UCLA
UCLA Electronic Theses and Dissertations

Title
Ensuring Correctness of Modern Software Systems by Example

Permalink
https://escholarship.org/uc/item/9cd41151

Author
Sivaraman, Aishwarya

Publication Date
2022

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA

Los Angeles

Ensuring Correctness of Modern Software Systems by Example

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Aishwarya Sivaraman

2022

© Copyright by

Aishwarya Sivaraman

2022

ABSTRACT OF THE DISSERTATION

Ensuring Correctness of Modern Software Systems by Example

by

Aishwarya Sivaraman
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2022
Professor Todd D. Millstein, Chair

Software is intertwined in our daily lives: it manages sensitive personal information, allows us to connect with our peers, and controls devices like planes, cars, etc. Recent years have seen a surge in the adoption of a new kind of software system that uses Machine Learning and Artificial Intelligence due to an abundance of data. Owing to the ubiquitous nature, the security and reliability of these systems are paramount. However, due to these systems' complex nature, they are fragile and suffer from design and implementation flaws that make them unreliable. Unfortunately, building systems with provable guarantees is still a niche domain and remains largely unapproachable for the software development community, since these techniques require significant investment in terms of time and effort.

In this dissertation, I present verification techniques that ease the manual burden required to build reliable software and machine learning systems. The proposed techniques leverage specifications in the form of input/output examples and program synthesis techniques to ensure system correctness. I present the first approach that reduces the general lemma synthesis problem to a data-driven program synthesis problem, for easing the proof burden in a foundational verification setting. I then present an extension to this data-driven synthesis

approach that expands the class of lemmas that can be synthesized. I conclude with the first counterexample-guided verification approach that can provably enforce correctness properties in Neural Networks.

The dissertation of Aishwarya Sivaraman is approved.

Guy Van den Broeck

Sorin Lerner

Jens Palsberg

Todd D. Millstein, Committee Chair

University of California, Los Angeles

2022

To my husband and my family for their unconditional love and for always inspiring me to be the best version of myself

TABLE OF CONTENTS

1 Introduction **1**

1.1 Why do Systems need Verification? 1

1.2 How do we currently verify systems? 3

1.2.1 Foundational Verification of Software Systems 3

1.2.2 Verification of Machine Learning Systems 6

1.3 Thesis Statement and Contributions 7

1.3.1 Example-Driven Lemma Synthesis for Interactive Theorem Proving . 8

1.3.2 Increasing Expressivity of Example-Driven Lemma Synthesis 9

1.3.3 Counterexample-Guided Verification of Neural Networks 10

2 Example-Driven Lemma Synthesis for Interactive Theorem Proving . . **11**

2.1 Introduction 11

2.2 Overview 15

2.2.1 Motivating Example 15

2.2.2 Approach 18

2.3 Algorithms 23

2.3.1 Preliminaries 23

2.3.2 Lemma Synthesis 24

2.3.3 Filtering 26

2.3.4 Ranking 27

2.3.5 Discussion 27

2.4 Implementation 28

2.4.1 Example Generation 29
2.4.2 Synthesis 30
2.4.3 Filtering and Ranking 30
2.4.4 Discussion 31
2.5 Experimental Results 31
2.5.1 Benchmark Suite 32
2.5.2 Experimental Setup 34
2.5.3 Synthesized Helper Lemmas 34
2.5.4 Comparison with Other Approaches 39
2.5.5 Sensitivity 41
2.6 Related Work 45
2.6.1 Lemma Synthesis 45
2.6.2 Data-driven Invariant Inference 46
2.6.3 Automated Proofs for Interactive Theorem Provers 47
2.7 Summary 48

3 Increasing Expressivity of Example-Driven Lemma Synthesis 49
3.1 Introduction 49
3.2 Overview 51
3.2.1 Motivating Example for Generating Equality Lemmas about Subterms 51
3.2.2 Motivating Example for Generating Conditional Lemmas 56
3.3 Algorithms 61
3.3.1 Equality Lemmas about Subterms 61
3.3.2 Conditional Lemmas 63

3.4 Experimental Evaluation 64
3.4.1 Benchmark Suite 64
3.4.2 Experimental Setup 66
3.4.3 Synthesized Helper Lemmas 66
3.4.4 Comparison with `lfind` 70
3.5 Summary 71
3.6 Future Work 72
3.6.1 Synthesizing Conditional Lemmas 72
3.6.2 Multiple Helper Lemmas 74

4 Counterexample-Guided Verification of Neural Networks 76
4.1 Introduction 76
4.2 Preliminaries: Finding Monotonicity Counterexamples 77
4.3 Counterexample-Guided Monotonic Prediction 80
4.3.1 Envelope Construction 81
4.3.2 Empirical Evaluation of Monotonic Envelopes 85
4.4 Counterexample-Guided Monotonicity Enforced Training 88
4.4.1 Counterexample-guided Learning 89
4.4.2 Empirical Evaluation of `COMET` 90
4.5 Related Work 95
4.6 Summary 97
4.7 Extensions and Future Work 97
4.7.1 Fairness 97
4.7.2 Scalability 98

5 Conclusion **99**

References **102**

LIST OF FIGURES

1.1 Typical Interactive Workflow when using Coq proof assistant. 4

2.1 A partial proof of a theorem in Coq that requires an auxiliary lemma. 16

2.2 The proof state when the user gets stuck. 16

2.3 A full proof provided by `lfind`. 17

2.4 Overview of `lfind`. 19

2.5 Given a stuck goal, `lfind` implements generalization, synthesis, filtering, and ranking in conjunction with existing tools to generate candidate lemmas. 28

2.6 `lfind` has a median total runtime of 4.8 min. Further, the tool has a median runtime of 1.2 min for the 109 cases (see Table 2.1) where it was able to a find a full automated proof (Λ_1). 38

2.7 `lfind` reduces the number of lemmas by 89.9% on average after application of both filters. 39

2.8 Total runtime of `lfind` increases when increasing number of synthesis terms per sketch. Runtime almost doubles when k increases from 5 to 15, while it is 1.3x more when it is increases from 15 to 25. 42

2.9 Median runtime of `lfind` decreases with an increase in PROVERBOT9001 timeout. While this is unintuitive, this is because the prover is allocated more time per call, enabling it to prove a candidate lemma earlier, which was otherwise not provable using a smaller timeout. 43

2.10 Median runtime of `lfind` is unaffected when increasing MYTH timeout. 43

2.11 There is a modest increase in median runtime of `lfind` from 3.4 min to 4.5 min when generating synthesis sketches from maximal terms compared to all terms. . 44

3.1 A partial proof of a theorem containing non-equality proposition in Coq that requires an auxiliary lemma. 52

3.2 The proof state when the user gets stuck. 52

3.3 A full proof provided by `lfind`. 54

3.4 A partial proof of a theorem in Coq that requires an auxiliary lemma. 57

3.5 The proof state when the user gets stuck. 58

3.6 Sketch Generation. 61

3.7 `lfind++` Algorithm. 62

3.8 Counterexample-Guided Refinement. 65

3.9 `lfind++` has a median total runtime of 2.6 min. Further, the tool has a median runtime of 1.0 min for the 186 cases (see Table 3.2) where it was able to a find a full automated proof (Λ_1). 70

3.10 Proof of a theorem in Coq that requires multiple helper lemmas. 72

3.11 The proof state when the user gets stuck. 72

3.12 Proof of a theorem in Coq that requires multiple helper lemmas. 74

4.1 Monotone envelopes around a simple non-monotone learned function 80

4.2 Empirically, the best learned baseline model is not monotonic. The figure presents the percentage of examples that have an upper or lower envelope counterexample for the *Auto MPG* dataset. 85

4.3 Prediction Time (s) vs. #Monotonic Features 88

4.4 Prediction Time (s) vs. Model Size 88

4.5 Monotonicity is a good inductive bias and helps in improving model accuracy. However, there is a tradeoff between performance and reducing the number of examples that have counterexamples. 93

LIST OF TABLES

2.1 Results 33

2.2 A sample of `lfind` synthesized lemmas and their associated rank and category. . 36

3.1 `lfind++` Benchmark Summary 66

3.2 Results 67

3.3 A sample of `lfind++` synthesized lemmas and their associated rank and category. 68

3.4 Expressivity extensions of `lfind++` significantly outperforms vanilla `lfind`. . . 70

4.1 Best parameter configurations on each dataset for each data fold found using grid search for baseline neural networks ($\mathrm{NN_b}$). 86

4.2 Empirically, the best baseline neural network model ($\mathrm{NN_b}$) trained on data is not monotonic. The table presents the percentage of examples that have an upper or lower envelope counterexample. 87

4.3 For regression (MSE, Left Table) and classification (Accuracy, Right Table) datasets, envelope predictions on test data have similar quality compared to baseline models. This means we can guarantee monotonic predictions with little to no loss in model quality. 87

4.4 Monotonicity is an effective inductive bias. Counterexample-guided Learning (CGL) improves the quality of the baseline model in regression (MSE, Left Table) and classification (Accuracy, Right Table) datasets 90

4.5 Counterexample-guided learning (CGL) is able to make a model more monotonic by reducing the number of test and train counterexamples compared to the baseline model ($\mathrm{NN_b}$). However, the algorithm is unable to guarantee monotonicity, motivating the need for monotonic *envelopes*. 91

4.6 For regression (MSE, Left Table) and classification (Accuracy, Right Table) datasets, counterexample-guided learning improves the envelope quality 92

4.7 `COMET` outperforms Min-Max networks on all datasets. `COMET` outperforms DLN in regression datasets and achieves similar results in classification datasets. . . . 92

4.8 With monotonicity data outliers, `lfind` produces models that are more robust than the baseline models (NN_b) for regression (MSE, Left Table) and classification (Accuracy, Right Table) datasets. 94

ACKNOWLEDGMENTS

Research: it takes a village. I have been very fortunate to have had strong social support throughout my graduate school. While my journey through grad school was exciting, it was challenging at times, and this strong social circle kept me going through tough times.

I am very grateful to my advisor Todd Millstein for giving me an opportunity to be his student when I switched research directions in 2019. Todd is the best advisor I could have asked for, allowing me to be independent and providing me with guidance where needed. I tend to be productive and creative only closer to a deadline. Todd was very patient and let me work at my pace and believed that I could get good work done. Our research discussions tend to be intense, but it was constructive, and with each discussion, I got better at presenting my research. Todd has also given me good advice over the years I have known him, even before becoming his student.

I want to thank Guy Van den Broeck, for being a great mentor, friend, and constant source of support, and inspiration. I did a course project with him in the first year, and he was nothing but encouraging, which lead me to publish my first work, ALICE with him. When writing the paper, he would encourage me to write and edit the draft with his feedback. At that point, I was not too happy and wanted him to help edit the draft. In retrospect, I realize that inculcating this writing independence early on was very useful. This training helped me gain confidence in writing initial drafts and editing them. He always had great advice and when grad school was getting tough, he taught me how to navigate this journey without getting overwhelmed. Guy played a crucial role, along with Todd in molding me to be a successful researcher.

I want to thank Sorin Lerner for being a wonderful collaborator and for serving on my committee. I want to thank Jens Palsberg for agreeing to serve on my committee. I also appreciate the insightful discussions and constructive criticism offered by my collaborators and co-authors. I want to thank Golnoosh for continuing to mentor me over the years. I

want to thank my mentors at Meta, Dr. Satish Chandra, and Dr. Walid Taha for helping me grow as a researcher in the industry and for their support after the internship. I also owe a debt of appreciation to Joseph Brown and Juliana Alvarez for being patient, and friendly, and helping me with my endless list of queries regarding administrative processing.

Many thanks to my friends and lab mates at UCLA who kept my spirits high throughout this journey: Siva, Shaghayegh, Kareem, Akshay, Pradeep, Arjun, Saikrishna, Tianyi, Saswat, Gulzar, Christian, John, Aayush, Vidhushi, Ashutosh, Alexis, Steven.

Siva and I started our Ph.D. and grew together over the years. I have spent countless hours chatting with him about research and non-research topics and playing Catan. He is the best roommate I could have asked for. When things got tough, he would always listen and provide candid advice and encouraged me to go on. He is also a great cook, and I definitely miss his "capsicum curry".

Shaghayegh was one of the warm friendly faces that greeted me when I first came to the lab. We instantly bonded over our shared interest in food ("shekamoos"). Thanks to Shaghayegh for being supportive, and encouraging, for sharing academic concerns, and for being a great source of comfort.

Many thanks to Kareem for being a great friend and for being the "fish in a pond".

Thanks to Irma for providing moral support and for being a great company during the first few years that I took to adjust to the life of grad school.

I want to thank my close friend Sai Ganesh Nagarajan with whom I have grown as a researcher since undergrad, making some of the best memories during the past 12 years. Although in different universities, we embarked on the Ph.D. journey together. Research can get lonely very fast, but having someone who understands this journey, from acceptances to rejects is a blessing. We have spent countless hours talking about random topics and I could not have worked on any ML project without his guidance, he is always my go-to machine learning "guru". Thanks for being patient, listening to my endless stories, crashing

conferences, and giving great advice! We are definitely on track to submit our work to Neurips 2050!

This thesis is dedicated to my husband (Murali) and my family. I am very grateful to Murali for being my personal rubber duck and cheerleader since NUS. In fact, he identified my research aptitude, pushing me to pursue research (which was the last thing I wanted to do in undergrad), guiding me through every step of the process right from my application to grad school to thesis writing. To my father (Sivaraman), for teaching me the importance of continued learning, pursuing your passion, embracing any situation with a smile, and remaining humble. To my mother (Lalitha), for being a constant source of encouragement and support, and for teaching me to dream big. To my sister (Ranjani), for teaching me to be independent and for being supportive. I would also like to thank Vasanthy Ramanujam and Ramanujam for their unconditional support, faith, and belief.

Finally, I would like to thank Google for supporting and recognizing my research with a Ph.D. fellowship.

VITA

2010, 2014	Full Scholarship for Undergraduate Studies at National University of Singapore (NUS), Singapore Airlines-Neptune Orient Lines.
2014	B.Eng (Hons) Computer Engineering, NUS.
2016-2017	Research Assistant with Prof. Khoo Siau Cheng, NUS.
2019	Teaching Assistant, Principles and Practices of Computing, UCLA.
2019	Founding Member and Co-President, Graduate Women in Computer Science, UCLA.
2020-2021	Member of Artifact Evaluation Committee, PLDI '20 & '21, OOPSLA '20.
2020	Intern (remote) with Dr. Satish Chandra, Meta Platforms, Inc., California
2021	Google Research Fellow
2021	Intern (remote) with Dr. Walid Taha, Meta Platforms, Inc., California
2022	Outstanding Mentorship Award, UCLA

PUBLICATIONS

Sivaraman, Aishwarya, Tianyi Zhang, Guy Van den Broeck, and Miryung Kim. "Active inductive logic programming for code search." In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 292-303. IEEE, 2019.

Sivaraman, Aishwarya, Golnoosh Farnadi, Todd Millstein, and Guy Van den Broeck. "Counterexampleguided learning of monotonic neural networks." Advances in Neural Information Processing Systems 33 (NeurIPS): 11936-11948, 2020.

Lau, Jason*, **Aishwarya Sivaraman***, Qian Zhang*, Muhammad Ali Gulzar, Jason Cong, and Miryung Kim. "HeteroRefactor: refactoring for heterogeneous computing with FPGA." In 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pp. 493-505. IEEE, 2020.[1]

Mohammadi, Kiarash, **Aishwarya Sivaraman**, and Golnoosh Farnadi. "Post-processing Counterexample-guided Fairness Guarantees in Neural Networks" In Combining Learning and Reasoning: Programming Languages, Formalisms, and Representations. 2021.

Sivaraman, Aishwarya, Rui Abreu, Andrew Scott, Tobi Akomolede, and Satish Chandra. "Mining Idioms in the Wild." In 2022 IEEE/ACM 44th International Conference on Software Engineering, Software Engineering in Practice (ICSE-SEIP), 2022.

Mohammadi, Kiarash, **Aishwarya Sivaraman**, and Golnoosh Farnadi. "FETA: Fairness Enforced Verifying, Training, and Predicting Algorithms for Neural Networks." arXiv preprint arXiv:2206.00553 (2022).

Aishwarya Sivaraman, Alex Sanchez-Stern, Bretton Chen, Sorin Lerner, Todd Millstein. "Data-Driven Lemma Synthesis for Interactive Proofs." Proceedings of the ACM on Programming Languages (OOPSLA 2022) *To Appear*.

[1] *are equal co-first authors ordered alphabetically by their last names

CHAPTER 1

Introduction

How can we enable programmers across different domains to formally guarantee that a software or machine learning system is "error-free" – for a given definition of correctness?

The field of formal verification attempts to answer this question and has come a long way in realizing this goal. In fact, the first attempt to prove such correctness was proposed by Alan Turing in his seminal paper "Checking a Large Routine" [Tur89]. In this work, Turing proved that the factorial function always terminates and produces the factorial of its input. The key insight from this work is "proof decomposition", i.e. Turing proposed to prove lemmas for each instruction and stitch these together to prove the correctness of the full program. This insight led to the development of several *program verification* algorithms, languages, and tools that allow programmers to build systems with strong guarantees, by formally proving the desired properties [Tel80, DKW08, BH14]. However, the increasing scale of software systems and rapidly evolving machine learning systems have made current verification techniques onerous (§ 1.2.1) or inapplicable (§ 1.2.2). This thesis aims to change that by proposing *example-driven* automated techniques to reduce the cost of verification, thereby aiding a programmer's effort in building reliable systems.

1.1 Why do Systems need Verification?

Software is intertwined in our daily lives: it manages sensitive personal information, allows us to connect with our peers, and controls devices like planes, cars, etc. Further, recent years have

seen a surge in the adoption of a new kind of software system that uses Machine Learning (ML) and Artificial Intelligence (AI) due to an abundance of data. Machine learning algorithms have helped make tremendous progress in complex computing tasks like object recognition, natural language processing, and so on. This has led to the development of a new kind of software system that is a combination of manually written code and automatically trained models. As software becomes more complex and assumes a greater role in our lives, it is important that we develop techniques that ensure its safety and reliability. However, due to these systems' complex nature, they are fragile and suffer from design and implementation flaws that make them unreliable. New methods are constantly sought to reduce complexity and improve the reliability of software and machine learning systems. Despite these growing efforts, in the past decade, researchers and/or hackers have identified software bugs, specifically security vulnerabilities that had a major impact on the economy and society. Examples include the Log4j vulnerability in Java [log], the Heartbleed vulnerability in Openssl [DLK14], and in 2011 Jerome Radcliffe showed they could wirelessly hack an insulin pump and cause it to deliver incorrect dosages of medication [Rad11].

Furthermore, machine learning systems exacerbate these problems since the prevailing practice is to train a system on a training data set and then test it on another set. While training with data reveals the average-case performance of models, it provides no guarantees on correctness invariants. Therefore, there are looming concerns about the privacy, security, fairness, and explainability of AI/ML-based models. These models are associated with a set of risks and concerns including errors in AI software, cyber-attacks, and safety of AI-based systems [AOS16, RDT15]. The failure of ML models has produced catastrophic results. For example, autonomous vehicles have been involved in numerous fatal crashes despite very high test and training accuracy. Some of the machine learning models like Deep Neural Networks are vulnerable to adversarial attacks that make them unsafe and unstable [PMG17, GSS18].

1.2 How do we currently verify systems?

1.2.1 Foundational Verification of Software Systems

One of the promising pillars of software verification is foundational verification, where programmers provide formal proof of the desired property of a program. Typically programmers use an interactive theorem prover (also known as *proof assistants*) such as Coq [BBC97] or Isabelle/HOL [Pau86] to state and prove the correctness properties of their programs. A formal proof can be "machine-checked", i.e. given a representation of a formal system and a proof of correctness, a computer program can decide if the proof is valid. This approach provides high levels of assurance. The interactive theorem prover makes sure that proofs of program properties are done in complete detail, without any implicit assumptions or forgotten proof obligations. Since this is an appealing advantage, the use of theorem proving for verification has been used to build pieces of system infrastructure that are widely relied upon to be correct and secure. For example, in compilers [Ler09], file systems [CCK17], database systems [MMS10], cryptographic primitives [App15], etc. Figure 1.1 illustrates a typical proof workflow when using an interactive theorem prover. A programmer starts by providing the proof assistant with the program definitions and specifications. To write a proof, they use tactics (like `induction`), and the proof assistant responds to each tactic by refining the current goal to some subgoal. If a tactic cannot be applied to the current goal, the proof assistant provides error feedback to the programmer. This loop of tactics and goals continues until no goals remain, at which point the programmer has constructed a sequence of tactics called a proof script.

Despite their tremendous success in providing provable guarantees, the adoption of foundational verification is limited. Unfortunately, building systems with provable guarantees is still a niche domain and remains largely unapproachable for the software development community. Although formal methods research has produced techniques and tools [BBC97, App15] that aid in reliable system building, verification is too costly today [BH14] and hence

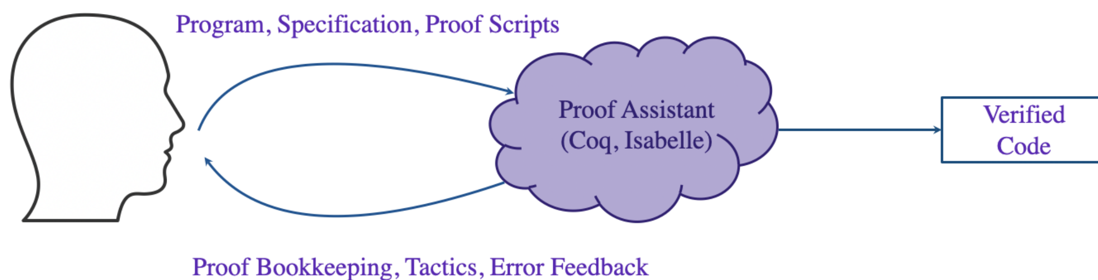

Figure 1.1: Typical Interactive Workflow when using Coq proof assistant.

is not done in practice, leaving software rife with bugs. This can be attributed to (1) large amounts of manual effort required to formalize the intended behavior, and (2) these techniques do not scale well for large software systems. First, a programmer should provide a formal specification or identify some key properties of their code and prove that the implementation follows these properties. A proof assistant is just a checker, therefore, all details of the proof need to be rigorously formalized in the theorem prover. A theorem prover doesn't a priori know any properties, for example, that addition is commutative `a + b = b + a` is true. To use this property as a lemma, it needs to be either proven, added as an assumption, or imported from an external library. Hence theorem proving remains esoteric, and the cost prohibits the proliferation of this approach as a go-to verification method. For example, the seL4 operating system kernel [KEH09] took 22 person-years to verify.

To combat the proof burden of interactive theorem proving, several techniques have been proposed to complete the proof steps by predicting the tactics [Chl13a] and lemmas to use [CK18a, KU15a]. Recent works have proposed the use of machine learning to reduce proof burden [SAS20a, BLR19a, FBG20a, GKU18]. For example, the IronFleet project used SMT solvers and Dafny – a verification language – to write a verified distributed system in 3.7 person-years [HHK15]. While this is an improvement over the manual proof burden of seL4 (22 person-years), it is still costly. This is due to the manual effort required by the programmer to provide necessary specifications and intermediate helper lemmas where necessary.

Techniques described so far aid a human prover with automation, there are works in the domain of automated theorem proving that aim to reduce manual proof burden by fully automating the proof search. The goal of any automated theorem proving (ATP) engine is to find proof of a given theorem statement with simply a push of a button. Typical ATP tries to find a proof using resolution refutation: it converts the premises and negation of the theorem into first-order clauses in conjunctive normal form. It then keeps generating new clauses by applying the resolution rule until an empty clause emerges, yielding a proof consisting of a long sequence of CNFs and resolutions [YD19a]. Exhaustive deductive reasoning tools quickly hit combinatorial explosion, and are unusable when reasoning with a very large number of facts without careful premise selection [BKP16]. Further, ATP representation of simple formulas can be long and complicated and make it difficult to benefit from the higher-level abstraction and manipulation that is common to human mathematical reasoning. Therefore, ATPs are limited in the type of problems they can solve. Since ATP is a search problem, to work around these difficulties, recent works focus on using machine learning for proof search [BLR19a, KUM18, Wha16a]. In addition to proof search, recent works have used machine learning algorithms for premise selection. The typical setting for the task of premise selection is the selection of a limited number of most relevant facts from a large corpus of knowledge, to prove a new conjecture [CK18a, ACE16]. Despite these advancements, ATPs suffer from a fundamental limitation that in general automatically proving that a property is true about a system can be undecidable. This limits the applicability of ATPs, emphasizing the importance of human-machine collaboration to formalize properties using proof-assistants.

