Ko, A. J. in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* 1 (Association for Computing Machinery, New York, NY, USA, 2018).

255. Corrado, A., Lamp, A., Walsh, B., Aryee, E., Yuen, E., Matthews, G., Madiedo, J., Laval, J., Torres, L., Leger, M., Hsu, P., Chen, P., Rait Seth Chong, T., Alharthi, W. & Tu, X. *Ink To Code* "https://www.microsoft.com/en-us/garage/profiles/ink-to-code/". 2018.

256. Reiss, S. P. *Seeking the user interface* in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* (Association for Computing Machinery, Vasteras, Sweden, 2014), 103.

257. Zeidler, C., Weber, G., Stuerzlinger, W. & Lutteroth, C. *Automatic Generation of User Interface Layouts for Alternative Screen Orientations* in *Human-Computer Interaction - INTERACT 2017* (Springer International Publishing, 2017), 13.

258. Huang, R., Long, Y. & Chen, X. *Automaticly Generating Web Page From A Mockup* in *SEKE* (2016), 589.

259. Nguyen, T. A. & Csallner, C. *Reverse Engineering Mobile Application User Interfaces with REMAUI (T)* in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2015), 248.

260. Beltramelli, T. *pix2code: Generating Code from a Graphical User Interface Screenshot* in *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (Association for Computing Machinery, Paris, France, 2018), 1.

261. Chen, C., Su, T., Meng, G., Xing, Z. & Liu, Y. *From UI design image to GUI skeleton: a neural machine translator to bootstrap mobile GUI implementation* in *Proceedings of the 40th International Conference on Software Engineering* (Association for Computing Machinery, Gothenburg, Sweden, 2018), 665.

262. Amigo. *Amigo* "https://play.google.com/store/apps/details?id=com.amigotrip.android". 2018.

263. Asobimasu. *Asobimasu* "https://github.com/DipanshKhandelwal/Asobimasu". 2018.

264. Candid. *Candid* "https://play.google.com/store/apps/details?id=in.voiceme.app.voiceme". 2018.

265. FBook. *FBook* "https://play.google.com/store/apps/details?id=com.framgia.book". 2018.

266. Lidstone, G. J. Note on the general case of the Bayes-Laplace formula for inductive or a posteriori probabilities. *Transactions of the Faculty of Actuaries* **8**, 13 (1920).

267. Johnson, W. E. Probability: The Deductive and Inductive Problems. *Mind* **41**, 409 (1932).

268. Choudhary, S. R., Prasad, M. R. & Orso, A. *X-PERT: Accurate identification of cross-browser issues in web applications* in *2013 35th International Conference on Software Engineering (ICSE)* (2013), 702.

269. Choudhary, S. R., Prasad, M. R. & Orso, A. *CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications* in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* (2012), 171.

270. Mahajan, S., Li, B., Behnamghader, P. & Halfond, W. G. J. *Using Visual Symptoms for Debugging Presentation Failures in Web Applications* in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (2016), 191.

271. Panchekha, P., Geller, A. T., Ernst, M. D., Tatlock, Z. & Kamil, S. Verifying that web pages have accessible layout. *SIGPLAN Not.* **53**, 1 (2018).

272. Moran, K., Li, B., Bernal-Cárdenas, C., Jelf, D. & Poshyvanyk, D. *Automated reporting of GUI design violations for mobile apps* in *Proceedings of the 40th International Conference on Software Engineering* (Association for Computing Machinery, Gothenburg, Sweden, 2018), 165.

273. Ellis, K., Ritchie, D., Solar-Lezama, A. & Tenenbaum, J. *Learning to Infer Graphics Programs from Hand-Drawn Images* in *Advances in Neural Information Processing Systems* (eds Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N. & Garnett, R.) **31** (Curran Associates, Inc., 2018), 6059.

274. Chugh, R., Hempel, B., Spradlin, M. & Albers, J. Programmatic and direct manipulation, together at last. *SIGPLAN Not.* **51**, 341 (2016).

275. Ellis, K., Solar-Lezama, A. & Tenenbaum, J. B. in *Advances in Neural Information Processing Systems 29* 1297 (Curran Associates, Inc., Barcelona, Spain, 2016).

276. Menon, A., Tamuz, O., Gulwani, S., Lampson, B. & Kalai, A. *A Machine Learning Framework for Programming by Example* in *Proceedings of The 30rd International Conference on Machine Learning* **28-I** (PMLR, Atlanta, Georgia, USA, 2013), 187.

277. Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P. & Gulwani, S. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples (2018).

278. Lee, W., Heo, K., Alur, R. & Naik, M. Accelerating search-based program synthesis using learned probabilistic models. *SIGPLAN Not.* **53**, 436 (2018).

279. Johnson, D., Larochelle, H. & Tarlow, D. Learning Graph Structure With A Finite-State Automaton Layer. *Adv. Neural Inf. Process. Syst.* **33** (2020).