While all of this prior work has been very helpful in pushing state-of-the-art, the vast majority of this work focuses on forms of automation techniques to help users prove a single lemma or subgoal within a lemma, using a given knowledge base. However, when we move beyond the simple example, the proof of the main theorem is typically decomposed into many smaller lemmas, and each of those lemmas has its helper lemmas too. One key challenge that limits the current techniques is the need to identify auxiliary lemmas that are required

in order to prove the desired theorem. For example, a lemma may be required in order to rewrite the subgoal at a particular point in the proof into a form that allows the induction hypothesis to be applied. As another example, the theorem's induction hypothesis may be too weak, thereby necessitating a stronger lemma that is amenable to an inductive proof. If the required auxiliary lemma is not provided as part of the knowledge base, the above-mentioned techniques fail to return a successful proof. Automating the discovery of these helper lemmas remain a challenge that this thesis addresses.

1.2.2 Verification of Machine Learning Systems

Neural networks and deep learning have revolutionized machine learning [KSH17, GBC16] achieving the state of the art performance on a wide range of complex prediction tasks. Neural networks are typically tested using the standard machine learning paradigm: If the performance of the network is sufficiently high on a test set that the network did not have access to while training, the network is deemed acceptable. However, this evaluation protocol is not sufficient in domains where for safety, ethical, and legal reasons, it is of utmost importance that decisions made by neural networks adhere to properties like fairness, monotonicity, etc. Even networks that perform well on a large sample of inputs may not correctly generalize to new situations and may be vulnerable to adversarial attacks [PMJ16].

Currently, there are three classes of techniques for the general verification of neural networks. (1) Constraint-based verification: These techniques take a neural network and a property of interest and encode the verification problem as a set of constraints that can be solved by a Satisfiability Modulo Theories (SMT) solver or a Mixed Integer Linear Program (MILP) solver. Such a prover can prove or disprove a property, returning a counterexample in the latter case [KBD17, HKW17, DJS18, Ehl17]. (2) Abstraction-based verification techniques employ abstract domains to evaluate the neural networks on sets of inputs. These techniques execute the neural network over an infinite set and show that all of those inputs satisfy desirable correctness properties. These techniques are incomplete and therefore are

more scalable than constraint-based verification approaches [GMD18, GDS18, WPW18]. (3) Certification-based verification techniques, for example, robustness-based certification techniques add regularizing terms to the loss function of the model such that the prediction for a small distance around a given input point does not change [SNV17, RSL18].

The first two verification works proposed above can only check for the presence or absence of correctness counterexamples. These techniques are post-hoc validation and currently, there is no way to rectify and encourage the model to obey correctness properties. Certification-based techniques do enforce the required behavior when learning a model, however, these guarantees are limited to only points that are very close to the train/test points. These techniques do not provide any guarantees for arbitrary points from the input distribution. Designing algorithms that provide provable correctness guarantees for all points in the input domain remains a challenge that this thesis addresses.

1.3 Thesis Statement and Contributions

The rise of modern software systems has led to a world where they control medical devices like insulin pumps, computer peripherals like printers, communication equipment like cellphones, various kinds of vehicles, and so on. Therefore it is of paramount importance that developers of these systems are equipped with the right tools and techniques to build reliable and secure systems. Further, as described above, building reliable modern software systems is onerous or does not provide provable guarantees, requiring new verification techniques and algorithms. My thesis is that, *it is possible to dramatically reduce the manual effort required by programmers to develop reliable modern software systems with automated techniques that use examples to verify and enforce key correctness properties.*

To design such techniques, we must address the following key concerns. First, we need to tackle the technical challenges that are necessary to overcome for that approach with a focus on reducing manual effort in practice. Second, we need to design approaches that are

extendable, such that they can support a variety of correctness properties. Lastly, we need to ensure that the proposed techniques do not cause significant performance degradation (specifically in the case of machine learning systems).

The key insight is that input/output examples of a system are easy to obtain, and act as an additional specification that helps navigate the verification search space efficiently. Hence, I use input/output examples of the system and example-based program synthesis techniques to design and develop approaches that ensure correctness. During my doctoral program, I have made significant progress on the proposed goals using this key insight. Through this thesis, I have made three contributions to reduce the manual effort needed to develop provably correct modern software systems. The following sub-sections enumerate this dissertation's contributions individually.

1.3.1 Example-Driven Lemma Synthesis for Interactive Theorem Proving

Interactive proofs of theorems often require auxiliary helper lemmas to prove the desired theorem. Existing approaches for automatically synthesizing helper lemmas fall into two broad categories. Some approaches are goal-directed, producing lemmas specifically to help a user make progress from a given proof state, but they have limited expressiveness in terms of the lemmas that can be produced. Other approaches are highly expressive and able to generate arbitrary lemmas from a given grammar, but they are completely undirected and hence not amenable to interactive usage.

In this work, I develop an approach to lemma synthesis that is both goal-directed and expressive. The key novelty is a technique for reducing lemma synthesis to a data-driven program synthesis problem, whereby examples for synthesis are generated from the current proof state. I also describe a technique to systematically introduce new variables for lemma synthesis, as well as techniques for filtering and ranking candidate lemmas for presentation to the user. I implement these ideas in a tool called `lfind`, which can be run as a Coq tactic. In an evaluation of four benchmark suites, `lfind` produces useful lemmas in 65% of the cases

where a human prover used a lemma to make progress. In these cases, `lfind` synthesizes a lemma that either enables a fully automated proof of the original goal or that matches the human-provided lemma.

1.3.2 Increasing Expressivity of Example-Driven Lemma Synthesis

Lemma synthesis approach proposed in § 1.3.1 suffers from two key limitations. First, the generated lemmas of `lfind` always have the same top-level structure as the goal. While these candidate lemmas are useful in applying to the full goal state or rewriting one side of the equality in the goal state, they fail to produce a useful candidate lemma in cases that require equality helper lemmas about the subterms in the goal. Second, many natural lemmas are also *conditional*, where a particular property is true only under certain circumstances. `lfind` fails to produce conditional candidate lemmas.

In this work, I propose two extensions to `lfind` that improve the expressivity of the data-driven paradigm for lemma synthesis. To address the first limitation of generating equalities about subterms, I extend the kind of lemma sketches defined by `lfind`. The key novelty is a technique to generate candidate lemmas based on subterms while reusing the data-driven synthesis problem setup in `lfind`. I address the second limitation by proposing a novel counterexample-guided refinement algorithm to generate conditional lemmas.

I implement these ideas in a tool called `lfind++`, which can be run as a Coq tactic. In an evaluation of six benchmark suites, with 323 evaluation locations, `lfind++` synthesizes useful lemmas in 76% of the cases where a human prover used a lemma to make progress. On these evaluation locations, `lfind` can only solve 49% of the cases, indicating the effectiveness of the proposed approach in improving expressivity.

1.3.3 Counterexample-Guided Verification of Neural Networks

The widespread adoption of deep learning is often attributed to its automatic feature construction with minimal inductive bias. However, in many real-world tasks, the learned function is intended to satisfy domain-specific constraints. I focus on monotonicity constraints, which are common and require that the function's output increases with increasing values of specific input features. I develop a counterexample-guided technique to provably enforce monotonicity constraints at prediction time. Additionally, I propose a technique to use monotonicity as an inductive bias for deep learning. It works by iteratively incorporating monotonicity counterexamples in the learning process. Contrary to prior work in monotonic learning, I target general ReLU neural networks and do not further restrict the hypothesis space. I have implemented these techniques in a tool called `COMET`. Experiments on real-world datasets demonstrate that the approach achieves state-of-the-art results compared to existing monotonic learners, and can improve the model quality compared to those that were trained without taking monotonicity constraints into account.

CHAPTER 2

Example-Driven Lemma Synthesis for Interactive Theorem Proving

2.1 Introduction

Interactive proof assistants [FHB97, Pau93, MKA15] are powerful frameworks for writing code with strong guarantees. While various tools exist to perform automated proof search [YD19b, GKU17, SAS20b, BLR19b, PLR20, SIS17, FBG20b, Wha16b] and to integrate external automated solvers [BBP11, KU15c, CK18b, KU15b], the manual proof burden remains high. One particular challenge is the need to identify auxiliary lemmas that are required to prove a desired theorem. For example, the theorem's induction hypothesis may be too weak, thereby necessitating a stronger lemma that is amenable to an inductive proof. As another example, a lemma may be required to rewrite a subgoal at a particular point in the proof into a form that allows the induction hypothesis to be applied.

Existing approaches to address this problem through a form of *lemma synthesis* fall into two categories. In the first category, heuristic rewrites are performed on the proof state at the point where the user is stuck to identify potentially useful lemmas [KM97, KS96, BSV93, JDB10, DF03, SDE12, Aub76, Cas85, Hum90, Hes92]. For example, the *generalization* technique [BM79, KM97] from ACL2 heuristically replaces one or more terms in the current subgoal with fresh variables. In the second category of approaches, candidate lemmas are generated from a grammar through a form of enumeration-based synthesis [CJR13, YFG19,

RK15]. For example, HipSpec [CJR13] enumerates many candidate lemmas and attempts to prove them with an automated prover.

The strength of the heuristic rewriting approach is that it is *goal-directed*, producing candidate lemmas that are directly related to the current proof state. However, the approach has *limited expressiveness*, as the space of possible candidates is dependent on a particular set of rewrite rules. The enumeration approach has the opposite strengths and weaknesses. Because candidate lemmas are enumerated from a grammar, they can be *highly expressive*. However, candidate lemmas are generated in an *undirected* fashion, independent of the particular state where the user is stuck. Hence this approach will generate many irrelevant lemmas and so is ill-suited for an interactive setting. Indeed none of the enumeration-based tools cited above support interactive usage.

In this chapter, I propose a new approach to lemma synthesis that combines the strengths of the existing approaches. I show how to reduce lemma synthesis to a *data-driven program synthesis* problem, which aims to synthesize an expression that meets a given set of input-output examples. The examples for synthesis are generated directly from the current proof state, ensuring that lemma candidates are targeted at the goal. At the same time, the approach enables the usage of off-the-shelf data-driven program synthesizers that generate expressions in a user-provided grammar [AGK13, OZ15, LCO20, FCD15, FOW16, MNB22]. This new approach allows us to successfully synthesize helper lemmas for more stuck proofs than ever before.

Reducing lemma synthesis to data-driven program synthesis requires us to solve several technical challenges. While data-driven synthesis is a common approach to generating other kinds of program invariants [GLM14, GNM16, END18, PSM16, ZMJ18, MPW20], for instance, loop invariants, these prior settings have several advantages that our setting lacks. In prior settings, the desired invariant is a predicate over a fixed set of variables, for example, the variables that are in scope at a loop. In contrast, it's common for auxiliary lemmas to require new variables that do not appear in the current proof state. Further, prior approaches employ

counterexample-guided inductive synthesis (CEGIS) [Sol09], because there exists a clear behavioral specification for the desired invariant: each candidate invariant is verified against the specification, and counterexamples become new input-output examples for synthesis. In our setting, we lack such a specification since a proof state can require an auxiliary lemma for many different reasons. Hence we cannot generate input-output examples using CEGIS. Finally, the lack of a specification also makes it difficult to determine whether any particular candidate lemma is useful.

To address the problem of lemmas that require variables not appearing in the proof state, I observe that the *generalization* technique [BM79, KM97] described above can be used not only to produce candidate lemmas but also as a systematic way to "lift" the current proof state to new variables for lemma synthesis. Hence our approach starts by producing all generalizations of the proof state, each formed by replacing one or more terms with fresh variables.

To generate examples for synthesis without counterexamples, I leverage the implicit observation underlying the heuristic rewriting approaches described earlier, that the necessary lemma often has a similar structure to the goal in the current proof state. The approach produces a set of *lemma sketches* for each generalized goal, each sketch consisting of a version of that goal but with one expression replaced by a *hole* to be synthesized. I sample valuations of the variables in the current goal to generate input examples, and the expected output value for each example is determined by the value of the hole's original expression. In this way, I require that the synthesized expression's behavior be consistent with that of the expression that it is replacing.

Finally, to address the lack of clear criteria for candidate lemmas to satisfy, I have developed techniques to *filter* candidate lemmas that are not useful and to *rank* the remaining candidates based on their likely utility to the user. Filtering removes lemmas that are determined to be either trivial, redundant, or invalid, the latter using existing tools for automated counterexample search [PHD15, CH00]. Since the ultimate utility of a lemma

is based on whether it is provable and allows the user to complete the current proof, the ranking approach employs existing tools for automated proof search to categorize lemmas for user inspection.

I have implemented this approach as a tactic for Coq and call the resulting tool `lfind`. Coq users can invoke `lfind` as a tactic at any point in their proof, and it will produce a set of ranked lemma candidates. Lemma synthesis in `lfind` is targeted for use in proving properties of programs (as opposed to other uses of interactive theorem proving, such as formalizing mathematics). This is a common use case for Coq, aligns with the focus of prior lemma synthesis approaches, and is compatible with the data-driven style that our tool employs. The approach is parameterized by a data-driven program synthesizer (for candidate lemma generation), counterexample searcher (for candidate filtering), and proof searcher (for candidate ranking). The implementation uses the MYTH [OZ15] data-driven program synthesizer for OCaml, the QUICKCHICK [PHD15] tool for counterexample search, and the state-of-the-art PROVERBOT9001[SAS20b] tool for proof script search. Note that the approach is *agnostic* to the specific toolset we use for implementation; in fact, future improvements in data-driven program synthesis, counterexample search, and proof search can be directly leveraged to improve lemma synthesis.

I evaluate my approach on two benchmark suites from prior work on lemma synthesis, CLAM [IB96] and LIA [YFG19], as well as two new benchmarks from diverse domains, FULL ADDER [cir95] and compiler correctness [Chl13b]. Together, there are 222 evaluation locations from these benchmarks, where a human prover used an auxiliary lemma to progress. `lfind` synthesizes a useful lemma for 144/222 of these locations, with a median runtime of 4.8 minutes (see § 2.5.3). At 109 of these locations `lfind` provides a full automated proof of the synthesized lemma and the goal; at the other 35 locations `lfind` produces a ranked list of lemma candidates where the human-written lemma is in the top 10. I also show that our approach significantly outperforms the prior technique of generalization as well as a version

of `lfind` that employs type-guided synthesis without examples (§ 2.5.4). Finally, in § 2.5.5 I evaluate `lfind`'s sensitivity to different hyperparameters and timeouts.

In summary, I make the following contributions:

1. I present the first approach that reduces the general lemma synthesis problem to a data-driven program synthesis problem. The approach derives both lemma sketches as well as examples for synthesis from a given stuck proof state, and it uses the existing generalization technique to lift the proof state to new variables for synthesis.

2. I describe a suite of filtering and ranking strategies for candidate lemmas, which are necessary for an interactive verification setting.

3. I have instantiated the approach in a tactic called `lfind` for Coq.

4. The experimental evaluation demonstrates the practical utility of the approach and tool, quantifies the benefits over multiple alternative approaches to lemma synthesis, and investigates the sensitivity of `lfind` to different parameter values.

2.2 Overview

2.2.1 Motivating Example

To illustrate how `lfind` works, we'll start with an example. Figure 2.1 shows Coq code that tries to prove a simple theorem: that reversing a list twice returns the same list. It starts by defining lists of `nats` along with definitions for appending and reversing lists. Following that is an attempt to prove the theorem, named `rev_rev`.

The proof proceeds by induction on the list `l`. The `Nil` case is easily proven, but the `Cons` case is trickier. After simplification, the user is stuck because the goal is not in a form that enables direct use of the induction hypothesis. Figure 2.2 shows the proof state at that point, including the current assumptions and goal.

```
1  Inductive lst : Type :=
2    | Nil : lst
3    | Cons : nat -> lst -> lst.

5  Fixpoint app (l1 : lst) (l2 : lst) : lst :=
6    match l1 with
7    | Nil => l2
8    | Cons n l1' =>  Cons n (app l1' l2)
9    end.

11 Fixpoint rev (l : lst) : lst :=
12   match l with
13   | Nil => Nil
14   | Cons n l1' => app (rev l1') (Cons n Nil)
15   end.

17 Lemma rev_rev : forall l, rev (rev l) = l.
18 Proof.
19   induction l.
20   - reflexivity.
21   - simpl. (* I'm stuck! *)
```

Figure 2.1: A partial proof of a theorem in Coq that requires an auxiliary lemma.

```
1 n : nat
2 l : lst
3 IHl : rev (rev l) = l
4 ---------------------------------------------
5 rev (app (rev l) (Cons n Nil)) = Cons n l
```

Figure 2.2: The proof state when the user gets stuck.

To get unstuck, the user can invoke our tool `lfind` as a tactic at this point. In this example, the top three lemmas that `lfind` produces are as follows:

```
1 (Λ1) Lemma lem1: forall l n,
2        rev (app l (Cons n Nil)) = Cons n (rev l).
3 (Λ2) Lemma lem2: forall l1 l2,
4        rev (app l1 l2) = app (rev l1) (rev l2).
5 (Λ2) Lemma lem3: forall l1 l2,
6        rev (app (rev l1) l2) = app (rev l2) l1.
```

```
Lemma lem1: forall l n,
  rev (app l (Cons n Nil)) = Cons n (rev l).
Proof.
    intros.
    induction l.
    simpl.
    eauto.
    simpl.
    rewrite IHl.
    eauto.
Qed.

Lemma rev_rev : forall l, rev (rev l) = l.
Proof.
  induction l.
  - reflexivity.
  - simpl. rewrite <- IHl. unfold app.
    rewrite IHl. rewrite lem1. rewrite IHl. easy.
Qed.
```

Figure 2.3: A full proof provided by `lfind`.

Each lemma is bucketed into one of three categories (Λ_1, Λ_2, or Λ_3), and the categories are presented to the user in that order. Λ_1 lemmas are those in which `lfind` can automatically find a complete proof of the original goal using the generated lemma and PROVERBOT9001, a state-of-the-art automated prover. In other words, `lfind` has successfully generated an appropriate auxiliary lemma, proven that lemma, and used the lemma to complete the original proof. The lemma `lem1` is such an Λ_1 lemma; the full proof of the theorem `rev_rev` using `lem1` is shown in Figure 2.3.

Λ_2 lemmas are those that are sufficient to automatically prove the original goal, not disprovable by QUICKCHICK, but PROVERBOT9001 cannot automatically prove the auxiliary lemma. `lfind` indicates that the second and third lemmas in the above listing are Λ_2 lemmas; indeed, each of them in turn depends on its own auxiliary lemmas, for example, the associativity of `app`. However, both lemmas are also still good options for the user: the lemma `lem2` is a more general version of `lem1`, while lemma `lem3` reduces to the original

`rev_rev` lemma when `l2` is `Nil`. Λ_3 lemmas are ones that are not disprovable by a tester like QUICKCHICK but automation using PROVERBOT9001 can't be used to prove either the goal or the auxiliary lemma; since they are similar to the goal and not disprovable, they might still be useful to the user.

In the rest of this section, I explain how `lfind` produces these results.

2.2.2 Approach

As mentioned in § 2.1, the generality of our setting induces several technical challenges. Lemma synthesis in `lfind` has four steps that are targeted at these challenges, as shown in Figure 2.4. We start by **generalizing** the goal state, in order to systematically introduce new variables that can be used in candidate lemmas. From each generalization, we create sketches and sample variable valuations from the current goal in order to reduce lemma synthesis to **data-driven program synthesis**. Finally, we **filter** the resulting lemma candidates to remove those that cannot be useful and **rank** and categorize the remaining candidates for user inspection.

Generalization In Coq, helper lemmas are generally used as arguments to the `apply` and `rewrite` tactics. To use the `apply` tactic, the consequent of the lemma has to structurally match the goal state to which it is applied. Similarly, if you are using the `rewrite` tactic, the lemma needs to be a setoid relation, one of whose arguments structurally match a portion of the goal state. It is for these reasons that prior techniques for lemma synthesis in the interactive setting [KM97, BSV93] work by making heuristic rewrites to the goal state.

Our approach starts from the same intuition and we observe that generalization provides a systematic way to introduce new variables to the synthesis process. Since we are not sure in advance how many and which variables a useful lemma might need, we exhaustively generate generalizations. Therefore, we start by producing all *generalizations* of the goal at the point where `lfind` is invoked, which are formed by replacing one or more terms within the goal

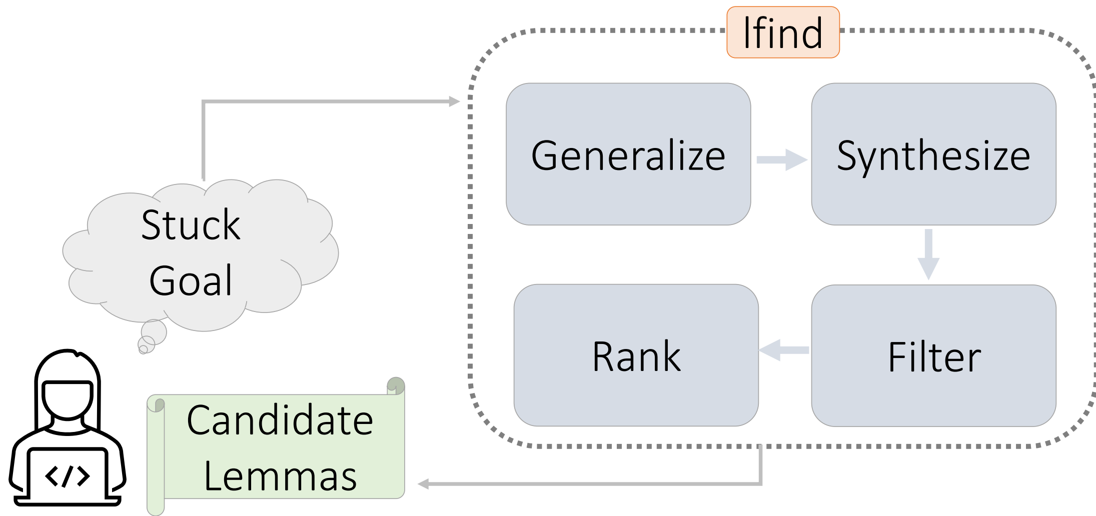

Figure 2.4: Overview of `lfind`.

with fresh variables [KM97]. In our example, there are six non-variable terms in the goal (Figure 2.2). While in principle there are 2^6 possible generalizations using these terms, there are only 16 unique ones, since some terms are only present as subterms of other ones.

For example, replacing `rev l` with a fresh variable `l1` of type `lst` produces the following generalization:

```
forall l n l1, rev (app l1 (Cons n Nil)) = Cons n l.
```

Alone, this generalization does not produce a valid lemma, as it does not hold when `l1` is not the reverse of `l`. Typically generalization is only applied on terms that appear more than once in a goal [KM97], to avoid these cases. In our example, there are no such terms, and in fact, all lemmas generated by generalization alone are easily disprovable.

Nonetheless, these generalizations play a crucial role in our approach. In addition to being treated as candidate lemmas themselves, we use each generalization as a starting point from which to produce many more candidate lemmas via data-driven program synthesis. Each generalization introduces new variables that can be leveraged as part of that synthesis process.

Synthesis From each generalization, we create a set of *sketches*, where each sketch is a version of that generalization with one term replaced by a *hole*. Note that synthesis is much more expensive than the generalization process described above which simply replaces some terms with variables. Furthermore, the need for lemmas to structurally match the goal state limits how many parts of that state can be usefully rewritten. For these reasons, we consider only one hole per lemma sketch, but `lfind`'s algorithm conceptually does not limit the number of holes per synthesis. Our technique can be extended to synthesize terms for each hole one at a time and then induce candidate lemmas from their combinations.

For example, if we replace the term `Cons n l` in the generalization above with a hole, then we end up with the following sketch (note that we remove variable `l` from the quantifier since it is no longer used):

```
forall l1 n, rev (app l1 (Cons n Nil)) = □.
```

Intuitively, we would like the expression that fills the hole to behave consistently with the expression that it is replacing. To that end, we generate concrete examples of the original goal in the stuck state and then map them to input-output examples for data-driven synthesis. In our running example, the original goal has two variables, `l` and `n`, so suppose we randomly generate the following `(l, n)` pairs (using regular list notation for clarity):

{`([], 4)`, `([0; 1], 2)`, `([2; 1], 1)`}.

Next, we map these examples to our sketch. We do so by leveraging the fact that the new variable `l1` replaced `rev l` from the original goal. Hence we evaluate `rev l` for each of our three examples to produce the following `l1` values: {`[]`, `[1; 0]`, `[1; 2]`}. Similarly, we produce the expected value of the hole for each example, by leveraging the fact that the hole replaced `Cons n l`. This yields the values {`[4]`, `[2; 0; 1]`, `[1; 2; 1]`}.

As a result of this mapping, we can now produce a set of input-output examples that act as a specification for synthesis, each mapping `(l1, n)` pairs to the expected output value of the term to be synthesized:

```
([], 4) ↦ [4]
([1; 0], 2) ↦ [2; 0; 1]
([1; 2], 1) ↦ [1; 2; 1]
```

Finally, we pass these input-output examples to a data-driven synthesizer. In addition to the examples, we provide the type of the function to be synthesized (which in this case is `lst * nat → lst`) and a grammar to use for term generation. `lfind` automatically creates a grammar consisting of the definitions that appear in the stuck proof state along with definitions that they recursively depend upon. In our example the grammar includes the constructors `Nil` and `Cons` and the functions `app` and `rev`. One term that the synthesizer generates from these inputs is `Cons n (rev l1)`. Substituting this expression into the hole in our sketch yields exactly the lemma `lem1` shown earlier, which enables a fully automated proof of the original lemma.

In summary, I have shown how to generate candidate lemmas in a targeted way, based on the current proof state, using a novel combination of generalization and data-driven program synthesis. While the expressions that are generated by synthesis can make use of a general grammar, the form of the lemmas that we generate are still limited to the structure of the sketches that we produce. As I demonstrate in §2.5, the approach can generate useful lemmas for a variety of interesting benchmarks.

Filtering As described above, the approach induces many generalizations of each goal, multiple sketches for each generalization, and multiple synthesis results for filling each sketch's hole. Hence, the set of candidate lemmas that are generated is quite large. In our running example, with default settings for the number of sketches to produce for each generalization and the number of synthesis results to produce for each sketch (see §2.5.2), `lfind` generates 276 candidate lemmas. While the ability to explore a large space of candidates is a strength of the approach, we must organize these candidates in a manner that is understandable and beneficial to users.

To that end, we filter out extraneous candidates in multiple ways.

First, we filter out candidates for which we can find a counterexample; we search for counterexamples using QuickChick, an existing counterexample-generating tool [PHD15]. Second, we filter out candidates representing trivial facts, for example `forall l, rev l = rev l`. We identify such cases using Coq's `trivial` tactic.

Finally, we filter out candidates that "follow directly" from the user's original lemma, a notion we explore in more detail in § 2.3.4. For instance, in our running example one candidate lemma is `forall n l, rev (rev (Cons n l)) = Cons n l`, which is a special case of the original `rev_rev` lemma and hence is discarded in this step.

Ranking After filtering, there are 21 candidate lemmas remaining in our running example. While that constitutes a 92.4% reduction, it is still too many candidates to require the user to examine. Hence, we rank candidates based on their likely utility to the user and present them in ranked order. Since ultimately the utility of a lemma is based on whether it allows the user to prove the original goal, our ranking leverages a state-of-the-art automatic prover for Coq, Proverbot9001, which searches the space of Coq tactics to try to prove a given goal [SAS20b].

Specifically, we use the automatic prover to partition the candidate lemmas into the three groups introduced in §2.2.1: Λ_1 lemmas that are automatically provable and enable automatic proof of the user's stuck proof state; Λ_2 lemmas that are not automatically provable but enable automatic proof of the user's stuck proof state; and the remaining Λ_3 lemmas. Next, we sort each group in order of size from least to greatest, since we expect smaller lemmas to be easier for users to understand and evaluate. Finally, we concatenate these sorted groups to form the final ranked list.

In our running example, there are 2, 2, and 17 lemmas respectively in each of these three categories. The first lemma in category Λ_1, which yields a fully automated proof, is

`lem1` shown earlier, so it is ranked first. Lemmas `lem2` and `lem3` are the smallest lemmas in category Λ_2 and hence are ranked next in our results.

2.3 Algorithms

In this section, I formally describe the core algorithms that make up the approach.

2.3.1 Preliminaries

The approach synthesizes lemmas for a given proof state Ψ, which is a tuple $\langle \mathcal{H}, g, \Gamma, \mathcal{D} \rangle$, where $\mathcal{H}$ is a set of logical formulas that are the current *hypotheses*, g is a logical formula that is the current *goal*, Γ is a type environment for all free variables in $\mathcal{H}$ and g, and $\mathcal{D}$ is a set of type and term definitions that are recursively referred to in $\mathcal{H}$ and g. It is required that the goal g be unquantified, which in practice typically means that the original lemma/theorem should have all variables universally quantified at the front.

I use ϕ to denote logical formulas, x for variables, v for values, t for terms of sort `Type`, and τ for the types of terms. A *s*ample for a proof state $\Psi = \langle \mathcal{H}, g, \langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle, \mathcal{D} \rangle$ is an environment $e = \langle x_1 : v_1, \ldots, x_n : v_n \rangle$ such that e is a model of $\mathcal{H} \to g$, denoted $e \vDash_{\mathcal{D}} \mathcal{H} \to g$. I also use the notation $t \Downarrow_e v$ to denote the evaluation of term t to value v under environment e.

Finally, I assume the existence of several black-box functions that have been created by others in prior work. I assume the availability of a black-box synthesizer that takes as input a grammar $\mathscr{G}$, consisting of typed constants and functions; a type signature $\tau_1 \to \tau_2$; and input-output examples of the form (v_1, v_2), where v_1 has type τ_1 and v_2 has type τ_2. This synthesizer returns a list of functions f of type $\tau_1 \to \tau_2$ in the grammar $\mathscr{G}$ such that $f(v_1) = v_2$ for all examples; or it fails after some time limit. I also assume the existence of a function $\textsc{Sample}(\Psi)$ that produces a set of samples. Last, I assume the existence of both automated theorem provers and disprovers. A *prover* $\mathcal{R}(\phi, \bar{\phi}, \mathcal{D})$ attempts to prove a given

formula in the context of a set of auxiliary lemmas $\bar{\phi}$ as well as a set of definitions, returning either VALID or DONT KNOW. A *disprover* $\mathcal{C}(\phi, \mathcal{D})$ searches for concrete counterexamples to ϕ and returns either INVALID or DON'T KNOW.

2.3.2 Lemma Synthesis

First, I describe how we reduce lemma synthesis to data-driven program synthesis. As described in the previous section, the first step is to produce *generalizations* of the current goal g, by replacing some set of terms in g with fresh variables. The following definition formalizes this notion of generalization.

Definition 2.1. (Generalization: $\mathcal{G}$) Given a goal g, a type environment Γ, and a set $\mathbf{T} = \{t_1, \ldots, t_n\}$, we define the *generalization* of g with respect to Γ and $\mathbf{T}$, denoted $\mathcal{G}(g, \Gamma, \mathbf{T})$, as the tuple $\langle g', \Sigma \rangle$, where $\Sigma = \langle x_1 \mapsto t_1, ..., x_n \mapsto t_n \rangle$ records the mapping from each new variable to the term that it replaces, variables $x_1, \ldots, x_n$ are not in the domain of Γ, and $g' = g[\bar{t_i} \mapsto \bar{x_i}]$.

`lfind` uses the generalizations that it constructs as candidate lemmas. In addition, generalizations are used as the basis for creating *sketches* for data-driven synthesis. Each sketch is simply a version of a generalized goal that has one term replaced by a *hole*, denoted $\square$.

Definition 2.2. (Sketch: $\mathcal{S}$) A *sketch* of goal g with respect to term t, denoted $\mathcal{S}(g, t)$, is $g[t \mapsto \square]$.

In order to produce a data-driven program synthesis problem, we must generate input-output examples. The following definition shows how I extend an environment to an input-output example, given a set of terms (which are used for generalization), and a term (used for creating a sketch). Intuitively, the new variables created by generalization become additional input variables, and the term used to create a sketch defines the expected output.

Definition 2.3. (Input-output example: IO) The input-output example corresponding to a given environment $e = \langle x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \rangle$, term mapping $\Sigma = \langle x'_1 \mapsto t_1, ..., x'_m \mapsto t_m \rangle$, and term t_s, denoted $\text{IO}(e, \Sigma, t_s)$, is defined as

$$\langle \langle x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, x'_1 \mapsto v'_1, ..., x'_m \mapsto v'_m \rangle, v_s \rangle$$

where $t_i \Downarrow_e v'_i$ for each t_i in $t_1, \ldots, t_m$ and $t_s \Downarrow_e v_s$.

Finally, we can put all of this together to specify how to reduce lemma synthesis to data-driven program synthesis.

Definition 2.4. (Lemma synthesis as data-driven program synthesis) Given a proof state $\Psi = \langle \mathcal{H}, g, \Gamma, \mathcal{D} \rangle$, a set of terms $\mathbf{T} = \{t_1, \ldots, t_m\}$ for generalization, and a sketch term t_s, we produce a data-driven program synthesis problem as follows. Let $\mathcal{G}(g, \Gamma, \mathbf{T}) = \langle g', \Sigma \rangle$, where $\Sigma = \langle x'_1 \mapsto t_1, ..., x'_m \mapsto t_n \rangle$. Let $\mathcal{S}(g', t_s) = g_s$.

- The grammar $\mathscr{G}$ for synthesis is defined by the type and term definitions in $\mathcal{D}$.
- Let Γ_s be Γ restricted to the variables that appear free in g_s. The input variables and associated types for the function to be synthesized are $\Gamma_s \texttt{@} \langle x'_1 : \tau_1, ..., x'_m : \tau_n \rangle$, where $\Gamma \vdash t_i : \tau_i$ for each $t_1, \ldots, t_m$.
- The output type for the function to be synthesized is τ_s, where $\Gamma \vdash t_s : \tau_s$.
- The input-output examples for synthesis are produced as follows. First we generate a set of samples $\textsc{Sample}(\Psi) = \langle e_1, \ldots, e_p \rangle$ for the given proof state. Then the set of input-output examples is $\langle \text{IO}(e_1, \Sigma, t_s), \ldots, \text{IO}(e_p, \Sigma, t_s) \rangle$.

We invoke the synthesizer with these inputs and ask for up to k functions (§ 2.5 reports sensitivity analysis for k) that meet this specification. For each such function f, with body expression t_f, the induced *candidate lemma* is created by universally quantifying all free variables in the term $g_s[\Box \mapsto t_f]$.

So far, I have formalized the process of lemma candidate generation from a single set of terms to be generalized and a single term to be used for creating a sketch. `lfind` performs this process many times, for many different generalizations and many different sketch terms. Various approaches to exploring this space are possible. `lfind` *exhaustively* explores the generalization space, producing one generalization for each subset of terms in the goal g. For each such generalization, `lfind` employs terms that have sort `Type` for creating sketches. There are several ways to pick a synthesis term for a sketch, and in § 2.5 I carry out sensitivity analysis for two natural approaches to choosing such terms.

2.3.3 Filtering

The approach described so far generates *a lot* of candidate lemmas. If there are t subterms in a given goal to use for generalization, m sketches per generalization, and we ask the synthesizer for k results, then without any filtering `lfind` would produce a maximum of $2^{t+1}mk$ candidates, including all generalizations and the lemmas derived from them using data-driven synthesis. Exploring a large space of candidates is advantageous, but clearly, we require techniques to filter out candidates that are not going to help the user.

I employ four different filtering techniques. First, it's common for there to be many duplicate lemmas. For example, it's possible for synthesis from two different sketches to produce the same result. It's also possible for synthesis from a single sketch to produce syntactically distinct results that are behaviorally equivalent. We identify and filter duplicates by applying Coq's `simpl` tactic and then comparing the results for syntactic equivalence. Second, we use the disprover $\mathcal{C}$ to search for counterexamples, filtering out any candidate ϕ such that $\mathcal{C}(\phi, \mathcal{D}) = \textsc{Invalid}$. Third, we remove lemmas that can be solved using Coq's `trivial` tactic, since they are self-evident and hence never needed as explicit auxiliary lemmas.

Finally, we filter lemmas that "follow directly" from the original lemma, as they will not help in proving that lemma. This is a subtle notion. For example, it is not a form of logical

implication, since if the candidate lemma is valid then *any* other lemma implies it. Instead, we formalize this filter via a binary relation $\preceq$, which says that one lemma is an instantiation of (or equivalent to) another, defined as follows:

Definition 2.5. ($\preceq$-operator) Given lemmas $l1$ and $l2$, we say $l1 \preceq l2$ if we can prove $l1$ using either of the following proof scripts:

```
1  intros. apply l2. Qed.
2  intros. rewrite <- l2. reflexivity. Qed.
3  intros. rewrite -> l2. reflexivity. Qed.
```

We then filter out any candidate lemma that is $\preceq$ than the original lemma.

2.3.4 Ranking

We rank the remaining candidate lemmas using the automated prover $\mathcal{R}$ we introduced earlier. For each candidate ϕ we use the prover to determine if the candidate can be used to automatically prove the goal g — $\mathcal{R}(g, \{\mathcal{H}, \phi\}, \mathcal{D})$ — and whether the candidate itself is automatically provable — $\mathcal{R}(\phi, \emptyset, \mathcal{D})$. Based on the results we partition the lemmas into three groups, Λ_1, Λ_2, and Λ_3. The Λ_1 group contains the lemmas for which both calls to $\mathcal{R}$ return VALID, meaning that we have obtained a fully automated proof of the user's original goal. The Λ_2 group contains the lemmas for which the first call to $\mathcal{R}$ return VALID, meaning that the lemma enables the goal to be automatically proven but the lemma is not itself automatically provable. The remaining lemmas go in the Λ_3 group. We sort the lemmas in each group in order of size from smallest to largest, since we expect smaller lemmas to be easier for users to understand and evaluate. Finally, we concatenate these sorted groups to form the ranked list.

2.3.5 Discussion

Note that `lfind`'s approach to candidate lemma generation imposes some important restrictions on its usage. I have already mentioned that the goal in the proof state must be

unquantified. Further, the approach relies on the ability to generate examples for the stuck state, which limits it to the capabilities of current test generation techniques. Because I reduce lemma synthesis to program synthesis I require the ability to extract necessary definitions as code and translate code back to Coq. Finally, because sketches for synthesis are derived from a generalization of the original goal, the generated lemmas will always have the same top-level structure as the goal. For example, if the original goal has the form `A = B` then the candidate lemmas will also have this form. §2.5.3 shows that despite these limitations, `lfind` can successfully identify non-trivial helper lemmas for a variety of examples. In addition, all of these limitations represent useful avenues of investigation in future work.

2.4 Implementation

Figure 2.5 illustrates the overall architecture of `lfind`, which leverages three black-box components: a data-driven synthesizer for candidate lemma generation; an automatic disprover for candidate filtering; and an automatic prover for candidate ranking. The implementation of `lfind` is 3.2 KLOC of OCaml code.

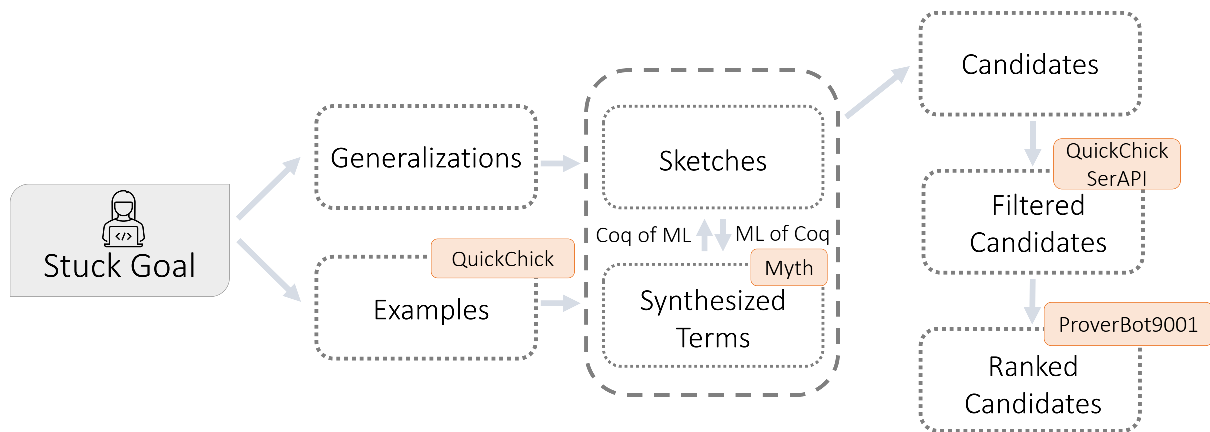

Figure 2.5: Given a stuck goal, `lfind` implements generalization, synthesis, filtering, and ranking in conjunction with existing tools to generate candidate lemmas.

2.4.1 Example Generation

To synthesize candidate lemmas, the approach relies on a SAMPLE function that can produce samples for the variables in the stuck state g. I leverage QUICKCHICK [PHD15], a state-of-the-art property-based randomized tester for Coq, for this purpose. While QUICKCHICK is intended as a testing tool, we log all of the test inputs that it generates and use them as the samples from which to produce examples for synthesis.

Specifically, for each user-defined type `T` in the stuck goal, `lfind` first generates the following Coq code, which enables the usage of QUICKCHICK for that type:

```
1  Derive Show for T.
2  Derive Arbitrary for T.
3  Instance Dec_Eq_T : Dec_Eq T.
4      Proof. dec_eq. Qed.
```

The `Show` typeclass is required for printing test cases and the `Arbitrary` typeclass is required to combine test-case generation with an operation for shrinking test inputs. QUICKCHICK supports automatic derivation of instances of these type classes for simple types. QUICKCHICK also requires that types have decidable equality, so we derive an instance of the `Dec_Eq` typeclass for `T`.

Next, to produce examples for the stuck proof state g, we create a Coq lemma for that state, defined as `Lemma stuckState:` $\mathcal{H} \rightarrow g$. We also create a function `collect_data` whose input type `V` is the tuple of the types of all free variables in $\mathcal{H} \rightarrow g$. The function logs the input values to a file and returns the valuation of $\mathcal{H} \rightarrow g$ on those input values:

```
1  Definition collect_data  (n:V) :=
2   in let _ := print_to_file (show n)
3   in stuckState n.
4  QuickChick collect_data.
```

Finally, we run QUICKCHICK on this function, thereby logging samples to use for data-driven synthesis and also searching for counterexamples to the stuck proof state. If QUICKCHICK returns any such counterexamples, then there is no way to complete the proof so we report this to the user and halt `lfind`. Otherwise we proceed with synthesis.

2.4.2 Synthesis

To my knowledge, there are no data-driven synthesizers that work directly for Coq. So I chose MYTH [OZ15] as the synthesizer because it accepts and generates OCaml code, for which tools exist to convert to/from Coq's language Gallina; it has a simple interface that is easy to use; and it has worked well for us in the past. MYTH requires an input grammar in OCaml, so I use Coq's `Extraction` feature to recursively extract reachable definitions and types from the stuck goal to OCaml. Additionally, I adapt MYTH slightly in two ways. First, MYTH supports only a subset of OCaml and does not support common syntactic sugars. For example, MYTH does not support the `function` keyword. To get around these limitations, I wrote a translator that desugars the definitions extracted from Coq into a form acceptable by MYTH. Second, I modified MYTH to return a set of candidate functions sorted by size, instead of just one. This enables the generation of multiple candidate lemmas as described earlier. Finally, to substitute the synthesized OCaml function body back into the lemma sketch, I use an open-source tool, `coq-of-ocaml` [coq03].

2.4.3 Filtering and Ranking

In § 2.3.3, I described multiple filters to remove extraneous candidate lemmas. To implement these filters, I declare each candidate as a Coq lemma and use QUICKCHICK to remove lemmas that have counterexamples. The remaining filters are implemented by running proof tactics using `SerAPI` [GPP20], a library for machine-to-machine interaction with Coq. To rank the filtered lemmas, I use PROVERBOT9001 [SAS20b], a state-of-the-art proof-synthesis tool that uses machine learning to produce proofs of Coq theorems. PROVERBOT9001 takes as input definitions, a theorem that needs to be proven, and a set of axioms that can be assumed, and returns a proof script or DON'T KNOW.

2.4.4 Discussion

In the implementation, I try to disprove each generalization eagerly and carry out synthesis from generalizations for which the disprover finds a counterexample. Intuitively, if a generalization is not disprovable then it is itself a candidate lemma, and so we would rather spend our synthesis resources elsewhere. Candidate lemmas are produced incrementally, as generalization and synthesis proceed. Hence the algorithm is *any-time*: we can stop at any point, collect up the current set of candidates, and filter and rank them. Furthermore, I stop synthesis as soon as we get a category Λ_1 lemma since we will have a fully automated proof of the user's original goal.

My implementation inherits the limitations of the black-box tools we rely on. Notably, Myth only supports a small subset of OCaml. As described above, I mitigate this limitation by implementing a translator, but this is not a solution that works for the full OCaml language, and so in some cases `lfind` can fail to produce code that Myth accepts. Myth also does not support polymorphic types.

2.5 Experimental Results

In this section I perform experiments to answer the following research questions:

RQ1. (§ 2.5.3) How effective is `lfind` in synthesizing useful helper lemmas? How fast can the tool synthesize these helper lemmas? What is the impact of its filtering and ranking techniques?

RQ2. (§ 2.5.4) How does `lfind`'s data-driven approach compare in effectiveness to prior approaches to lemma synthesis?

RQ3. (§ 2.5.5) How sensitive is `lfind` to hyperparameters and timeouts?

2.5.1 Benchmark Suite

The approach generates candidate helper lemmas from a given proof context, and the tool is implemented as a *tactic*. Hence, to evaluate `lfind` we need to invoke `lfind` at each point in the proofs where a user-provided helper lemma was used. These are called *evaluation locations*. Concretely, a proof state is an *evaluation location* if a human prover has used either the `apply` or `rewrite` tactics with a helper lemma that they created. We evaluate `lfind` on a total of 222 evaluation locations. These benchmark locations are drawn from the following sources.

- CLAM (140): This benchmark suite consists of 86 theorems about natural numbers as well as various data structures, including lists, queues, and trees, and it has been used to evaluate prior forms of lemma synthesis [IB96, YFG19]. These benchmarks lack associated proofs, so we converted them to Coq and manually proved each theorem (more details on this process below). Out of the 86 CLAM theorems, 67 require at least one helper lemma, with many requiring multiple lemmas. In total, the CLAM suite contains 184 unique evaluation locations that employ a helper lemma. Implementation limitations mentioned in § 2.4.4 preclude 44 locations from CLAM from being used for evaluation, leaving 140 remaining evaluation locations.
- Full Adder (62): This project [cir95] from the `coq-contribs` collection formalizes a full adder and proves it correct [cir95]. The program first builds a half-adder circuit (which takes two binary digits, and outputs two binary digits) and proves properties about it. Then the half-adder circuit is used to build a full-adder circuit (which takes two binary digits, plus a "carry" digit, and outputs two binary digits). Finally, the program chains together a sequence of full adders to create an adder circuit, which is proven correct. All of the 40 theorems in this project require at least one helper lemma, and the project contains 62 evaluation locations in total.

Table 2.1: Results

	CLAM	FULL ADDER	COMPILER	LIA
Setup				
# Theorems	86	40	1	9
# Evaluation Locations	140	62	1	19
Results				
# fully proven lemma and goal	68	34	0	7
# else human match in top 1	14	0	1	0
# else human match in top 5	9	1	0	3
# else human match in top 10	6	0	0	0
# else more general than human lemma in top 1	1	0	0	0
Summary	98/140	35/62	1/1	10/19

- COMPILER (1): This benchmark is the compiler example from Chapter 2 of Chlipala's CPDT textbook [Chl13b], which is a certified compiler from a source language of expressions to a target language of a stack machine. The final theorem formalizes the correctness of the compiler. This benchmark contains one theorem, which uses one helper lemma, which is the evaluation location. Though it contains only a single evaluation location, I chose this example as a benchmark because it showcases a different application and the required helper lemma is relatively large and complex.

- LIA (19): This benchmark suite consists of 9 theorems about data structures that require linear integer arithmetic, from a prior work on lemma synthesis for fully automated proofs about data structures (see Table 1 in [YFG19]). As with the CLAM benchmarks, I converted them to Coq and manually proved each theorem. Each proof requires at least one helper lemma, and there are a total of 19 evaluation locations.

The FULL ADDER and COMPILER benchmark suites already contain full Coq proofs written by others, which in turn determine our evaluation locations. The theorems in the CLAM and LIA benchmark suites lack proofs, so each theorem was manually proven by one of three of us, with varying experience from novice to expert in interactive theorem proving. Specifically, one person had only done a small class project with Coq previously, one has been using Coq for the past few years on a research project, and one has used it on and off

for a decade. The proofs were completed independently of `lfind`'s evaluation, and helper lemmas were used wherever the human prover deemed necessary. In § 2.5.4, I show that the vast majority of these helper lemmas are indeed necessary, in the sense that a state-of-the-art automated prover cannot complete the proof of the theorem without a helper lemma.

2.5.2 Experimental Setup

For each evaluation location, `lfind` generates 50 input-output examples from the current proof state and is allowed to generate candidate lemmas with a maximum timeout of 100 minutes. Despite the large search space, in § 2.5.3 I show that the tool is performant with a median runtime of only 4.8 min. The tool has a 12s timeout for each call to Myth and a 15s timeout for each call to Proverbot9001. In addition to the timeout parameters, two key hyperparameters to the algorithm are the choice of subterms to use for generating sketches and the number of synthesis results k to obtain per sketch. In the experiments, I generate sketches from *all subterms* of sort `Type`, and ask for *5* synthesis terms per sketch. Empirically I have found these choices to provide good results, but I also present a sensitivity analysis of other choices for timeout and hyperparameters in § 2.5.5.

All evaluations were performed on a machine that runs MacOS (10.15.6) in a 2.3 GHz-Quad-Core Intel Core i7, with 32GB memory.

2.5.3 Synthesized Helper Lemmas

Table 2.1 summarizes the results for all the benchmarks. I consider the use of `lfind` at an evaluation location to be successful in three scenarios. First, `lfind` is successful if it can produce a candidate helper lemma that is automatically proven by Proverbot9001 and this helper lemma enables Proverbot9001 to automatically prove the user's goal. This is the best-case scenario, as `lfind` has produced a complete proof for the user. Second, `lfind` is successful if a lemma that matches the human-provided lemma is ranked highly (top-10) by

the tool. Third, `lfind` is successful if a lemma that is more general than the human-provided lemma is ranked highly by the tool. I use the $\preceq$ operator defined in § 2.3.3 to automatically identify if a candidate lemma l matches or is more general than the human-provided lemma h. Specifically we say that l matches h if both $l \preceq h$ and $h \preceq l$, and we say that l is more general than h if $h \preceq l$ but not vice versa. These are reasonable success metrics for the tool, as I expect versions of the human-provided lemma to be "natural" for people to understand, and also we know that the human-provided lemma does indeed lead to a full proof of the goal. Note however that the metrics are conservative, as there could be other lemmas produced by `lfind` that are natural and appropriate but do not fall into one of the above three categories.

In total, based on the evaluation metrics we see that `lfind` succeeds in 144 (64.9%) of the 222 evaluation locations across all benchmarks. Further, as shown in the third row of the table, in 109 (75.7%) of these successful 144 locations, `lfind` was able to synthesize a lemma that led to a fully automated proof of the user's goal. Rows 4-7 of Table 2.1 show a breakdown of the remaining 35 successful locations. Notably, for 15 of these evaluation locations, the top-ranked candidate lemma produced by `lfind` matches the helper lemma provided by the human prover. These results demonstrate the effectiveness of the filtering and ranking strategies in surfacing relevant lemmas toward the top, and often as *the* top result.

Examples. Table 2.2 shows examples of lemmas synthesized by `lfind` along with their rank and category (see § 2.3.4 for category notations). I describe the first four of them in detail.

The first example from the `Compiler` benchmark formalizes the correctness of a compiler from a source language of expressions to a target language of a stack machine. In this case, type `exp` defines the source language of arithmetic expressions. `evalExp` function evaluates the programs in this language. The target language's instructions are of type `instr`, which are executed on a stack machine. The function `execI` takes an instruction and a stack

Table 2.2: A sample of **lfind** synthesized lemmas and their associated rank and category.

#	Original Theorem	**lfind** Synthesized Lemma	Λ
1.	Theorem correct_compilation:∀ (e :exp), execIs (compile e) Nil = (evalExp e) :: Nil.	Lemma lem1:∀ (e: exp) (l: list instr) (s: list nat), execIs (compile e ++ l) s = execIs l (evalExp e :: s).	Λ_2
2.	Theorem BV_full_adder_nil_ok :∀ (v:BoolList) (cin:bool), BV_to_nat (BV_full_adder v Nil cin) = BV_to_nat v + Bool_to_nat cin.	Lemma lem1:∀ (l:BoolList), BV_to_nat (BV_full_adder_sum l Nil false ++ BV_full_adder_carry l Nil false::Nil) = BV_to_nat l.	Λ_1
3.	Theorem app_revflat:∀ (x:tree) (y:list nat), (revflat x) ++ y = qrevaflat x y.	Lemma lem10:∀ (l l1 l2:list nat), (l ++ l1) ++ l2 = l ++ (l1 ++ l2).	Λ_3
4.	Theorem queue_push:∀ (q:queue) (n:nat), qlen (qpush q n) = 1 + (qlen q).	Lemma lem1:∀ (l l1:list nat), len l + len l1 = len (l ++ rev l1).	Λ_2
5.	Theorem qreva_qreva:∀ (x:list nat), (qreva (qreva x Nil) Nil) = x.	Lemma lem9:∀ (n:nat) (l:list nat), qreva (append l n::Nil) Nil = n::(qreva l Nil).	Λ_2
6.	Theorem rotate_len:∀ (x:list nat), rotate (len x) x = x.	Lemma lem2:∀ (l l1:list nat), rotate (len l) l ++ l1 = l1 ++ l.	Λ_2
7.	Theorem add_even:∀ (x y:nat), even(x+y) = even(y+x).	Lemma lem1:∀ (n x:nat), negb (even(n+x)) = even(n+(S x)).	Λ_1
8.	Theorem drop_elem:∀ (v w x y:nat) (z:list nat), drop (S v) (drop w (drop x y::z)) = drop v (drop w (drop x z)).	Lemma lem1:∀ (n x:nat) (l:list nat), drop (S x) (drop n l) = drop x (drop (S n) l)).	Λ_1

(represented as a list of `nats` and returns an updated stack, and `execIs` uses this function to execute a list of instructions. Finally, the `compiler` function translates source programs to a list of instructions. The theorem itself is not inductive, necessitating an inductive helper lemma that implies the theorem [Chl13b]. `lfind` was not able to identify a helper lemma that leads to a fully automated proof of the theorem. However, it produces candidate lemmas in categories Λ_2 and Λ_3, and the top-ranked candidate in category Λ_2, shown in the table, *exactly matches* the human-provided lemma. The lemma is non-trivial as it involves multiple calls to `execIs`, an arbitrary list of stack instructions `l1`, and an arbitrary stack `l2`.

The second example is from the FULL ADDER benchmark. The theorem says that if we convert to a natural number the result of adding a binary number, we get the same natural number we would if we converted that input to a natural number. I present a synthesized helper lemma in Table 2.2, which belongs to category Λ_1 and hence led to a full proof of the theorem.

The third example in the table is from the CLAM benchmark suite and proves the equivalence of two functions for converting a binary tree into a list. For this example, `lfind` produced candidate helper lemmas in both categories Λ_2 and Λ_3. The tenth-ranked candidate, shown in the table, matches the human-provided lemma.

The fourth example in the table is from the LIA benchmark suite and reasons about how pushing onto a queue affect its length. This is a case in which our evaluation does not deem `lfind` to have succeeded, since it does not produce a fully automated proof and does not produce a match for the human-provided lemma in the top ten results. However, the top-ranked result, shown in the figure, is *very* close to the human-provided lemma, which simply replaces the term `(rev l1)` with `l1`. Further, this lemma is itself equally useful in completing the proof, despite being slightly more complex.

Runtime Performance. Figure 2.6 plots the runtime distribution of `lfind` across all 222 evaluation locations. The tool ran to completion on each of these benchmarks with a

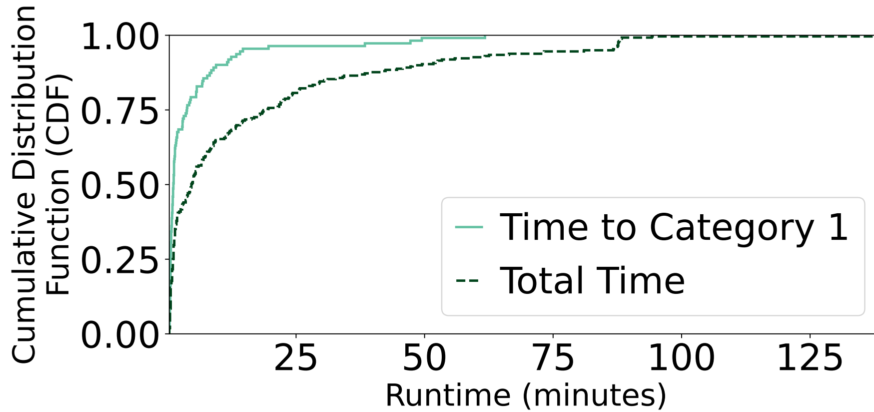

Figure 2.6: `lfind` has a median total runtime of 4.8 min. Further, the tool has a median runtime of 1.2 min for the 109 cases (see Table 2.1) where it was able to a find a full automated proof (Λ_1).

median runtime of 4.8 min (shown in the plot where the curve labeled Total Time reaches a CDF of 0.50). Recall that `lfind` produces a full automated proof (category Λ_1) in 75.7% (see Table 2.1) of the successful evaluation locations. As shown by the curve labeled Time to Category 1 in Figure 2.6, the median and 75th percentile runtime of the tool were only 1.2 min and 3.8 min respectively. These runtimes indicate the viability of the approach and its instantiation in `lfind` to support interactive usage.

Impact of Filtering and Ranking Figure 2.7 provides a detailed view of how many candidate lemmas were generated and filtered, for the results presented in Table 2.1. As explained in § 2.3.3, the approach indeed generates a *lot* of candidate lemmas. For example, `lfind` generates a median of 168 candidate lemmas for each evaluation location from the benchmarks (shown where the solid curve reaches a CDF of 0.50). However, the filtering techniques are very effective in removing useless lemmas. As mentioned in § 2.4, I filter Invalid candidates (labeled `Filter 1` in the figure) as we generate candidate lemmas. I then filter lemmas (labeled `Filter 2`) that are either syntactically similar to each other, or trivial, or restatements or special versions of the theorem statement. After `Filter 1`, the median number of lemmas is reduced to 112. Further, after `Filter 2` there is a median of 17

candidate lemmas. Hence on average, `Filter 1` reduced the candidate lemmas by 33.3%, and `Filter 2` reduced the remaining candidates by 84.8%.

Finally, as mentioned above even after filtering we are left with a median of 17 lemmas for each benchmark suite. This highlights the importance of our ranking strategy, which was already shown to be effective in the results of Table 2.1.

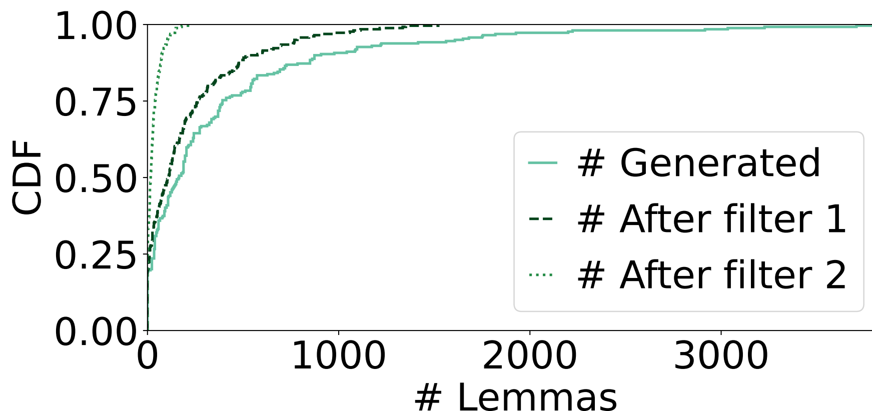

Figure 2.7: `lfind` reduces the number of lemmas by 89.9% on average after application of both filters.

2.5.4 Comparison with Other Approaches

In order to understand how `lfind` compares with other approaches to lemma synthesis, I performed an ablation study in which I compare `lfind` against versions of it that have certain features disabled. First, I compare against a version of the tool that generates *no lemmas*, instead simply using a state-of-the-art automated prover to try to complete the proof from the evaluation location (proof context). Second, I compare against a version of `lfind` that only generates candidate lemmas through generalization, without performing any synthesis. This version of the tool allows us to compare against the commonly used generalization technique [BM79, KM97]. Finally, I compare against a version of `lfind` that is identical to the original version except that it provides no examples to Myth for synthesis. This

change has the effect of forcing MYTH to do *type-guided synthesis*, thereby providing a closer comparison with the term enumeration approach to lemma synthesis [CJR13, YFG19].

No Synthesis In this study, I ran PROVERBOT9001 on each evaluation location across all benchmarks, without providing any synthesized lemma from `lfind`. PROVERBOT9001 can automatically prove only 22.1% of the evaluation locations. In contrast, with a lemma synthesized by `lfind`, PROVERBOT9001 can automatically prove 49.1% of the evaluation locations (109 out of 222), and as shown earlier overall `lfind` provides a useful lemma in 64.9% of the cases. This experiment highlights the need for lemma synthesis and shows how `lfind` complements existing work on automated proofs. These results also serve as a measure of the quality of the human proofs, as the human-provided lemmas are required in the vast majority of cases. Situations where a lemma is used but not needed could arise due to the inexperience of the human prover or simply for readability purposes.

Generalization For this comparison, I disable `lfind`'s synthesis process, so MYTH is not used at all, but all other parts of `lfind` work as described earlier. This version of `lfind` can be seen as a best-case version of the generalization technique [BM79, KM97], since we *exhaustively* consider all possible generalizations, while in prior tools typically only one or a small number of generalizations are heuristically chosen [CDK11, YFG19]. According to the success metrics defined in § 2.5.3, a generalization is deemed useful in only 19.4% of all evaluation locations, as compared with 64.9% of locations for `lfind`.

Type-guided Synthesis For this comparison, I use a version of `lfind` that does not provide any examples to MYTH whenever it is invoked, but is otherwise identical to `lfind`. Without examples, all terms of the desired type will be considered by MYTH to meet the given specification, so the effect is that MYTH will perform a type-guided synthesis through the given grammar, Hence this version of `lfind` is related to the enumeration techniques from prior work on lemma synthesis, like HipSpec [CJR13] and AdtInd [YFG19]. This

version synthesizes a successful helper lemma according to our success metric in 67 evaluation locations, whereas `lfind` does so in 109 evaluation locations. Note that these results exclude cases where generalization produces the useful lemma for an evaluation location, since the two versions of `lfind` are identical in those cases. These results demonstrate the benefits of data-driven synthesis: the examples act as a specification that allows for early filtering of candidate lemmas, which in turn enables the synthesizer to provide higher-quality candidates.

2.5.5 Sensitivity

As described in § 2.3, `lfind` has two hyperparameters: (1) number of synthesis results per sketch, and (2) which terms to select for generating sketches. Further, as described in § 2.4, `lfind` uses PROVERBOT9001 to rank candidate lemmas and the MYTH synthesis engine for term generation. I limit the time spent on each of these tools to efficiently search over the large space of candidate lemmas using available resources. I carry out four separate experiments on the largest benchmark suite (CLAM, with 140 evaluation locations) to understand `lfind`'s sensitivity to each of these parameters. To quantify the sensitivity of a parameter, in each experiment we vary one parameter while fixing all others.

Number of Synthesis Terms. In the first experiment, I vary the number of synthesis results (k) that we ask of MYTH per sketch. We generate sketches from maximal subterms, and use 10s and 12s timeout for PROVERBOT9001 and MYTH respectively. I study the sensitivity to this parameter by varying k to be 5, 15, and 25. Respectively for these settings, `lfind` is successful in 85, 89, and 80 CLAM evaluation locations. There is a modest 4.7% increase in effectiveness from $k = 5$ to $k = 15$, since there is a large search space of candidate lemmas as k increases. However, there is a significant drop in effectiveness from $k = 15$ to $k = 25$ — as the search space increases, the useful candidates can more easily fail to be highly ranked. Figure 2.8 plots the total runtime for different k values, and as expected, the median total time increases with increasing k. Median total time of $k = 5$ is 4.4 min (labeled TOP 5),

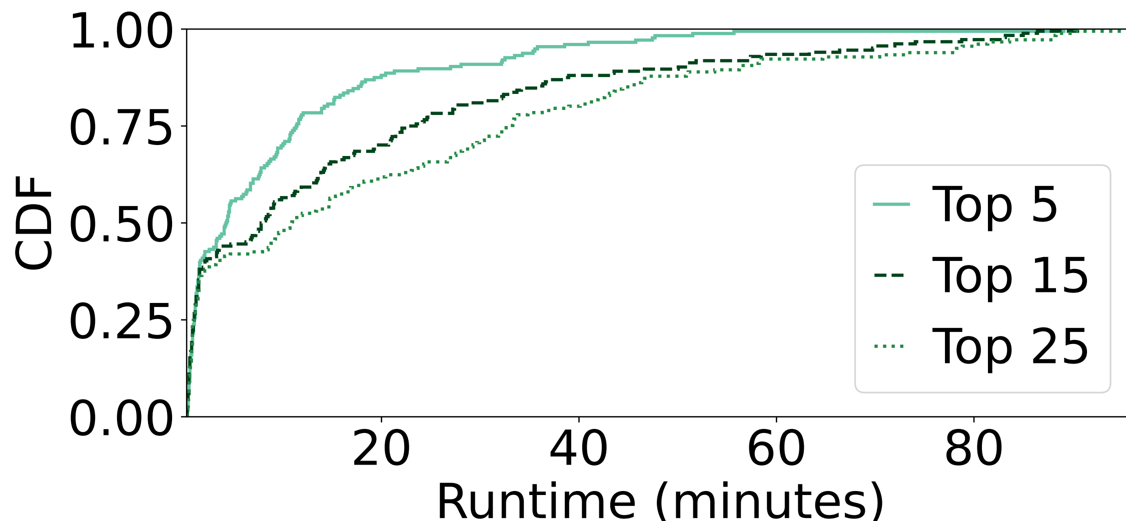

Figure 2.8: Total runtime of `lfind` increases when increasing number of synthesis terms per sketch. Runtime almost doubles when k increases from 5 to 15, while it is 1.3x more when it is increases from 15 to 25.

while it is 8.0 min and 10.9 min for $k = 15$ and $k = 25$ respectively (labeled Top 15, Top 25). I pick $k = 5$ as the optimal number of synthesis terms for the remaining experiments, since the increase to $k = 15$ has a large time cost and only a modest effectiveness benefit.

Proverbot Timeout. In the second experiment, I vary Proverbot9001 timeout to be 5s, 10s, and 15s, setting $k = 5$ and keeping other parameters similar to the previous experiment. Respectively for these settings, `lfind` is successful in 50, 85, and 94 clam evaluation locations. The tool performs poorly with a 5s timeout, since Proverbot9001 spends the first few seconds in setup, leaving too little time for the actual proof search. Figure 2.9 plots the runtimes for the 10s and 15s timeout cases. Median total runtime for 10s (labeled 10 seconds) is 4.4 min, while it is only 3.4 min for 15 seconds (labeled 15 seconds). It is perhaps unintuitive that allowing Proverbot9001 more time leads to lower time overall, but the additional time for Proverbot9001 can allow it to complete a proof that would otherwise not be possible, thereby finding a category Λ_1 lemma sooner. Therefore, I pick 15s as the optimal timeout parameter for Proverbot9001 in the remaining experiments.

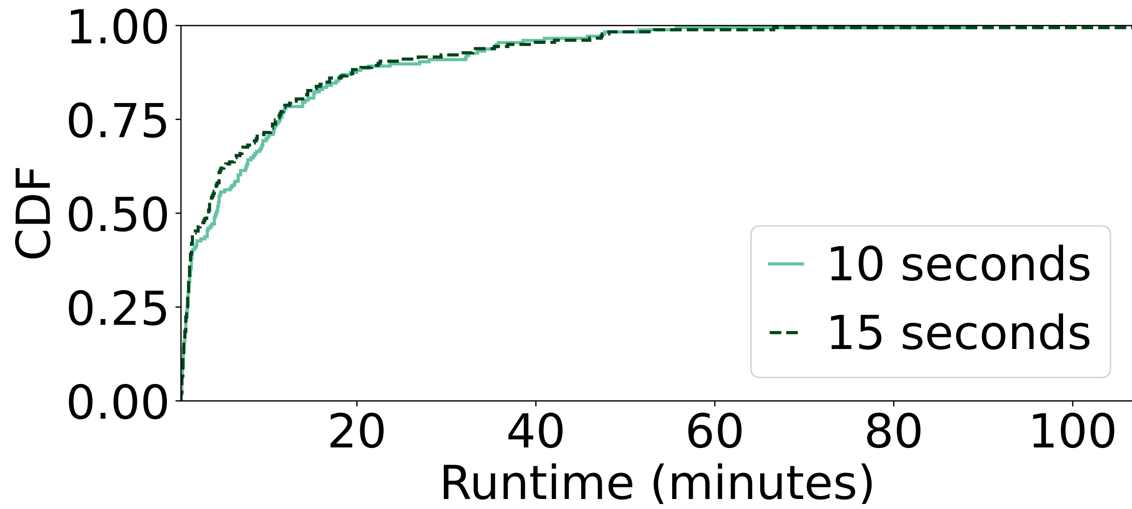

Figure 2.9: Median runtime of `lfind` decreases with an increase in PROVERBOT9001 timeout. While this is unintuitive, this is because the prover is allocated more time per call, enabling it to prove a candidate lemma earlier, which was otherwise not provable using a smaller timeout.

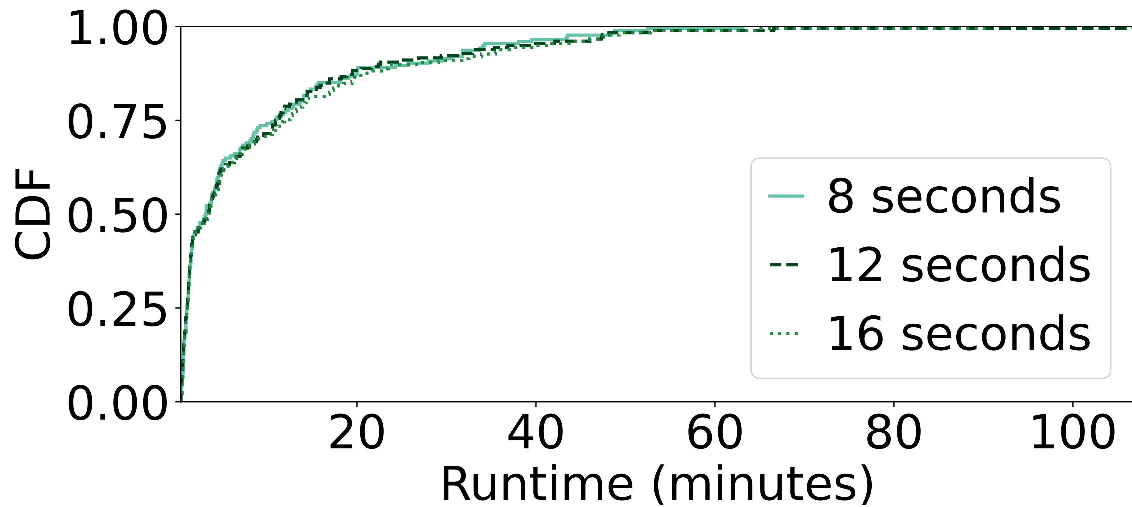

Figure 2.10: Median runtime of `lfind` is unaffected when increasing MYTH timeout.

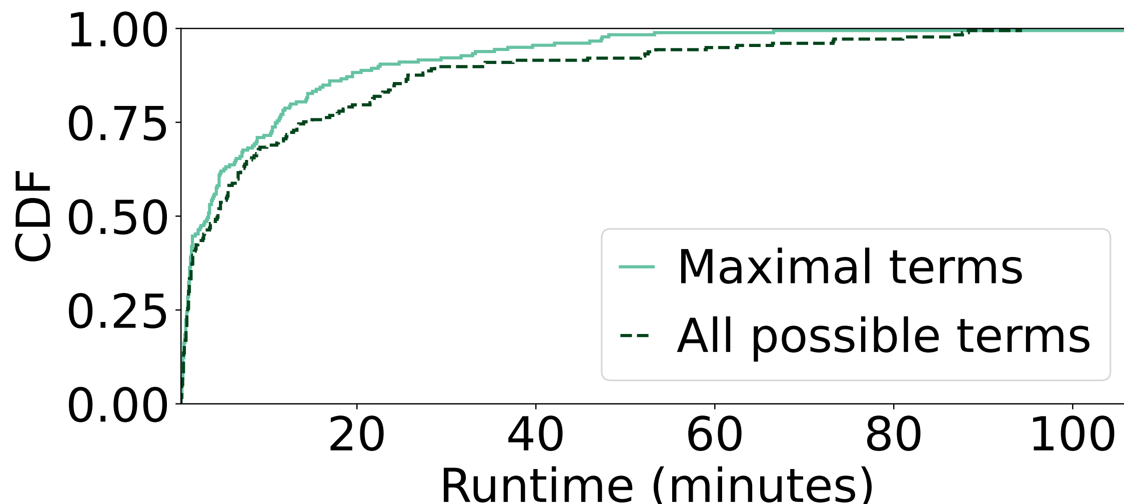

Figure 2.11: There is a modest increase in median runtime of `lfind` from 3.4 min to 4.5 min when generating synthesis sketches from maximal terms compared to all terms.

Myth Timeout. The third experiment varies the MYTH timeout to be 8s, 12s, and 16s, updating PROVERBOT9001 timeout to 15s and keeping other parameters similar to the second experiment. Respectively for these settings, `lfind` is successful in 87, 94 and 94 CLAM evaluation locations. Figure 2.10 plots the total runtime for these timeout values. Despite increasing timeouts, the total runtime is very similar among the three settings, with a median timeout of 3.1 min, 3.4 min, and 3.5 min for 8s, 12s, and 16s respectively. Therefore, I pick 12s as the optimal timeout parameter for MYTH.

Sketch Generation. In this final experiment, I explore two choices for sketch generation, using the optimal choices for other parameters based on the previous experiments. I generate synthesis sketches from (1) all subterms of sort `Type` or (2) only from maximal subterms of sort `Type`. To make the use of maximal terms more feasible, for that setting we also use a heuristic that requires the synthesized expression to refer to all generalized variables from the sketch. The use of all terms is successful in 98 evaluation locations while the use of maximal terms is successful in 94 locations. Figure 2.11 plots the total runtime for these settings, and as expected, the total runtime is more when generating sketches from all subterms compared

to only maximal subterms. However, the difference in the median runtime is only one minute. Therefore, I pick *all subterms* as the optimal parameter for sketch generation.

2.6 Related Work

2.6.1 Lemma Synthesis

As described in § 2.1, there are a variety of existing approaches to lemma synthesis, and they broadly fall into two categories. Many techniques perform rewrites on the target theorem or the current proof state, in order to identify stronger induction hypotheses and helper lemmas. Most common among these is the *generalization* technique [BM79, Aub76, Cas85, Hum90, Hes92, DF03, KM97], whereby selected terms are replaced by fresh variables. Other works go beyond generalizing variables to a broader set of rewrites [KS96, BSV93, JDB10, SDE12]. For example, the *rippling* technique [BSV93] employs a set of rewrite rules in order to make the current goal match the induction hypothesis.

The other category synthesizes candidate lemmas from a grammar using bottom-up enumeration. QuickSpec [CSH10] employs this approach and filters candidates by searching for counterexamples [CH00]. HipSpec [CJR13] combines QuickSpec with an automated prover in order to synthesize a set of provably correct lemmas. A similar enumerate-and-filter strategy is used to automate induction in the CVC4 solver [RK15]. Finally, AdtInd [YFG19] employs bottom-up enumeration in order to search for candidate lemmas in the context of an automated prover for abstract datatypes. Notably, like `lfind`, AdtInd leverages both generalization and sketches (which they call *templates*) for synthesis, but it is unclear how generalizations are chosen and the sketches are user-provided.

`lfind`'s key innovation over these prior works is showing how to reduce the problem of lemma synthesis to a form of *data-driven* program synthesis. Versus the first category of approaches, `lfind` explores a wider space of potential lemmas via grammar-based synthesis and can leverage off-the-shelf program synthesizers. Versus the second category of approaches,

`lfind` generates candidates that are directly targeted toward the current goal, which is critical in an interactive setting. However, the approach borrows several techniques from these prior works. First, `lfind` also employs generalization, but it is used not only to directly produce candidate lemmas but also as the basis for producing sketches for program synthesis. Second, `lfind` employs counterexample search to filter candidates, which has been previously used for filtering in both of the earlier approaches [CDK11, CSH10]. Third, `lfind` also employs automated provers, though due to the interactive setting we use them to rank rather than verify candidates.

2.6.2 Data-driven Invariant Inference

Data-driven invariant inference has been widely used for various software engineering tasks, at least since Ernst's dissertation on inferring likely program invariants from data [Ern00]. In this approach, data about concrete program executions is used to generate *positive* and/or *negative* examples, and the goal is to synthesize a predicate that separates these two sets of examples. Recently these techniques have become state of the art for automated program specification and verification [GLM14, GNM16, END18, PSM16, ZMJ18, AMS19]. For example, prior work has shown how to generate examples for data-driven synthesis of *loop invariants* that are sufficient to prove that a function meets its specification [GLM14, GNM16, PSM16]. To our knowledge, only one prior work uses data-driven synthesis in the context of interactive proofs: the Hanoi tool [MPW20] infers likely *representation invariants* to aid users of interactive theorem provers in proving that a data structure implementation meets its specification.

As described in §2.1, the existing data-driven verification techniques fundamentally exploit the specific kind of invariant being targeted, which has a clear logical specification over a fixed set of variables. This enables a natural approach based on CEGIS [Sol09] for both generating examples and verifying candidate invariants. Our setting of lemma synthesis is more general and poses a challenge for data-driven inference, as we lack both a fixed set of variables for the lemma and clear criteria upon which to classify examples as positive or

negative. Hence, I have devised a new reduction to data-driven program synthesis: `lfind` produces sketches from generalizations of the goal state and generates examples for synthesis using the heuristic that a synthesized term should behave consistently to the term that it replaces. I have also developed new approaches to filtering and ranking lemma candidates, to address the lack of clear success criteria in our setting.

2.6.3 Automated Proofs for Interactive Theorem Provers

A variety of tools exist for automatically generating proofs in interactive settings, both in Coq and other languages. Recent techniques use a form of machine learning, for example a neural network, to guide a heuristic proof search, given a set of proof tactics as well as a set of existing lemmas/theorems [SAS20b, FBG20b, YD19b, HDS19, PLR20, BLR19b, GKU17]. Another class of techniques serialize the proof context into a format for input to an external automated solver and then serialize the resulting proof back into the interactive theorem prover [CK18b, BBP11, KU15b, KU15c].

My contribution is orthogonal to these works, which do not perform lemma synthesis. For example, while the machine-learning-based approaches leverage existing lemmas as part of the proof search, they will fail if the existing lemmas are not sufficient. As shown in §2.5.3, `lfind` can improve the capabilities of Proverbot9001 [SAS20b], a state-of-the-art automated prover for Coq based on neural networks, synthesizing lemmas that allow it to prove goals that it otherwise could not. `lfind` uses Proverbot9001 to rank candidate lemmas and produce proofs for ones that are fully automatable. However, the approach is independent of the particular prover used and so for example could instead employ a solver-based prover like CoqHammer [CK18b] or even employ multiple provers to leverage their relative strengths.

2.7 Summary

In this chapter, I developed a new approach to lemma synthesis for interactive proofs that is both goal-directed and expressive. The key technical contribution is a new reduction from the general lemma synthesis problem to a data-driven program synthesis problem. The approach leverages the information available in a given stuck proof state in multiple ways: sampling variable valuations for example generation, generalizing the state to systematically introduce new variables for synthesis, and deriving synthesis sketches from the current goal. I also describe several techniques for filtering and ranking candidate lemmas, which are critical in an interactive setting. While the problem of lemma synthesis is hard in general, the experimental evaluation of our resulting tool `lfind` demonstrates the promise of the approach and quantifies the benefits over other approaches.

CHAPTER 3

Increasing Expressivity of Example-Driven Lemma Synthesis

3.1 Introduction

In the previous chapter, I proposed a data-driven lemma synthesis approach that is goal-directed and expressive. Despite its success, this approach suffers from two key limitations. First, recall that sketches for synthesis are derived from a generalization of the original goal, therefore, the generated lemmas will always have the same top-level structure as the goal. For example, if the original goal has the form `A = B` then the candidate lemmas will also have this form. These candidate lemmas are useful in applying to the full goal state or rewriting one side of the equality in the goal state. However, some cases may require equality helper lemmas about the subterms in the goal. `lfind` fails to produce candidate lemmas that target these subterms of the goal state. This limitation is exacerbated by lemmas with arbitrary propositions. This is because lemmas with arbitrary propositions often rely on the use of subterm rewriting that requires helper lemmas with equality proposition.

Second, many natural lemmas are also *conditional*, where a particular property is true only under certain circumstances and `lfind` fails to produce these conditional candidate lemmas. If the stuck proof state has a goal similar to the consequent of the helper lemma, then `lfind` can potentially synthesize that consequent. However, if the required helper lemma is conditional, then `lfind` produced lemma, despite matching the consequent, would not be valid and is not useful.

To address the first limitation of generating equalities about subterms, I extend the kind of lemma sketches defined by `lfind`. At each generalization point, we produce an additional sketch with an equality proposition, where one side consists of a version of a subterm of the goal and the other side is a *hole* to be synthesized. We sample valuations of the variables in the subterm to generate input examples, and the expected output value for each example is determined by the value of the subterm. In this way, we require that the synthesized expression's behavior be consistent with that of the subterm. A key insight is that we are able to use the same data-driven synthesis setup as `lfind` for this additional sketch.

I address the second limitation of `lfind` by proposing a counterexample-guided refinement algorithm to generate conditional lemmas. Recall that `lfind` uses the goal in the stuck proof state to create candidate lemmas, by first generalizing the goal, then creating a hole in the goal, and finally synthesizing a new term for the hole. But in general, a stuck-proof state will also include a set of assumptions, for example, the induction hypothesis but also other assumptions from the statement of the theorem and the case analysis. Hence, any candidate lemma l generated by the approach can also induce a candidate of the form $h_1 \rightarrow ...h_n \rightarrow l$, where h_i formulas are generalized versions of the assumptions in the current proof state. Technical challenges include determining assumptions that are likely to be useful and minimizing the number of assumptions required in the final synthesized lemma. To address this challenge, I propose a counterexample-guided refinement procedure to choose the right set of hypotheses from the goal state when generating a candidate lemma. We collect counterexamples to the validity of a candidate lemma and use these valuations to identify hypotheses that are contradicted. In this way, I show how to iteratively choose the right set of hypotheses from the goal state to generate candidate lemmas that are conditional.

I have implemented the proposed extensions in a tool called `lfind++`, which extends `lfind`. Coq users can invoke `lfind++` as a tactic at any point in their proof, and it will produce a set of ranked lemma candidates.

I evaluate the approach on four benchmark suites used by `lfind`, namely, CLAM [IB96] and LIA [YFG19], FULL ADDER [cir95], compiler correctness [Chl13b], as well as two new benchmarks hardware [har96], and additions [add18]. Together, there are 323 evaluation locations from these benchmarks, where a human prover used an auxiliary lemma to progress. `lfind++` synthesizes a useful lemma for 244/323 of these locations, with a median runtime of 2.6 minutes (see § 3.4.3). At 186 of these locations `lfind++` provides a full automated proof of the synthesized lemma and the goal; at the other 58 locations `lfind++` produces a ranked list of lemma candidates where the human-written lemma is in the top 10. I also show that the approach outperforms `lfind` which could synthesize a useful candidate lemma in only 158 locations (see § 3.4.4). The proposed extensions of `lfind++` synthesizes 54% more candidates than vanilla `lfind`.

3.2 Overview

In this section, I describe the contributions of `lfind++` using two examples.

3.2.1 Motivating Example for Generating Equality Lemmas about Subterms

To illustrate how `lfind++` works on identifying equality helper lemmas for subterms of the goal state, we'll start with an example. Figure 3.1 shows Coq code that tries to prove the following strict inequality: length of two appended lists is less than length + 1 of the two lists appended in the reverse order. It starts by defining lists of nats along with definitions for append and length of lists. Following that is an attempt to prove the theorem, named `appLenS`.

The proof proceeds by induction on the list `l`. The `Nil` case is proven using a relatively simple `appToNil` helper lemma. For this case, `lfind++` is able to synthesize the required helper lemma as the top result. In the `Cons` case, after simplification, and application of a library lemma `lt_n_S`, the user is stuck, because the goal is not in a form that enables direct

```
1  Inductive lst : Type :=
2    | Nil : lst
3    | Cons : nat -> lst -> lst.

5  Fixpoint app (l1 : lst) (l2 : lst) : lst :=
6  match l1 with
7    | Nil => l2
8    | Cons n l1' =>  Cons n (app l1' l2)
9  end.

11 Fixpoint len (l : lst) : nat :=
12 match l with
13     | Nil => 0
14     | Cons x y => 1 + len y
15 end.

17 Lemma appToNil : forall l, len (app l Nil) = len l.
18 Proof.
19     ...

21 Lemma lt_n_S : forall n m, n < m -> S n < S m.
22 Proof.
23     ...

25 Lemma appLenS : forall x y, (len (app x y)) <  S (len (app y x)).
26 Proof.
27     intros.
28     induction x.
29     - simpl. rewrite appToNil. auto.
30     - simpl. apply lt_n_S. (* I'm stuck! *)
```

Figure 3.1: A partial proof of a theorem containing non-equality proposition in Coq that requires an auxiliary lemma.

```
1  n: nat
2  x, y: lst
3  IHx: len (append x y) < S (len (append y x))
4  ----------------------------------------------
5  len (append x y) < len (append y (Cons n x))
```

Figure 3.2: The proof state when the user gets stuck.

use of the induction hypothesis. Figure 3.2 shows the proof state at that point, including

the current assumptions and goal. Note that the hypothesis and the goal reason about the length of a list using an inequality proposition.

To get unstuck, the user can invoke our tool `lfind++` as a tactic at this point. In this example, the top three lemmas that `lfind++` produces are as follows:

```
1 (Λ1) Lemma lem1: forall n l l1,
2        len (app l (Cons n l1)) = S (len (app l l1)).
3 (Λ1) Lemma lem2: forall n l l1,
4        len (app l (Cons n l1)) = len (Cons n (app l l1)).
5 (Λ1) Lemma lem3: forall n l l1,
6        len (app l (Cons n l1)) = len (Cons O (app l l1)).
```

Similar to `lfind`, each lemma is bucketed into one of three categories (Λ_1, Λ_2, or Λ_3), and the categories are presented to the user in that order. Λ_1 lemmas are those in which `lfind++` can automatically find a complete proof of the original goal using the generated lemma and Proverbot9001, a state-of-the-art automated prover. In this case, the top three lemmas produced by `lfind++` are Λ_1 lemmas. The full proof of the theorem produced by the tool for `appLensS` using `lem1` is shown in Figure 3.3. Additionally, `lem1` exactly matches the human-provided helper lemma. Although the original proof state contains $<$ proposition, `lem1` contains $=$ proposition. Note that `lfind` does not identify any of these lemmas, in fact, it does not produce any useful lemma for this example.

In the rest of the section, I explain how `lfind++` produces these results.

Approach. Similar to `lfind` we start by **generalizing** the goal state. From each generalization, we create sketches and sample variable valuations from the current goal in order to reduce lemma synthesis to data-driven program synthesis. The key innovation is in extending the type of sketches per generalization while reusing the same data-driven synthesis setup as `lfind`. This enables the synthesis of candidate lemmas for subterms in the goal state. Finally, we use similar filters as `lfind` to remove candidates that cannot be useful and rank and categorize the remaining candidates for user inspection.

```
1  Lemma lem1: forall n l l1,
2        len (app l (Cons n l1)) = S (len (app l l1)).
3  Proof.
4      induction l.
5      simpl.
6      intros.
7      eauto.
8      intros.
9      simpl.
10     rewrite IHl.
11     easy.
12 Qed.

14 Lemma appLenS : forall x y, (len (app x y)) <  S (len (app y x)).
15 Proof.
16     intros.
17     induction x.
18     - simpl. rewrite appToNil. auto.
19     - simpl. apply lt_n_S. unfold len. simpl. rewrite lem1. eauto.
20 Qed.
```

Figure 3.3: A full proof provided by `lfind`.

Generalization. As with `lfind`, I start by generalizing `(append x y)` with a fresh variable `l1` of type `lst` produces the following generalization:

```
forall l1 y n x, (len l1) < len (append y (Cons n x)).
```

This generalization does not produce a valid lemma, therefore I set up the following synthesis problem.

Synthesis. From each generalization, I create multiple sketches to generate candidate lemmas for goal state with arbitrary proposition while reusing the same data-driven synthesis problem as `lfind`. Specifically, I create two kinds of *sketches*. The first sketch is the same as what `lfind` produces, where the sketch is a version of that generalization with one term replaced by a *hole*. For example, if we replace the term `len (append y (Cons n x))` in the generalization above with a hole, then we end up with the following sketch. Note that we quantify over variables present in the sketch.

```
forall l1, (len l1) < □.
```

Intuitively, the first kind of sketch help creates candidate lemmas that are structurally similar and share a root proposition with the original lemma. As explained in § 3.1, while these are useful, they are not the only kind of helper lemmas that are useful for a proof state. We may require equality lemmas about the subterms in a proof state. Therefore, we create a second kind of sketch with an equality proposition, where one side is filled with a subterm from the generalization and the other side is a *hole*. For example, for the subterm `len (append y (Cons n x))` from the goal state, we create the following sketch

```
forall y n x, len (append y (Cons n x)) = □
```

Note that the second kind of sketch can be used as a helper lemma in a goal state with equality and non-equality propositions. In the case of equality propositions, it is used to rewrite smaller subterms within the goal state with equivalent expressions.

Recall from `lfind` that we generate concrete examples of the original goal in the stuck state and then map them to input-output examples for data-driven synthesis. The synthesis process is the same for both these sketches. Just for illustration, I explain the synthesis process for the second sketch, however, this is the same as what `lfind` does. In the running example, the original goal has three variables, `n`, `x` and `y`, so we randomly generate values for `(n, x, y)` tuples.

Next, we map these examples to our second sketch. We already have values for variables in the sketch, i.e. for `n`, `x` and `y`. We just need to produce the expected value of the hole for each example, by leveraging the fact that the value of the hole will be the same as the subterm `len (append y (Cons n x))`.

As a result of this mapping, we can now produce a set of input-output examples that act as a specification for synthesis, each mapping `(n, x, y)` tuple to the expected output value of the term to be synthesized.

Finally, we pass these input-output examples to a data-driven synthesizer. In addition to the examples, we provide the type of the function to be synthesized (which in this case is `nat * lst * lst → nat`) and a grammar to use for term generation. Similar to `lfind`, we automatically create a grammar consisting of the definitions that appear in the stuck proof state along with definitions that they recursively depend upon.

In our example the grammar includes the constructors `Nil` and `Cons` and the functions `app` and `len`. One term that the synthesizer generates from these inputs is `S (len (app x y))`. Substituting this expression into the hole in our sketch yields exactly the lemma `lem1` shown earlier, which enables a fully automated proof of the original lemma.

In summary, I have shown how to generate useful helper lemmas for subterms in a goal state, in a targeted way, based on the current proof state, using a novel observation regarding the different ways helper lemmas can be used in a proof context and augmenting sketch creation to cater to these different cases, making them applicable to a wider class of lemmas. As I demonstrate in § 3.4.4, `lfind++` can generate significantly more useful lemmas for a variety of interesting benchmarks compared to `lfind`. Note that we apply the same filtering and ranking techniques as `lfind`.

3.2.2 Motivating Example for Generating Conditional Lemmas

I illustrate how `lfind++` generates conditional helper lemmas using an example. Figure 3.4 shows Coq code that tries to prove a theorem: count of the occurrence of an element x in a list l where an element y is inserted using insertion sort is the same as the count of an element x in list l when x is not the same as y. It starts by defining lists of `nats` along with definitions for counting, and insertion sort of lists. Following that is an attempt to prove the theorem, named `count_insort`.

The proof proceeds by induction on list l. The `Nil` case is proven using a helper lemma `eqb_false_iff` and hypotheses from the goal. Although this case uses a helper lemma, I

```
1  Inductive lst : Type :=
2    | Nil : lst
3    | Cons : nat -> lst -> lst.

5  Fixpoint count  (arg1 : lst) (arg2 : nat): nat:=
6  match  arg1,arg2 with
7    | Nil, x => O
8    | Cons y z, x => if eqb x y then
9                        S (count z x)
10                    else
11                       count z x
12 end.

14 Fixpoint insort  (arg1 : lst) (arg2 : nat) : lst:=
15 match arg1, arg2 with
16  | Nil, i => Cons i Nil
17  | Cons x y, i => if less i x then
18                      Cons i (Cons x y)
19                    else
20                      Cons x (insort y i)
21 end.

23 Lemma eqb_false_iff: forall (x y: nat), x <> y -> eqb x y = false.
24 ...

26 Theorem count_insort: forall (x y: nat) (l: lst),
27 x <> y -> count (insort l y) x = (count l x).
28 Proof.
29     intros.
30     induction l.
31     - simpl. apply eqb_false_iff in H. rewrite H. auto.
32     - simpl. destruct (less y n) eqn:El; destruct (eqb x n) eqn:Ee.
33         + simpl.
34           apply eqb_false_iff in H.
35           rewrite H. rewrite Ee.
36           auto with arith.
37         + (* I'm stuck! *)
```

Figure 3.4: A partial proof of a theorem in Coq that requires an auxiliary lemma.

illustrate our contributions using the `Cons` case. In this case, destructing `less` and `eqb` leads to four sub-cases. And in the second sub-case, a user is stuck with the proof state illustrated in Figure 3.5.

```
1 x, y, n: natural
2 l: lst
3 H: x <> y
4 IHl: count (insort l y) x = count l x
5 El: less y n = true
6 Ee: eqb x n = false
7 ---------------------------------------------
8 count (Cons y (Cons n l)) x = count l x
```

Figure 3.5: The proof state when the user gets stuck.

To get unstuck the user can invoke `lfind++` as a tactic at this point. In this example, the top lemma produced by `lfind++` is:

```
1 (Λ2) Lemma lem1: forall x y l,
2     x <> y -> count (Cons y l) x = count l x.
```

`lem1` is a Λ_2 lemma which was sufficient to automatically prove the original goal, but PROVERBOT9001 cannot automatically prove the auxiliary lemma, since this requires additional helper lemmas. Note that `lem1` generated by `lfind++` is a conditional lemma, and this exactly matches the user-provided helper lemma.

Approach. In addition to the generalization and synthesis steps, `lfind++` includes a counterexample-guided refinement step to generate conditional lemmas. Recall that a candidate lemma is produced via a combination of generalization and synthesis. The key innovation is in utilizing counterexamples to the validity of a candidate lemma to choose the correct set of hypotheses to include from the stuck proof state to generate a valid candidate lemma. Note that filtering and ranking techniques proposed by `lfind` remain useful for this case.

`Generalization.` As with the previous example, we start by producing all generalizations of the goal state.

For example, replacing `Cons n l` with a fresh variable `l1` of type `lst` produces the following generalization:

```
forall l y x l1, count (Cons y l1) x = count l x.
```

In addition to generalizing terms in the goal state, we also generalize the same terms present in the hypotheses. For this particular generalization, there are no terms in the hypotheses that need to be generalized. However, let's say we generalize the term `count l x` to a variable `l2`, then we need to generalize the same term in hypothesis `IHl` with the same variable.

`Synthesis.` For the above generalization candidate we set up the following synthesis problem. We create a sketch where `count l x` is replaced with a hole.

```
forall y x l1, count (Cons y l1) x = □.
```

Similar to the example described in § 3.2.1, we want to fill the hole by synthesize an alternate expression to `count l x` using variables `y`, `x`, and `l1`. For brevity, we skip the details of the data-driven synthesis setup. One such expression generated by the synthesizer is `count l1 x`. Substituting this expression into the hole in our sketch yields the following candidate lemma:

```
forall y x l1, count (Cons y l1) x = count l1 x.
```

`Counterexample-Guided Refinement.` This candidate lemma generated by `lfind++` is invalid. When we run QuickChick on this candidate lemma, there are valuations to variables `y`, `x`, and `l` such that the lemma is false. A potential way to repair an invalid lemma is to identify conditions under which a particular lemma is true. In the most general setting, identifying such a sufficient condition can be set up as a data-driven synthesis problem using examples and counterexamples. Our key insight is that we can avoid this expensive computation by utilizing additional information (i.e. hypotheses) present in the goal. In Coq, typically when a conditional helper lemma is used in a proof context, it introduces new goals, (i.e. conditions from the helper lemma) which are then proved using one or more hypotheses

from the goal state. Note that there are cases when new goals introduced by conditional lemmas require additional hypotheses, but the most common case for conditional lemmas is the former. Therefore, we propose a counterexample-guided refinement procedure to choose the right set of hypotheses from the goal state when generating a candidate lemma.

To this end, we track a candidate lemma's validity counterexamples. In our running example, the current candidate is invalid for the following valuations of variables.

```
x = 0
y = 0
l1 = []
```

Next, we evaluate the hypotheses in the goal state for these values and identify those that are violated. From Figure 3.5, there are four hypotheses (i.e. `H`, `IHl`, `El`, `Ee`) in the goal state. Note that to evaluate a hypothesis, we need the variables of the hypothesis to be a subset of the variables of the candidate lemma. For our example, only hypothesis `H` from the goal state is applicable, the other hypothesis contains variables that do not have valuations.

To identify if a hypothesis is violated, we substitute the counterexample values. In this case, for `H`, we substitute values for `x` and `y`.

```
 0 <> 0.
```

This is clearly a contradiction, therefore we add `H` as a condition to the candidate lemma and repeat this process until there are no more counterexamples or hypotheses left from the goal state. Adding the hypothesis leads to the generation of candidate lemma `lem1`, which is also the user-provided helper lemma for this proof state.

```
forall x y l1, x <> y -> count (Cons y l1) x = count l1 x.
```

In summary, I have shown how to generate conditional candidate lemmas in a targeted way, based on the current proof state, using a novel combination of data-driven program

```
1  Input:l:LEMMA, s:SUBTERM

3  Returns:A set sketches

5
    1: SKETCHES := ∅
    2: SKETCHES := SKETCHES ∪ l[s ↦ □]
    3: SKETCHES := SKETCHES ∪ s = □
    4: return SKETCHES
```

Figure 3.6: Sketch Generation.

synthesis and counterexample-guided refinement. While the conditions for the lemmas can be arbitrary, we propose a solution to the common case where the conditions can be derived from the hypothesis. As I demonstrate in § 3.4, our approach can generate useful lemmas for a variety of interesting benchmarks.

3.3 Algorithms

In this section, I describe the core algorithms that make up our approach.

3.3.1 Equality Lemmas about Subterms

Figure 3.7 presents the combined algorithm of `lfind` and the proposed extension to generate equality lemmas about subterms. We are given $\mathcal{H}$, containing a set of logical formulas that are the current *hypotheses*, g is a logical formula that is the current *goal*, Γ is a type environment for all free variables in $\mathcal{H}$ and g, $\mathcal{D}$ is a set of type and term definitions that are recursively referred to in $\mathcal{H}$ and g, and $\mathcal{E}$ is a set of examples that satisfy the current proof state, which is generated using a *disprover*. Note that we require that the goal g be unquantified, which in practice typically means that the original lemma/theorem should have all variables universally quantified at the front. The goal is to return a set of ranked candidate lemmas L_r.

```
1  Dependencies: A synthesizer Synth  a disprover Disprover, and a prover Prover

3  Input:𝓗:HYPOTHESES, g:GOAL, Γ:TYPE ENVIRONMENT, 𝓓:GRAMMAR,
      𝓔:EXAMPLES

5  Returns:A set of ranked candidate lemmas

7
    1: L_c := ∅
    2: GENPOINTS := 2^{GETTERMS(g)}
    3: 𝓛_g := GENERALIZELEMMA(𝓗, g, Γ, GENPOINTS)
    4: for l_g ∈ 𝓛_g do
    5:   if IS_LIKELYVALID(DISPROVER(l_g))) then:
    6:     L_c := L_c ∪ (l_g)
    7:   else:
    8:     SYNPOINTS := GETTERMS(l_g)
    9:     for s in SYNPOINTS do
   10:       SKETCHES := GETSKETCHES(l_g, s)
   11:       for SKETCH in SKETCHES do
   12:         l_s= SYNTHESIZELEMMA(Synth, SKETCH, 𝓔, s)
   13:         if  IS_LIKELYVALID(DISPROVER(l_s))) then:
   14:           L_c := L_c ∪ (l_s)
   15: L_f := FILTER(Disprover, L_c)
   16: L_r := RANK(Prover, L_c)
   17: return L_r
```

Figure 3.7: `lfind++` Algorithm.

The initial set of candidate lemmas is empty and the tool starts by collecting all terms in the goal state g and constructing the power set of these terms named GENPOINTS. Based on this, the tool generates a set of generalized lemmas, where terms from GENPOINT are replaced with a fresh variable (see Definition 2.1). `lfind++` then iteratively performs a loop on lines 4-14 to generate candidate helper lemmas. First, if a lemma is not disprovable, it is added to the list of candidate lemmas. If it is disprovable, then the tool iteratively carries out data-driven synthesis (lines 9-14) for each term in the generalized lemma l_g. The algorithm uses *sketches* to generate candidate lemmas, where new subexpressions that behave consistently with some term are synthesized (lines 11-14). Figure 3.6 describes the sketches produced by `lfind++`. This function takes as input a lemma, and a subterm and returns

possible sketches for synthesis. The first sketch replaces the subterm s with a hole and the second sketch is an equality lemma where the subterm is equal to the hole. This additional sketch creation is the only difference from `lfind`. Note that GetSketches function returns sketches that enable generation of candidate lemmas that can be used with apply and rewrite tactic. Note that we are able to use the same synthesis process. Finally, the candidate lemmas are filtered and a ranked set of lemmas are returned for user inspection.

3.3.2 Conditional Lemmas

In this section, I describe how we reduce hypotheses selection from the goal state to a counterexample-guided refinement procedure. For notations refer to § 2.3 from Chapter 3. As described in § 3.2.2, the first step before generating conditional lemmas is to produce candidate lemmas via the combination of generalization and synthesis. We start by identifying if a candidate lemma is disprovable.

Definition 3.1. (Counterexample: ρ) Given a disprover $\mathcal{C}$, candidate lemma l_c, and $\mathcal{D}$ which is a set of type and term definitions recursively referred to in l_c, we define counterexample $\rho = \mathcal{C}(l_c, \mathcal{D})$, such that $l_c \Downarrow_\rho False$, where ρ is $\langle x_1 : v_1, ..., x_n : v_n \rangle$ and x_i are universally quantified variables of l_c.

`lfind++` uses this counterexample to identify hypotheses that are violated when evaluated with the variable values of the counterexample. Note that ρ is empty when $\mathcal{C}$ returns Don't Know.

Definition 3.2. (Violation) Given a proposition ϕ, a sample valuation $\rho = \langle x_1 : v_1, ..., x_n : v_n \rangle$ for universally quantified variables in ϕ, and $\mathcal{D}$ which is a set of type and term definitions recursively referred to in ϕ, we say a sample valuation violates a formula, denoted Violated(ϕ, ρ) is defined as $\phi[x_i \mapsto v_i] \Downarrow_\rho False$.

Identifying if a proposition is false is a hard problem in general. Instead, we use a more efficient incomplete approach and use a *disprover* to test it.

Finally, we can put all of this together to specify how to reduce hypothesis selection to counterexample-guided refinement.

Definition 3.3. (Hypothesis Selection as Counterexample-Guided Refinement) Given a candidate lemma l_c, a set of generalized hypotheses $\mathcal{H}_g$, and $\mathcal{D}$ which is a set of type and term definitions recursively used in l_c, we produce the following counterexample-guided refinement problem.

- Let ρ be the counterexample to l_c, where $\rho = \langle x_1 : v_1, ..., x_n : v_n \rangle$.
- Let $\mathcal{H}_v$ be $\mathcal{H}_g$ restricted to those hypotheses such that $\textsc{Violated}(\mathcal{H}_i,\ \rho)$ where $\mathcal{H}_i \in \mathcal{H}_g$.

We add $\mathcal{H}_v$ as additional hypotheses to l_c forming a new candidate $\mathcal{H}_v \rightarrow l_c$.

The above definition details a single refinement step. Further, Figure 3.8 presents the hypothesis selection algorithm which is run as part of the algorithm presented in Figure 3.7 whenever it finds a candidate lemma that is disprovable. We are given a candidate lemma, a set of hypotheses, and an initial counterexample to the lemma, which contains valuations for the free variables of the lemma. The initial set of hypotheses to be added to the candidate lemma is empty. It is possible that identifying the right set of hypotheses requires multiple refinement steps. Therefore, we iteratively continue to add hypotheses, if the disprover returns a counterexample and ends when there are no more counterexamples or hypotheses left to be added.

3.4 Experimental Evaluation

3.4.1 Benchmark Suite

As with `lfind`, the approach generates candidate helper lemmas from a given proof context. Hence, I evaluate `lfind++` on *evaluation locations* which are points in the proof where a user-provided helper lemma was used (see § 2.5.3). I evaluate `lfind++` on a total of 323

```
1  Input:l:LEMMA,H:HYPOTHESES,ρ:COUNTEREXAMPLE

3  Returns:A candidate lemma

5
    1: do
    2:   H_r := ∅
    3:   for h in H do
    4:     if VIOLATED(h, ρ) then
    5:       H_r := H_r ∪ h
    6:   l_c := H_r ∪ l_c
    7:   ρ := COUNTEREXAMPLE(l_c)
    8: while ISNOTEMPTY(ρ) or ISNOTEMPTY(H_r)
    9: return l_c
```

Figure 3.8: Counterexample-Guided Refinement.

evaluation locations. Of these locations, 222 of those are from the benchmarks used in `lfind`. The rest of the benchmark locations are drawn from the following sources.

- LIA (41): 19 of the 38 evaluation locations of this benchmark is the same as `lfind`. We augmented the rest of the evaluation locations to require linear integer arithmetic and contain arbitrary propositions or require conditional helper lemmas.
- HARDWARE (38): This project [har96] from `coq-contribs` collection, formalizes hardware linear arithmetic structures. The project contains several theorems about arithmetic, booleans, representation of natural numbers as lists of digits in a given base, and so on.
- ADDITIONS (32): This project [add18] from `coq-contribs` collection, formalizes common mathematical properties.

The HARDWARE and ADDITIONS benchmark suites already contain full Coq proofs written by others, which in turn determine the evaluation locations. Table 3.1 provides more details on the proof state and the kind of helper lemmas used in the evaluation locations. As shown in the first row of the table, 76.7% of the evaluation locations contain a proof state with an equality proposition. This is expected since 222/323 evaluation locations were taken from

Table 3.1: lfind++ Benchmark Summary

	#
Proof state with eq prop	248
Proof state with non-eq prop	75
Proof state requires a conditional helper lemma	64
Proof state requires a helper with non-eq prop	6
Proof state with non-eq prop but requires a helper with eq prop	22

the lfind which works only on lemmas with equality proposition. As shown in the second row, 23% of the evaluation locations contain propositions other than equality like lt, le, and so on. The last three rows summarize the kind of helper lemmas provided by the human prover in these evaluation locations.

3.4.2 Experimental Setup

For each evaluation location, lfind++ generates 50 input-output examples from the current proof state and is allowed to generate candidate lemmas with a maximum timeout of 100 minutes. lfind++ has a median runtime of 2.6 min. The tool has a 12s timeout for each call to Myth and a 15s timeout for each call to Proverbot9001. For these experiments, I generate sketches from *all subterms* of sort Type, and ask for *5* synthesis terms per sketch. These choices were based on the sensitivity analysis done by lfind in § 2.5.5.

All evaluations were performed on a machine that runs MacOS (10.15.6) in a 2.3 GHz-Quad-Core Intel Core i7, with 32GB memory.

3.4.3 Synthesized Helper Lemmas

Table 3.2 summarizes the results for all the benchmark locations. Recall from the previous chapter that we consider the use of lfind++ at an evaluation location to be successful in three scenarios. First, we say that lfind++ is successful if it can produce a candidate helper lemma that is automatically proven by Proverbot9001 and this helper lemma enables

Table 3.2: Results

	CLAM	FULL ADDER	COMPILER	LIA	HARDWARE	ADDITIONS
Setup						
# Theorems	86	40	1	27	105	55
# Evaluation Locations	149	62	1	41	38	32
Results						
# fully proven lemma and goal	82	37	0	20	32	15
# else top 1	31	2	0	2	0	0
# else top 5	10	1	1	2	0	1
# else 10	3	1	0	2	1	1
# else more general top 1	0	0	0	0	0	0
Summary	126/149	41/62	1/1	26/41	33/38	17/32

PROVERBOT9001 to automatically prove the user's goal. This is the best-case scenario, as `lfind++` has produced a complete proof for the user. Second, we say that `lfind++` is successful if a lemma that matches the human-provided lemma is ranked highly (top-10) by the tool. Third, we say that `lfind++` is successful if a lemma that is more general than the human-provided lemma is ranked highly by the tool. I use the $\preceq$ operator defined in § 2.3.3 to automatically identify if a candidate lemma l matches or is more general than the human-provided lemma h. Specifically we say that l matches h if both $l \preceq h$ and $h \preceq l$, and we say that l is more general than h if $h \preceq l$ but not vice versa.

In total, based on the evaluation metrics we see that `lfind++` succeeds in 244 (75.5%) of the 323 evaluation locations across all benchmarks. Further, as shown in the third row of the table, in 186 (76.2%) of these successful 244 locations, `lfind++` was able to synthesize a lemma that led to a fully automated proof of the user's goal. Rows 4-7 of Table 3.2 shows a breakdown of the remaining 58 successful locations. Notably, for 36 of these evaluation locations, the top-ranked candidate lemma produced by `lfind++` matches the helper lemma provided by the human prover.

Examples. Table 3.3 shows examples of lemmas synthesized by `lfind++` along with their rank and category (see § 2.3.4 for category notations). I describe the first four of them in detail.

Table 3.3: A sample of **lfind++** synthesized lemmas and their associated rank and category.

#	Original Theorem	**lfind++** Synthesized Lemma	Λ
1.	Theorem tree_insert_all : ∀ (l:list nat) (t:tree), (tsize t) <= (tsize (tinsert_all t l)).	Lemma lem2 : ∀ (t:tree) (n:nat), tsize (tinsert t n) = S (tsize t).	Λ_2
2.	Theorem sorted_sort : ∀ (x:lst), sorted (sort x) = true.	Lemma lem2: ∀ (n:nat) (l:list nat), sorted l = true → sorted (insort n l) = true.	Λ_2
3.	Lemma dif_0_pred_eq_0_eq_1 : ∀ (n:nat), n <> 0 → pred n = 0 → n = 1.	Lemma lem1: ∀ (n:nat), pred n = O → n = O ∨ n = S O.	Λ_1
4.	Lemma exp_n_incr : ∀ (n m p:nat), n <= m → exp_n n p <= exp_n m p.	Lemma lem1: ∀ (n:nat)(lf1:nat) (m:nat) (n0:nat), n <= m -> lf1 <= exp_n m n0 -> n <= m -> n * lf1 <= m * exp_n m n0.	Λ_1
5.	Lemma half_lt : ∀ a b : nat, 0 < b → b = shift a ∨ b = S (shift a) → a < b.	Lemma lem1 : ∀ (n:nat), n < S (shift n).	Λ_1
6.	Lemma le_reg_minus: ∀ (n m p:nat), n <= m → n - p <= m - p.	Lemma lem1: ∀ n:nat, n = n - O.	Λ_1
7.	Theorem lst_int_subset : ∀ (x:list nat) (y:list nat), lst_subset x y = true -> lst_eq (lst_intersection x y) x = true.	Lemma lem1: ∀ (x:list nat) (lf2:lst) (lf1:lst), lst_subset x lf1 = true -> lst_subset x (lf2 ++ lf1) = true.	Λ_2
8.	Lemma lenSConsRev : ∀ l1 l2 n, len (l1 ++ n :: l2) = S (len (l2 ++ l1)).	Lemma lem1 : ∀ (l : list nat) (l1 : list nat), len (l ++ l1) = (len l1) + (len ls0))).	Λ_1
9.	Lemma rev_nth : ∀ l d n, n < len l -> nth n (rev l) d = nth (len l - S n) l d.	Lemma lem1 : ∀ l, len (rev l) = len l.	Λ_1
10.	Theorem theorem0 : ∀ (x:list nat) (y:list nat), len (qreva x y) = (len x) + (len y).	Lemma lem1 : ∀ (n:nat) (m:nat), n + S m = S n + m.	Λ_1

The first example from the LIA benchmark suite states that the number of elements of a tree is less than the number of elements in a tree when additional elements are added from a list. The inductive case requires a helper lemma that captures the special case of the tree size when one element is inserted. I present the synthesized lemma in Table 3.3, which belongs to Λ_2, and this exactly matches the human-provided helper lemma. Further, note that the proof state contains `<=` proposition, whereas the helper lemma contains equality proposition. This shows how the proposed approach in § 3.3.1 is effective in synthesizing the required helper lemma with arbitrary propositions.

The second example is from the CLAM benchmark suite. The theorem says that checking for sortedness property after sorting a list would always be true. For this example, `lfind++` produces a conditional candidate lemma in category Λ_2. This example illustrates the effectiveness of the proposed approach in § 3.3.2 in synthesizing condtional helper lemmas.

The third example is from the HARDWARE benchmark suite. For this example, `lfind++` identified a candidate in category Λ_1, and hence led to a full proof of the theorem.

The fourth example is from ADDITIONS benchmark suite and reasons about the exponential function. For this example, `lfind++` identified a candidate in category Λ_1, and hence led to a full proof of the theorem. Further, the candidate lemma contains multiple conditions.

Runtime Performance. Figure 3.9 plots the runtime distribution of `lfind++` across all 323 evaluation locations. The tool ran to completion on each of these benchmarks with a median runtime of 2.6 min (shown in the plot where the curve labeled TOTAL TIME reaches a CDF of 0.50). Recall that `lfind++` produces a full automated proof (category Λ_1) in 76.2% (see Table 3.2) of the successful evaluation locations. As shown by the curve labeled TIME TO CATEGORY 1 in Figure 3.9, the median and 75th percentile runtime of the tool were only 1.0 min and 2.85 min respectively. Despite the increased search space of candidate lemmas, these runtimes are lower than that of `lfind`, with a median total runtime is 4.8 min and time to category Λ_1 is 1.2 min. It is perhaps unintuitive that searching over a larger search

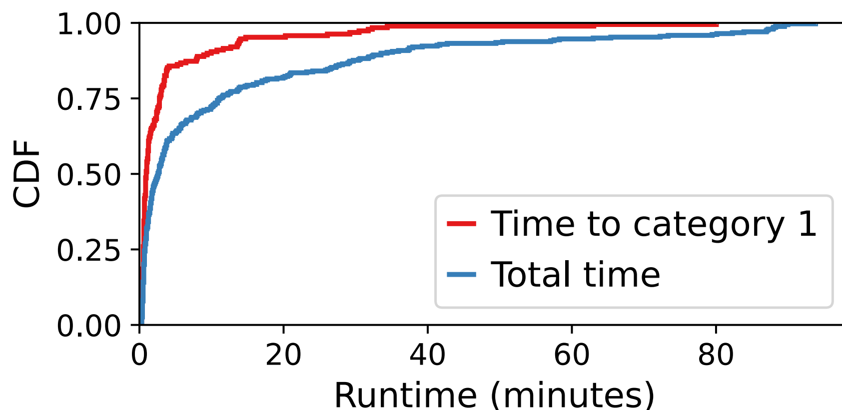

Figure 3.9: `lfind++` has a median total runtime of 2.6 min. Further, the tool has a median runtime of 1.0 min for the 186 cases (see Table 3.2) where it was able to a find a full automated proof (Λ_1).

Table 3.4: Expressivity extensions of `lfind++` significantly outperforms vanilla `lfind`.

	`lfind`	`lfind++`
CLAM	100	126
Full Adder	35	41
Compiler	1	1
LIA	15	26
Hardware	5	33
Additions	2	17
Total	158	244

space leads to lower time overall, but the increased expressivity leads to a useful lemma being identified earlier, leading to a lower runtime. These results indicate the viability of the approach and its instantiation in `lfind++` to support interactive usage.

3.4.4 Comparison with `lfind`

To understand the benefits of the proposed extensions to increase expressivity, I compare `lfind++` against `lfind`. Table 3.4 compares the number of cases `lfind` and `lfind++` were

successful in each benchmark suite. `lfind` succeeds in 158 of the 323 evaluation locations, whereas `lfind++` does so in 244 evaluation locations. The proposed approach outperforms `lfind` on all benchmark suites. As shown by rows 1-3 in Table 3.4, both tools are successful in identifying the required helper lemma in the case where the proof state contains an equality proposition. However, `lfind` performs rather poorly on benchmarks (rows 4-6) that contain proof states with arbitrary propositions or conditional lemmas. In fact, on these three benchmarks `lfind` is deemed useful in only 19.8% of the cases, as compared with 68.4% of `lfind++`. These results demonstrate the benefits of the proposed extensions to data-driven synthesis: adding a new kind of sketch to synthesize helper lemmas about subterms in a goal and using counterexamples to pick the required hypotheses to construct conditional lemmas, which in turn enables the synthesizer to provide higher-quality candidates.

3.5 Summary

In this chapter, I have proposed two extensions to the data-driven lemma synthesis approach for interactive proofs that is significantly more expressive than prior lemma synthesis approaches. Two key technical contributions include (1) a new sketch generation that uses the same synthesis setup as `lfind` while enabling helper lemma generation for subterms in the goal state; (2) a new counterexample-guided refinement procedure that leverages validity counterexamples to select the required hypotheses from the goal state to generate conditional candidate lemmas. While the problem of lemma synthesis is hard in general, the experimental evaluation of our resulting tool `lfind++` demonstrates the promise of the approach and quantifies the benefits over `lfind`.

```
1  Fixpoint power (x n : nat) {struct n} : nat :=
2    match n with
3    | O => 1
4    | S n' => x * power x n'
5    end.

7  Theorem le_mult_right : forall a b : nat, 0 < b -> a <= a * b.
8  Proof.
9  ...

11 Lemma power_lt_O : forall x n : nat, 0 < x -> 0 < power x n.
12 Proof.
13 ...

15 Lemma power_le : forall x n : nat, 0 < n -> x <= power x n.
16 Proof.
17     intros x n; case n; simpl in |- *; auto.
18     intros H'; inversion H'.
19     intros n'; case x; intros; auto.
20     apply le_mult_right; auto.
21     apply power_lt_O; auto with arith.
22 Qed.
```

Figure 3.10: Proof of a theorem in Coq that requires multiple helper lemmas.

```
1  x, n, n', n0: nat
2  H: 0 < S n'
3  ---------------------------------------------
4  S n0 <= S n0 * power (S n0) n'
```

Figure 3.11: The proof state when the user gets stuck.

3.6 Future Work

3.6.1 Synthesizing Conditional Lemmas

To generate conditional lemmas, the extension described in § 3.3.2 proposes to choose a set of hypotheses from the goal state. However, it is possible that the required hypotheses are not available in the goal state. Figure 3.10 illustrates a case where the required helper lemma is conditional and the hypothesis is not available in the goal state. In this example,

a user-provided helper lemma, `le_mult_right` is applied at line 20. Figure 3.11 shows the stuck state before applying the helper lemma. `lfind++` would identify candidate lemmas from this stuck state.

`lfind++` creates the following generalization of the stuck state, where `S n0` is replaced with variable `a` and `power (S n0) n'` is replaced with b. This generalization matches the consequent of the user-provided helper lemma, `le_mult_right`.

```
forall a b, a <= a * b
```

However, no hypothesis in the goal state matches the required condition `0 < b`. Therefore, the counterexample-guided refinement process described in § 3.3.2 would not produce the required candidate lemma.

This limitation can be addressed using a data-driven hypotheses synthesis setup. When we generalize a term with a variable, we lose information about the term's properties. For example, `S n0` is always greater than 0, however, when we generalize we lose this information. Therefore, for each term we generalize, we need to identify and add their properties as additional hypotheses. Once we add these additional hypotheses, we can reuse the counterexample-guided hypotheses selection approach described in § 3.3.2 to generate conditional candidate lemmas.

Concretely, we want to identify predicates that satisfy the values of the generalized terms. To do this, we would set up a boolean synthesis problem containing positive examples. For our example, `S n0` would take values `1, 2, 3` and so on. Next we map these values to `true`.

(1) $\mapsto$ `true`

(2) $\mapsto$ `true`

(3) $\mapsto$ `true`

Finally, we pass these input-output examples to a data-driven synthesizer. In addition to the examples, and grammar inferred from the goal state, we provide the most commonly used

```
1 Lemma mult_succ : forall (x y : nat), x * y +  x = x * S y.
2 Proof.
3    intros.
4    induction x.
5    - reflexivity.
6    - simpl. rewrite plus_succ. rewrite plus_assoc.
7      rewrite (plus_commut y x). rewrite <- plus_assoc.
8      rewrite IHx. rewrite plus_succ. reflexivity.
9 Qed.
```

Figure 3.12: Proof of a theorem in Coq that requires multiple helper lemmas.

operators like `>`, `<` , `and`, `not`, `or` etc. One term synthesized for these inputs is `S n0 > 0`. Similarly, we can synthesize a boolean predicate for the term `power (S n0) n' > 0`.

Once these predicates are added to the hypotheses, `lfind++` can identify the required conditional lemma using the method described in § 3.3.2.

3.6.2 Multiple Helper Lemmas

As currently formulated, a user invokes `lfind++` as a tactic at any point in the proof, and it will produce a set of ranked candidate lemmas. The current ranking scheme is designed to maximize provability metrics, where a lemma is most useful if it leads to proof of the goal state and proof of the lemma itself. However, this ranking scheme might be too restrictive in cases where the required helper lemma is generated by `lfind++`, but an automated prover fails to prove the goal state or the helper lemma. For instance, a proof could require multiple helper lemmas interleaved with other Coq tactics. Figure 3.12 illustrates a case where the inductive case of the proof requires five helper lemmas. In this case, it is possible that even if `lfind++` identifies the correct helper lemma, an automated prover will not be able to prove the goal state without the other helper lemmas. Hence, the ranking scheme will fail to rank this candidate in top-k. This limitation can be addressed with a new ranking scheme based on proof progress rather than full provability. A simple proof progress metric could be to identify if a candidate lemma enables the use of certain Coq tactics like `simpl` or `rewrite` of

the induction hypothesis. Sophisticated progress metrics can potentially be inferred from the failure proof trees produced by Proverbot9001.

CHAPTER 4

Counterexample-Guided Verification of Neural Networks

4.1 Introduction

Deep neural networks are increasingly used to make sensitive decisions, including financial decisions such as whether to give a loan to an applicant [HPS16] and as controllers for safety critical systems such as autonomous vehicles [BDD16, ZHL20]. In these settings, for safety, ethical, and legal reasons, it is of utmost importance that some of the decisions made are monotonic. For example, one would expect an individual with a higher salary to have a higher loan amount approved, all else being equal, and the speed of a drone to decrease with its proximity to the ground. Learning problems in medicine, revenue-maximizing auctions [FNP18], bankruptcy prediction, credit rating, house pricing, etc., all have monotonicity as a natural property to which a model should adhere. Guaranteeing monotonicity helps users to better trust and understand the learned model [GCP16]. Furthermore, prior knowledge about monotonic relationships can also be an effective regularizer to avoid overfitting [DBB01].

Unfortunately, there is no easy way to specify that a trained neural network should be monotonic in one or more of its features. Existing approaches to this problem, such as min-max networks [Sil98], monotonic lattices [FCC16], and deep lattice networks [YDC17], guarantee monotonicity by construction but do so at the cost of significantly restricting the hypothesis class. Other solutions, such as learning a linear function with positive coefficients, are even more restrictive. Furthermore, techniques that enforce monotonicity as a soft constraint in neural networks [SA97, GSM19] suffer from not being able to provide any

provable monotonicity guarantee at prediction time. Finally, the well-known framework of isotonic regression [BB72, SS97] is effective only when the training data can be partially ordered, which is rarely the case in high dimensions.

In this chapter, I develop techniques to incorporate monotonicity constraints for standard ReLU neural networks without imposing further restrictions on the hypothesis space. These techniques leverage recent work that employs automated theorem provers to formally verify robustness and safety properties of neural networks [XTJ17, XTJ18, GMD18, KBD17]. First, I present a counterexample-guided algorithm that provably guarantees monotonicity at prediction time, given an arbitrary ReLU neural network. This approach works by constructing a *monotonic envelope* of the given model on-the-fly via verification counterexamples. Empirically I show that our approach can guarantee monotonicity with little to no loss in model quality at a computational cost on the order of a few seconds on standard datasets. Second, I propose a new counterexample-guided algorithm to incorporate monotonicity as an inductive bias during training. The approach identifies monotonicity counterexamples on the training data, inducing additional supervision for training the network, and perform this process iteratively. I also show that monotonicity is an effective regularizer: the counterexample-guided learning algorithm improves the overall model quality. Empirically, the two algorithms, when used in conjunction, enable better generalization while guaranteeing monotonicity for both regression and classification tasks. I have implemented our algorithms in a tool called "COunterexample-guided Monotonicity Enforced Training" (`COMET`). Finally, I demonstrate that `COMET` outperforms min-max and deep lattice networks [YDC17] on four real-world benchmarks.

4.2 Preliminaries: Finding Monotonicity Counterexamples

I begin by introducing some common notation. Let $\mathcal{X}$ be the input space consisting of d features, and suppose that it is a compact finite subset $\mathcal{X} = [L, U]^d$ of $\mathbb{R}^d$. Let $\mathcal{Y}$ be the

output space. I consider regression and (probabilistic) binary classification tasks where $\mathcal{Y}$ is totally ordered.

The goal will be to learn functions that are monotonic in some of their input features.

Definition 4.1. A function $f : \mathcal{X} \to \mathcal{Y}$ is *monotonically increasing* in features S iff each feature in S is totally ordered and for any two inputs $x, x' \in \mathcal{X}$ that are (i) non-decreasing in features S, $\forall i \in S, x[i] \leq x'[i]$, and (ii) holding all else equal, $\forall k \notin S, x[k] = x'[k]$, the output of the function is non-decreasing: $f(x) \leq f(x')$.

Formal properties of functions are often characterized in terms of their counterexamples. Counterexample-guided algorithms are prevalent in the field of formal methods, for example to verify [CGJ00] and synthesize programs [STB06]. The techniques proposed in this chapter will be centered around using counterexamples to the monotonicity specification.

Definition 4.2. A pair of inputs $x, x' \in \mathcal{X}$ is a *monotonicity counterexample pair* for the ith feature of function $f : \mathcal{X} \to \mathcal{Y}$ iff the points are (i) non-decreasing in feature i, that is, $x[i] \leq x'[i]$, (ii) holding all else equal, that is, $\forall k \neq i, x[k] = x'[k]$, and (iii) the function is decreasing: $f(x) > f(x')$.

Notably, for a function to be (jointly) monotonic in features S, it is both necessary and sufficient that there does not exist a monotonicity counterexample pair for any of the individual features in S.

ReLU neural networks generalize well and are widely used [GBB11, XCL16, SPD19], particularly in the context of verification and robustness. Hence, we will assume that f is a ReLU neural network.

Definition 4.3. A *ReLU neural network* is a directed acyclic computation graph consisting of neurons that compute $\mathrm{ReLU}(\sum_i w_i x_i + b)$, where the activation function is a rectified linear unit $\mathrm{ReLU}(y) = \max(0, y)$, the weights w_i and bias b are parameters associated with each neuron, and neuron inputs x_i are either input features or values of other neurons. The value of a designated output neuron defines the value of a function $f : \mathcal{X} \to \mathcal{Y}$.

Counterexample-guided algorithms rely on the ability to *find* counterexamples, usually by relegating the task to an off-the-shelf solver. This requires that both the counterexample specification and the object of interest — in this case the function f — can be encoded in a formal language amenable to automated reasoning. We will use a *satisfiability modulo theories* (SMT) solver [BT18] for this purpose. Recall that satisfiability (SAT) is the problem of deciding the existence of assignments of truth values to variables such that a propositional logical formula is satisfied. SMT generalizes SAT to deciding satisfiability for formulas with respect to a decidable background theory [BT18]. We will use the background theory of linear real arithmetic (LRA), which allows for expressing Boolean combinations of linear inequalities between real number variables.

The encoding of ReLU neural networks into SMT(LRA) is well-known and readily available [KBD17, HKW17]. Briefly, the relationship between any neuron value and its inputs is encoded in SMT(LRA) as follows. The linear sum over neuron inputs is already a linear constraint. Additionally, we encode the non-linearity of the ReLU activation function using logical implications in SMT. Concretely, for $z = \text{ReLU}(y) = \max(0, y)$, we add two SMT constraints: $y > 0 \rightarrow z = y$ and $y \leq 0 \rightarrow z = 0$.

We can now ask an SMT solver to find monotonicity counterexample pairs: we simply take the (linear) conditions in Definition 4.2 and conjoin with the SMT (LRA) encoding of the function f. Linear real arithmetic is a decidable theory [Tar98]; hence we will always obtain a correct counterexample if one exists. In §4.3, we require the ability to obtain a counterexample that maximally violates the monotonicity specification. Hence, I use Optimization Modulo Theories (OMT) [ST15], which is an extension of SMT for finding models that optimize secondary linear objectives, which is again decidable. Note that the above definitions consider monotonically increasing features, and I assume that form of monotonicity throughout. We can analogously define corresponding notions for monotonically decreasing features, and our algorithms can be applied straightforwardly to that setting as well.

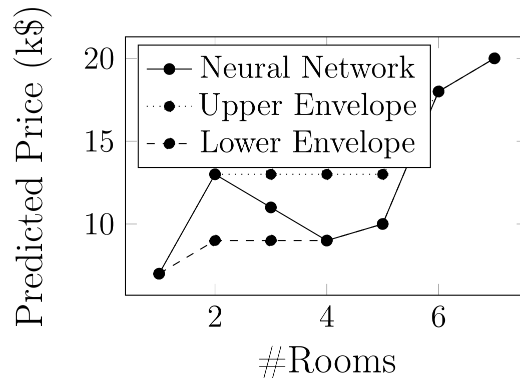

Figure 4.1: Monotone envelopes around a simple non-monotone learned function

While this setup allows us to verify monotonicity of a learned function, it is not at all clear how to *guarantee* monotonicity, or how to enforce monotonicity during training as an inductive bias. The next two sections present the counterexample-guided algorithms that address these challenges.

4.3 Counterexample-Guided Monotonic Prediction

A neural network trained using traditional approaches is not guaranteed to satisfy monotonicity constraints. In this section, I describe a technique to convert a non-monotonic model to a monotonic one. The technique leverages monotonicity counterexamples to construct a monotonic *envelope* (or *hull*) of the learned model. Further, this technique is *online*: the monotonic envelope is constructed on-the-fly at prediction time.

As an example, consider the regression task of predicting house prices, which monotonically increase with the number of rooms. Suppose that the solid line (—) in Figure 4.1 plots the learned model's predictions. This function is not monotonic; for example $f(3) > f(4)$. The two dotted lines in Figure 4.1 show two monotonic envelopes that the technique produces: an *upper* envelope (·····) that increases the output where necessary to ensure monotonicity, and a *lower* envelope (- - -) that decreases the output where necessary to ensure monotonicity. The

rest of this section describes these envelopes formally and presents an empirical evaluation of the technique.

4.3.1 Envelope Construction

First, I describe envelope construction for the case with a single monotonic feature (with any number of other features) and then generalize the approach to handle multiple monotonic features.

4.3.1.1 Envelope - Single Monotonic Feature

Recall that Definition 4.2 in the previous section defines when a pair of inputs constitutes a monotonicity counterexample. To construct the envelope we require a special form of such counterexamples, namely *maximal* ones in terms of the degree of monotonicity violation, while fixing a single input example.

Definition 4.4. Consider example $x \in \mathcal{X}$, function $f : \mathcal{X} \to \mathcal{Y}$, and feature i. Let set $\mathcal{A}$ (resp. $\mathcal{B}$) consist of all examples x' such that (x, x') (resp. (x', x)) is a counterexample pair for f and i. Then, a *lower envelope counterexample* for example x, function f and feature i is an example $x' \in \mathcal{A}$ that minimizes $f(x')$. An *upper envelope counterexample* is an example $x' \in \mathcal{B}$ that maximizes $f(x')$.

For example, consider Figure 4.1 again. The upper envelope counterexample for input 3 is the input 2, since $f(2)$ has the maximal value of all counterexamples below 3. The lower envelope counterexample for the input 3 is 4, since $f(4)$ has the minimal value of all counterexamples above 3.

Now we can define the upper and lower envelopes of a function.

Definition 4.5. The *upper envelope* f_i^u of function $f : \mathcal{X} \to \mathcal{Y}$ for feature i is defined as follows:

$$f_i^u(x) = \begin{cases} f(x') & \text{where } x' \text{ is an upper envelope counterexample for } x,\ f, \text{ and } i \\ f(x\ ) & \text{if no such counterexample exists} \end{cases}$$

The *lower envelope* f_i^l is defined analogously.

I observe that it is not necessary to construct the envelope function explicitly. Rather, to ensure monotonicity, it suffices to construct the envelope incrementally at prediction time. Given an input x^t, we make a single query to an SMT solver to find the input's upper (lower) envelope counterexample or determine that no such counterexample exists. Note that this query is much simpler than would be required to verify that the original function is monotonic. Doing the latter would require searching for an arbitrary monotonicity counterexample pair (Definition 4.2), which is a pair of points. In contrast, our query is given the input x^t and hence only requires the SMT solver to search over the space of inputs that are identical to x^t except in the ith dimension. Concretely, for a feature i in the bounded interval $[L, U]$, the upper envelope search is over the interval $[L, x^t[i])$ and the lower envelope search is over the interval $(x^t[i], U]$. Empirically we will later show that our envelope construction is faster than querying for an arbitrary counterexample pair (see Figure 4.3).

4.3.1.2 Envelope - Multi-Dimensional Case

I now generalize the envelope construction to the case where multiple dimensions are monotonic. For space reasons, I present only the upper envelope construction; the lower envelope is analogous.

Recall from § 4.2 that, to verify if a function is monotonic in more than one dimension, it is sufficient to verify that it is monotonic in each dimension separately. However, to construct the envelope, it is not sufficient to identify maximal counterexamples in each dimension and

then take the maximum of these maxima. The envelopes produced using that approach are not guaranteed to be monotonic, which I now demonstrate with an example. Consider a function f that is intended to be monotonically increasing in its two input features. Now, consider the point $(3, 5)$, suppose that $(1, 5)$ and $(3, 3)$ are the upper envelope counterexamples in each dimension (Definition 4.4), and suppose that $f(3, 3) > f(1, 5)$ so we set $f^u_{\{0,1\}}(3, 5)$ $= f(3, 3)$. Now consider a second point $(7, 5)$, suppose that $(1, 5)$ and $(7, 2)$ are the upper envelope counterexamples in each dimension, and suppose that $f(1, 5) > f(7, 2)$ so we set $f^u_{\{0,1\}}(7, 5) = f(1, 5)$. Since $f(3, 3) > f(1, 5)$ we have that $f^u_{\{0,1\}}(3, 5) > f^u_{\{0,1\}}(7, 5)$, which violates monotonicity.

To overcome this problem, we generalize to multiple dimensions by searching *jointly* in all monotonic dimensions and prove that this approach is correct.

Definition 4.6. Consider example $x \in \mathcal{X}$, function $f : \mathcal{X} \to \mathcal{Y}$, and set of features S. Let set $\mathcal{B}$ consist of all examples x' such that $\forall i \in S$, $x'[i] \leq x[i]$ and $\forall i \notin S, x'[k] = x[k]$ and $f(x') > f(x)$. An *upper envelope counterexample* is an example $x' \in \mathcal{B}$ that maximizes $f(x')$.

It is easy to show that this approach does not identify spurious counterexamples: if an upper envelope counterexample exists for x and f and set of features S, then there is a dimension $i \in S$ and points x' and x'' such that x' and x'' are a monotonicity counterexample for f in the ith dimension.

I now define the upper envelope function, analogous to the single-dimensional case:

Definition 4.7. The *upper envelope* f^u_S of function $f : \mathcal{X} \to \mathcal{Y}$ for feature set S is defined as follows:

$$f^u_S(x) = \begin{cases} f(x') & \text{where } x' \text{ is an upper envelope counterexample for } x,\ f, \text{ and } S \\ f(x\ ) & \text{if no such counterexample exists} \end{cases}$$

Finally, I prove that the upper envelope is in fact monotonic, even when the function f is not.

Theorem 4.1. *For any function f and set of features S, the* upper envelope f^u_S *is monotonic in S.*

Proof. Let $i_0 \in S$ and x and x' be any two inputs such that $x[i_0] \leq x'[i_0]$ and $\forall k \neq i_0, x[k] = x'[k]$. I will prove that $f^u_S(x) \leq f^u_S(x')$ and hence that f^u_S is monotonic. There are two cases:

1. An input x'_e is the upper envelope counterexample for x', f, and S, so $f^u_S(x') = f(x'_e)$. We have two subcases.

   - An input x_e is the upper envelope counterexample for x, f, and S, so $f^u_S(x) = f(x_e)$. By Definition 4.6 we have that $\forall i \in S$, $x_e[i] \leq x[i] \land \forall i \notin S, x_e[k] = x[k]$, so also $\forall i \in S$, $x_e[i] \leq x'[i] \land \forall i \notin S, x_e[k] = x'[k]$. Therefore again by Definition 4.6 it must be the case that $f(x_e) \leq f(x'_e)$.
   - There is no upper envelope counterexample for x, f, and S, so $f^u_S(x) = f(x)$. Since $\forall i \in S$, $x[i] \leq x'[i] \land \forall i \notin S, x[k] = x'[k]$, by Definition 4.6 it must be the case that $f(x) \leq f(x'_e)$.

2. There is no upper envelope counterexample for x', f, and S, so $f^u_S(x') = f(x')$. We have two subcases.

   - An input x_e is the upper envelope counterexample for x, f, and S, so $f^u_S(x) = f(x_e)$. By Definition 4.6 we have that $\forall i \in S$, $x_e[i] \leq x[i] \land \forall i \notin S, x_e[k] = x[k]$, so also $\forall i \in S$, $x_e[i] \leq x'[i] \land \forall i \notin S, x_e[k] = x'[k]$. Therefore again by Definition 4.6 it must be the case that $f(x_e) \leq f(x')$, or else x' would have an upper envelope counterexample.
   - There is no upper envelope counterexample for x, f, and S, so $f^u_S(x) = f(x)$. Then again by Definition 4.6 it must be the case that $f(x) \leq f(x')$, or else x' would have an upper envelope counterexample. □

Hence, our envelope construction algorithm guarantees monotonicity of the predictive function, regardless of where it is evaluated, and regardless of the underlying learned function.

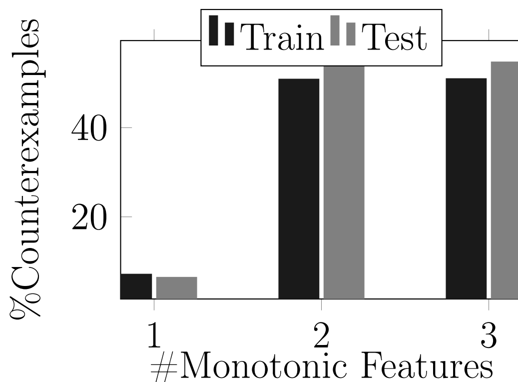

Figure 4.2: Empirically, the best learned baseline model is not monotonic. The figure presents the percentage of examples that have an upper or lower envelope counterexample for the *Auto MPG* dataset.

4.3.2 Empirical Evaluation of Monotonic Envelopes

I report the experimental results on the quality and performance of the envelope construction algorithm. Experiments were implemented in Python using the Keras deep learning library [Cho15], using the ADAM optimizer [KB14] to perform stochastic optimization of the neural network models, and Optimathsat [ST18] solver for counterexample generation.

Data and experiment setup: I use four datasets: *Auto MPG* and *Boston Housing* are regression datasets used for predicting miles per gallon (monotonically decreasing with respect to features *weight (W)*, *displacement (D)*, and *horsepower (HP)*) and housing prices (monotonically decreasing in *crime rate* and increasing in *number of rooms*) respectively and are obtained from the UCI machine learning repository [BM98]; *Heart Disease* [GLF89] and *Adult* [BM98] are classification datasets used for predicting the presence of heart disease (monotonically increasing with *trestbps (T)*, *cholestrol (C)*) and income level (monotonically increasing with *capital-gain* and *hours per week*) respectively. For each dataset, we identify the best baseline architecture and parameters by conducting grid search and learning the best ReLU neural network ($\mathrm{NN_b}$). I carry out our experiments on three random 80/20 splits and report average test results, except for the Adult dataset, for which we report on one random split.

Table 4.1: Best parameter configurations on each dataset for each data fold found using grid search for baseline neural networks (NN_b).

	Auto-MPG			Boston			Heart			Adult		
	Batch Size	# Epochs	LR	Batch Size	# Epochs	LR	Batch Size	# Epochs	LR	Batch Size	# Epochs	LR
0	32	2000	0.01	64	1000	0.01	32	400	0.01	1024	500	0.01
1	32	1500	0.01	64	1000	0.001	32	400	0.01	-	-	-
2	32	2000	0.01	32	500	0.01	32	400	0.001	-	-	-

System Specifications and Architecture Setup: All experiments were run on an Intel(R) Xeon(R) Gold 5220 CPU @ 2.20GHz CPU with 512GB of DDR3 RAM running Ubuntu 18.04.3 LTS with kernel 5.3.0-28-generic. For each dataset, we train five baseline architectures from a set of configurations and choose the best architecture based on train error. For *Boston Housing*, *Heart Diseases*, and *Adult* dataset, best baseline architecture includes three layers and 16 hidden neurons per layer. For *Auto MPG* dataset, best baseline architecture includes three layers and 12 hidden neurons per layer (see Table 4.1 for best baseline neural network parameters).

Q1. Is a deep neural network trained on such data monotonic? Figure 4.2 shows that the best baseline model (NN_b) is not monotonic, motivating the need for *envelope* predictions that guarantee monotonicity. The percentage of data points that have a counterexample can be as high as 50% for *Auto MPG*. See Table 4.2 for detailed results on all datasets, where the percentage can be as high as 98%.

Q2. When enforcing monotonicity using an envelope, does it come at a cost in terms of prediction quality? In this experiment, I compare the quality of the original model (NN_b) with its envelope on the test data. I select the envelope with the lowest train mean squared error (MSE) in case of regression and the highest train accuracy in case of classification. Table 4.3 demonstrates that an envelope can be used with a single or multiple monotonic features with little to no loss in prediction quality. In fact, in some cases (see rows in **bold**), the envelope has better average quality. This can be explained as follows: although the true data distribution is naturally monotonic, existing learning algorithms might

Table 4.2: Empirically, the best baseline neural network model (NN_b) trained on data is not monotonic. The table presents the percentage of examples that have an upper or lower envelope counterexample.

Dataset	**Feature**	**Train % CG**	**Test % CG**
Auto-MPG	Weight	7.11	6.41
	Displ.	48.62	52.99
	W,D	50.85	54.7
	W,D,HP	50.96	54.7
Boston Housing	Rooms	7.59	7.92
	Crime	16.75	16.5
Heart	Trestbps	73.14	74.86
	Chol.	86.91	87.98
	T,C	97.38	98.91
Adult	Cap. Gain	1.57	1.39
	Hours	18.93	19.58

Table 4.3: For regression (MSE, Left Table) and classification (Accuracy, Right Table) datasets, envelope predictions on test data have similar quality compared to baseline models. This means we can guarantee monotonic predictions with little to no loss in model quality.

Dataset	**Feature**	**NN_b**	**Envelope**
Auto-MPG	Weight	9.33±3.22	**9.19±3.41**
	Displ.	9.33±3.22	9.63±2.61
	W,D	9.33±3.22	9.63±2.61
	W,D,HP	9.33±3.22	9.63±2.61
Boston	Rooms	14.37±2.4	**14.19±2.28**
	Crime	14.37±2.4	**14.02±2.17**

Dataset	**Feature**	**NN_b**	**Envelope**
Heart	Trestbps	0.85±0.04	0.85±0.04
	Chol.	0.85±0.04	0.85±0.05
	T,C	0.85±0.04	0.85±0.05
Adult	Cap. Gain	0.84	0.84
	Hours	0.84	0.84

be missing simpler monotonic models and instead overfit a non-monotonic function because of noise in the training data.

Q3. How scalable is on-the-fly envelope construction? In this experiment, I report the run times for the *Auto MPG* dataset. Recall that the envelope approach need only search for maximal counterexamples relative to a given input. Owing to the narrowed search space, we see that envelope prediction time is comparable to the baseline model's prediction time in smaller models (see Figure 4.4). Overhead caused by envelope construction is only a *few seconds*. In contrast, the overhead to finding a maximal counterexample *pair* (Definition 4.2)

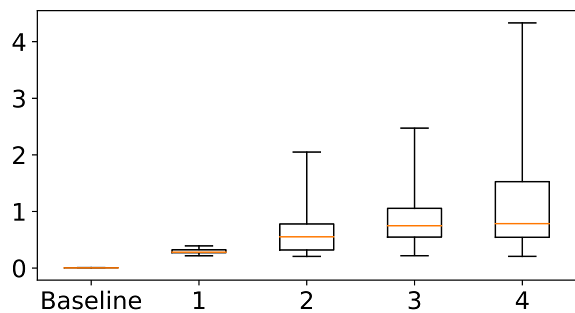

Figure 4.3: Prediction Time (s) vs. #Monotonic Features

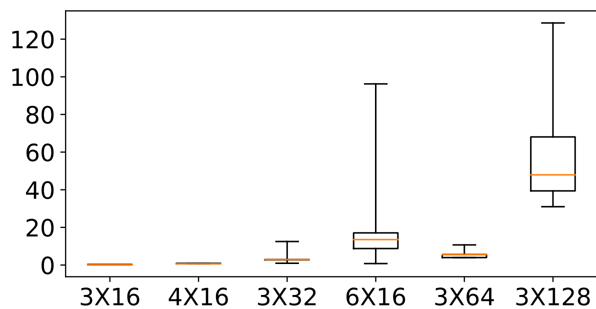

Figure 4.4: Prediction Time (s) vs. Model Size

for a single monotonic feature is 48.29 minutes. As a scalability study, in Figure 4.4, I plot the time taken to obtain a monotonic prediction for various model sizes. We can see that the envelope prediction time is comparable to the baseline prediction time in smaller models but grows with the model size. The main challenge for the OMT solver is in dealing with the ReLU piece, which leads to a worst-case complexity of NP-hard. Intuitively, the conditional (non-linearity) introduced by the ReLU may force the solver to explore both branches of the conditional. Therefore the set of possible paths to explore can grow exponentially with the number of available ReLUs. The growth is significantly less pronounced in the number of monotonic features (see Figure 4.3). Of course, when violating monotonicity leads to safety, ethical or legal problems, the question is not whether we can scale monotonicity enforcement, but whether it is safe to use machine learning at all. In this context, the computational price of enforcing monotonicity, even if it ends up being significant, is entirely warranted.

4.4 Counterexample-Guided Monotonicity Enforced Training

In this section, I propose an algorithm that uses monotonicity as an inductive bias during learning to improve model quality. This algorithm is orthogonal to the envelope prediction technique of the previous section; I evaluate the learning algorithm both on its own and in conjunction with the envelope technique.

4.4.1 Counterexample-guided Learning

The learning algorithm consists of two phases that alternate: the training phase and the verification phase. The training phase is given labeled input data and produces the best candidate model f. The verification phase checks if a given model is monotonic; if not, it generates one or more counterexamples, which are provided as additional data for the next iteration of the training phase. These two phases repeat for T epochs, which is a hyperparameter to the algorithm.

The algorithm is universal in the sense that it is compatible with any training technique that produces ReLU models and does not further restrict the hypothesis class. This gives our approach an advantage over prior monotonic learners [Sil98, YDC17].

The verification phase could use Definition 4.2 to identify monotonicity counterexamples, but this has two major drawbacks: (1) it is computationally expensive as the size of the pre-trained model grows; (2) an arbitrary counterexample might include out-of-distribution examples, which are therefore not representative. Hence, we instead appeal to Definition 4.6 to generate maximal counterexamples relative to each training point. In each epoch, for each train point we generate and use both *upper* and *lower envelope counterexamples* as additional data for the next round of training.

At this point, we are almost done with the algorithm, with the following detail to address. Counterexamples generated by the verification procedure do not have a known ground-truth label. There are different heuristics that one could adopt to label these points and encourage the learned function to become more monotonic. In our algorithm, for regression tasks, we calculate the average prediction values of upper and lower counterexamples and the given training point and assign this average as the label for these counterexamples and the training point. The hypothesis is that using the average value will result in a smoother loss with respect to monotonicity. For classification tasks, we assign each counterexample point the

same label as the corresponding training point. Empirically (see Table 4.6), I show that this labeling heuristic is sufficient to improve the model quality.

Data augmentation through counterexamples could cause drift in the model quality. The proposed approach guards against this in multiple ways. First, data augmentation with counterexamples is recomputed for each batch at every epoch. This ensures that: 1) an incorrect old counterexample does not burden the learning, and 2) learning incorporates multiple counterexamples at a time and so is less sensitive to any particular one. Second, the labeling heuristic for counterexamples provides a smoother loss with respect to monotonicity. Empirically (see Table 4.4), I show that there is no drift in the model quality. The quality of the re-trained model is similar to or better than a model trained without monotonicity constraints.

4.4.2 Empirical Evaluation of COMET

I will now evaluate the iterative algorithm for training with monotonicity counterexamples, as well as the entire COMET pipeline, which also includes the envelope technique from the previous section. We use the same datasets as in § 4.3.2.

Table 4.4: Monotonicity is an effective inductive bias. Counterexample-guided Learning (CGL) improves the quality of the baseline model in regression (MSE, Left Table) and classification (Accuracy, Right Table) datasets

Dataset	Feature	NN_b	CGL
Auto-MPG	Weight	9.33±3.22	**9.04±2.76**
	Displ.	9.33±3.22	**9.08±2.87**
	W,D	9.33±3.22	**8.86±2.67**
	W,D,HP	9.33±3.22	**8.63±2.21**
Boston	Rooms	14.37±2.4	**12.24±2.87**
	Crime	14.37±2.4	**11.66±2.89**

Dataset	Feature	NN_b	CGL
Heart	Trestbps	0.85±0.04	**0.86±0.02**
	Chol.	**0.85±0.04**	**0.85±0.05**
	T,C	0.85±0.04	**0.86±0.06**
Adult	Cap. Gain	**0.84**	**0.84**
	Hours	**0.84**	**0.84**

Q4. Is the stronger inductive bias of our learning algorithm able to improve the overall quality of the original non-monotonic model? In this experiment, I compare the test quality of the model learned with monotonicity counterexamples with the

original model (NN_b). From Table 4.4, we can see that monotonicity is indeed an effective inductive bias that helps improve the model quality. It is able to reduce the error on all regression datasets, with the biggest decrease from 14.37 to 11.66 for the *Boston Housing* dataset when employing monotonicity counterexamples based on the Crime Rate feature. Although the algorithm improves the quality, it does not guarantee monotonic predictions.

Table 4.5: Counterexample-guided learning (CGL) is able to make a model more monotonic by reducing the number of test and train counterexamples compared to the baseline model (NN_b). However, the algorithm is unable to guarantee monotonicity, motivating the need for monotonic *envelopes*.

Dataset	**Features**	**Train**		**Test**	
		$\mathbf{NN_b}$	**CGL**	$\mathbf{NN_b}$	**CGL**
Auto-MPG	Weight	22.33	11.33	5	2
	Displ.	139.67	37	37	10.33
	W,D	159.67	85.67	42.67	22.67
	W,D,HP	149.67	61.33	39.33	15
Boston	Rooms	30	15.67	8	6.33
	Crime	80	38.67	19	8
Heart	Trestbps	188.67	31	49	7
	Chol.	212.67	45.33	53	10.67
	T,C	235.67	169.67	60.33	40.33
Adult	Cap. Gain	7407	2755	1903	700
	Hours	379	0	84	0

Q5. Does our learning algorithm make the original non-monotonic model more monotonic? To quantify if a function is more monotonic, I calculate the reduction in the number of counterexamples. On average, the algorithm reduces the number of test counterexamples by 62%. Although in some cases we can remove all counterexamples, in general this is not the case (see Table 4.5). This motivates the need for using monotonic *envelopes* (described in § 4.3) in conjunction with the counterexample-guided learning algorithm, to guarantee monotonic predictions.

Table 4.6: For regression (MSE, Left Table) and classification (Accuracy, Right Table) datasets, counterexample-guided learning improves the envelope quality

Dataset	Features	NN_b Env.	`lfind`
Auto-MPG	Weight	9.19±3.41	**8.92±2.93**
	Displ.	9.63±2.61	**9.11±2.25**
	W,D	9.63±2.61	**8.89±2.29**
	W,D,HP	9.33±2.61	**8.81±1.81**
Boston	Rooms	14.19±2.28	**11.54±2.55**
	Crime	14.02±2.17	**11.07±2.99**

Dataset	Features	NN_b Env.	`lfind`
Heart	Trestbps	0.85±0.04	**0.86±0.03**
	Chol.	0.85±0.05	**0.87±0.03**
	T,C	0.85±0.05	**0.86±0.03**
Adult	Cap. Gain	**0.84**	**0.84**
	Hours	**0.84**	**0.84**

Q6. Does counterexample-guided learning help improve the quality of the original model's envelope? In § 4.3.2 Q2, (Table 4.3), I showed that the envelope has similar model quality compared to the baseline model. By additionally enforcing monotonicity constraints through counterexample-guided re-training, we further improve the envelope quality (Table 4.6). In this experiment I re-train NN_b with counterexamples for 40 epochs, model selection is based on train quality, and we report the change in the quality of the test envelope (see below for additional model selection experiments). Thus, we get both a monotonicity guarantee and better generalization performance.

Table 4.7: `COMET` outperforms Min-Max networks on all datasets. `COMET` outperforms DLN in regression datasets and achieves similar results in classification datasets.

Dataset	Features	Min-Max	DLN	`COMET`
Auto-MPG	Weight	9.91±1.20	16.77±2.57	**8.92±2.93**
	Displ.	11.78±2.20	16.67±2.25	**9.11±2.25**
	W,D	11.60±0.54	16.56±2.27	**8.89±2.29**
	W,D,HP	10.14±1.54	13.34±2.42	**8.81±1.81**
Boston	Rooms	30.88±13.78	15.93±1.40	**11.54±2.55**
	Crime	25.89±2.47	12.06±1.44	**11.07±2.99**

Dataset	Features	Min-Max	DLN	`COMET`
Heart	Trestbps	0.75±0.04	0.85±0.02	**0.86±0.03**
	Chol.	0.75±0.04	0.85±0.04	**0.87±0.03**
	T,C	0.75±0.04	**0.86±0.02**	**0.86±0.03**
Adult	Cap. Gain	0.77	**0.84**	**0.84**
	Hours	0.73	**0.85**	0.84

Additional model selection experiment. In § 4.4.2, model selection was based on minimum train error. In this experiment, I carry out model selection based on the least number of counterexamples. Overall, we find that monotonicity counterexamples act as a good inductive bias and improve model quality. However, there is a tradeoff on how much one could enforce monotonicity as a bias. Figure 4.5 plots test envelope MSE of *Auto MPG*

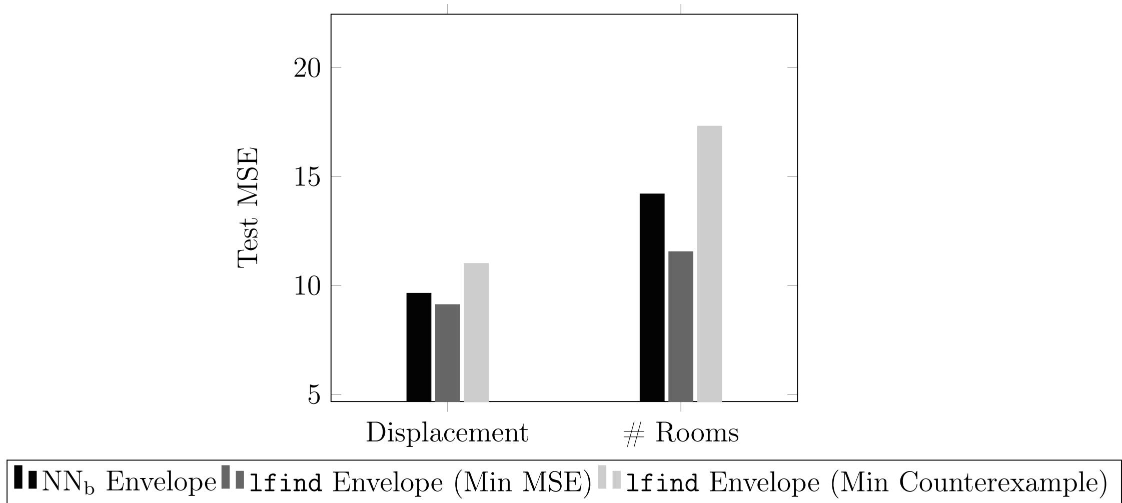

Figure 4.5: Monotonicity is a good inductive bias and helps in improving model accuracy. However, there is a tradeoff between performance and reducing the number of examples that have counterexamples.

and *Boston Housing* datasets. We can see that envelope construction on a function with minimum counterexamples has a higher error than the original model's envelope.

Q7. How does the performance of `COMET` compare to existing work? First I describe the Min-Max and Deep Lattice Network setup. Min-Max networks [DV10] proposes a fixed, feedforward three-layer (two hidden layer) architecture. The first layer computes different linear combinations of input that are partitioned into different groups. If increasing monotonicity is desired, then all weights connected to that input are constrained to be positive. Corresponding to each group, the second layer computes the maximum, and the final layer computes the minimum over all groups. For monotone features that are decreasing, we negate the feature to use the same architecture. The Deep Lattice Network [YDC17] architecture consists of six layers as proposed by the authors: calibrators, linear embedding, calibrators, ensemble of lattices, calibrators, and linear embedding. Note that for these approaches, for each dataset, I tune parameters separately for each combination of monotonic features at each fold using grid search; hence it is optimized for each monotone prediction task. However,

for `lfind` it is sufficient to tune parameters for the original neural network (NN_b) once per dataset.

Table 4.7 reports the MSE and accuracy of `COMET` compared to two existing methods that guarantee monotonicity: min-max networks [DV10] and deep lattice networks (DLN) [YDC17]. We tune Adam stepsize, learning rate, number of epochs, and batch size on all methods. Additionally, for DLN we tune calibration keypoints and report the results based on the six-layer architecture as proposed by the authors. The results in Table 4.7 indicate that `COMET` outperforms min-max networks on all datasets and DLN on all except for Adult, where we are similar.

Q8. How robust is `COMET` to data outliers? `COMET` constructs its monotonic envelope on the learned function and not on the data. Therefore, individual data outliers will not affect it too much. Moreover, if the function to be learned is naturally monotonic, enforcing invariants counteracts noise and outliers, leading to improved robustness. To illustrate this advantage, we duplicate 1% of the data and modify the value of the monotonic feature and the label for each new point in order to introduce monotonicity outliers (violations). For example, for an increasing monotonic feature, we reduce the label and increase the value of the monotonic feature. Table 4.8 shows that our approach produces *more robust models*, with `COMET` improving baseline model quality.

Table 4.8: With monotonicity data outliers, `lfind` produces models that are more robust than the baseline models (NN_b) for regression (MSE, Left Table) and classification (Accuracy, Right Table) datasets.

Dataset	Features	NN_b	`lfind`
Auto-MPG	Weight	13.54±4.65	**10.50±1.87**
	Displ.	12.00±2.94	**10.34±1.25**
	W,D	15.35±2.30	**13.84±3.09**
	W,D,HP	10.26±2.19	**9.48±1.29**
Boston	Rooms	12.79±3.88	**10.23±1.95**
	Crime	21.13±4.41	**19.20±6.64**

Dataset	Features	NN_b	`lfind`
Heart	Trestbps	0.77±0.07	**0.78±0.07**
	Chol.	**0.77±0.06**	**0.77±0.06**
	T,C	0.77±0.06	**0.81±0.03**
Adult	Cap. Gain	**0.82**	**0.82**
	Hours	**0.82**	**0.82**

4.5 Related Work

Monotonic Networks. Liu et al. [LHZ20] propose a concurrent work that uses verification techniques to learn certified monotonic networks. The approach encodes an arbitrary ReLU neural network using mixed-integer linear programming and solves an optimization problem to verify monotonicity. The optimization problem is to identify if the minimum derivative of the function is non-negative. Further, the approach learns monotonic networks by training with heuristic monotonicity regularizations and gradually increasing the regularization magnitude until it passes the monotonicity verification. We differ from this work in two ways. First, our envelope technique produces a monotonic version of an arbitrary ReLU neural network without having to retrain it. Second, we solve an optimization problem to identify the maximum violation for a given point, which is necessary for the envelope construction.

Other related work in this area can be categorized into algorithms that (1) guarantee monotonicity by restricting the hypothesis space, or (2) incorporate monotonicity during learning without any guarantees. In the first category, Archer and Wang [AW93] propose a monotone model by constraining the neural net weights to be positive. Other methods enforce constraints on model weights [DK99, Wan94, MVL10, DBB09, AXK17] and force the derivative of the output to be strictly positive [WL19]. Monotonic networks [Sil98] guarantee monotonicity by constructing a three-layer network using monotonic linear embedding and max-min-pooling. Daniels and Velikova [DV10] generalized this approach to handle functions that are partially monotonic. Deep lattice networks (DLN) [YDC17] proposed a combination of linear calibrators and lattices for learning monotonic functions. Lattices are structurally rigid thereby restricting the hypothesis space significantly. Our envelope technique is similar to these works in that it guarantees monotonicity. However, it does so at prediction time and can do so for any ReLU neural networks, without needing to restrict the hypothesis space further. Finally, isotonic regression [BB72, SS97] requires the training data to be partially

ordered, which is unlikely to happen; in general input points over many features are not partially ordered.

In the second category, monotonicity can be incorporated in the learning process by modifying the loss function or by adding additional data. Monotonicity Hints [SA97] proposes a modified loss function that penalizes non-monotonicity of the model. The algorithm models the input distribution as a joint Gaussian estimated from the training data and samples random pairs of monotonic points that are added to the training data. Gupta et. al. [GSM19] introduce a point-wise loss function that acts as a soft monotonicity constraint. Our approach is similar to these works in that it adds additional data to enforce monotonicity. However, `COMET`'s counterexample-guided learning and envelope technique together guarantee monotonicity, while these works provide no such guarantees. In addition, unlike prior work, we look beyond the neighborhood of a training point by identifying maximal violations. Other works enforce monotonicity to accelerate learning of probabilistic models in data-scarce and knowledge-rich domains [ON18, ARD12, YN13]. Similarly, these works fail to provide any formal guarantee for the learned model.

Verification of Neural Networks and Adversarial Learning. Reluplex [KBD17], an augmented SMT solver, verifies properties of networks with ReLU activation functions. Huang et. al. [HKW17] leverage SMT for verification of safety properties by discretizing the continuous region around an input and show that there are no counterexamples. Our approach leverages the SMT encodings of neural networks from this prior work but uses them only to obtain counterexamples rather than for verification. Recently, many approaches propose adversarially robust algorithms which can be divided into empirical [KGB16, MMS17, GSS14, GR14] and certified defenses [WK18, SND18, RSL18, HA17, SLR19, SGM18, GMD18]. We are closely related to these works, in that we carry out adversarial training using counterexamples. However, we differ in two ways. First, to the best of our knowledge, there is no related work in the adversarial robustness literature for ensuring monotonicity. Second, related work in adversarial training only ensures correctness in the neighborhood of a training point, while

we globally search for a counterexample and are able to discover long-range monotonicity violations. Counterexample-driven learning has also been used to enforce fairness constraints on Bayesian classifiers [CFB20].

4.6 Summary

I presented two algorithms that incorporate monotonicity constraints into neural networks: counterexample-guided prediction that guarantees monotonicity and counterexample-guided training that enforces monotonicity as an inductive bias. I demonstrate the effectiveness of these techniques on regression and classification tasks.

4.7 Extensions and Future Work

4.7.1 Fairness

In this chapter, I have explored provable guarantees for monotonicity. However, machine learning models can benefit from other kinds of inductive bias, such as those coming from algorithmic fairness. The FETA work by Mohammadi *et al.* [MSF22] utilizes key ideas from `COMET` to enforce and guarantee individual fairness. Specifically, to enforce fairness, FETA reuses the counterexample-guided retraining approach of `COMET` and uses fairness counterexamples instead of monotonicity counterexamples. Additionally, FETA also proposes a counterexample-guided online technique to provably enforce fairness constraints at prediction time. Empirical evaluation of FETA indicates that it is able to guarantee fairness on-the-fly at prediction time and is able to train accurate models exhibiting a much higher degree of individual fairness.

4.7.2 Scalability

In this chapter, I have shown that enforcing monotonicity constraints can improve model quality while providing provable guarantees. Unfortunately, the approach is limited in its applicability by the size of the network and the network architecture. `COMET` uses an SMT solver to encode the property of interest and to obtain a counterexample. While these solvers are complete (i.e., do not have any false positives), it can be challenging to scale these to large networks. Future work can address this limitation may be using specialized SMT solvers, or by using MaxSMT solvers instead of OMT for finding maximal counterexamples.

CHAPTER 5

Conclusion

"Have you tried turning it off, then on?" is now a common adage due to the ubiquitous nature of software and how it has firmly intertwined with our everyday lives. But this adage is not always applicable — coffee-makers can be restarted mid-operation, but what about autonomous vehicles? When we take a step back and think about this, it seems like software has a tendency to creep into unexpected states more often than desired, and resetting a system is one way to bring it back to a known-good state. These systems are complex, so they are fragile and suffer from design and implementation flaws that make them unreliable. In recent years, software bugs have become more than a mere annoyance, they have had a major impact on the economy and society. Examples include the Log4j vulnerability in Java [log], the Heartbleed vulnerability in Openssl [DLK14], and in 2011 Jerome Radcliffe [Rad11] showed they could wirelessly hack an insulin pump and cause it to deliver incorrect dosages of medication.

"Why should systems land at unknown states in the first place when we have perfectly good program verification techniques that can ensure the reliability of a given system at all times?" - This guileless question has been bothering me ever since I encountered a large system with recurring bugs transacting in billions of dollars during my first job straight out of undergrad. Traditional program verification techniques are being proposed regularly but they tend to work well only on small programs or require significant manual work. When exploring this problem, I realized that full automation for reliability is a very hard goal. We need new ways to harness human intuition while automating parts of the verification process

that machines are good at. To this end, Chapters 2 and 3 of this dissertation present my contributions to the reduction of the manual effort required by human-in-the-loop verification approaches, by proposing semi-automated techniques to guarantee system reliability.

While investigating recent automation available for human-in-the-loop verification a.k.a foundational verification, I found that there are several tactic-based automation approaches that search for a proof script given a set of library lemmas. While this is useful, these approaches fail when the set of lemmas is incomplete. If a required lemma is not available as prior input, these approaches cannot successfully identify a valid proof script. In fact, providing the right set of lemmas is an onerous manual effort. This motivated me to focus on lemma synthesis. Unfortunately, blind enumeration and filtering to identify the correct set of candidate lemmas do not scale well. Instead, I needed to develop techniques that were directed and used information from the proof context to guide the candidate lemma search. The idea to use examples as a specification to better guide the search was immediate, however, identifying a formal synthesis model that worked well took significant effort. This data-driven synthesis approach is presented in Chapter 2. I also had to solve several technical challenges w.r.t to example generation and synthesis of candidates, which are detailed in the chapter.

While this was a good first exploration, I quickly realized that the synthesis setup performs well only on certain classes of helper lemmas, namely those that share the same root structure as the goal state. However, many natural lemmas can contain helper lemmas that do not share the root structure. While this looks like a search-space explosion, I was able to get around this by looking into how helper lemmas are typically used. This key insight helped me easily extend and reuse the same synthesis problem as `lfind` to cater to lemmas that are used to rewrite a subterm in a goal state. Additionally, I observed that helper lemmas can be conditional. The first look at this problem made me think that we need an elaborate boolean program synthesis algorithm that can synthesize these conditions. Once again this looked like a computationally expensive process and we would need good heuristics to arrive

at the correct conditional predicate for the lemma. However, after perusing several real-world examples it became apparent that, in the common case, the required hypotheses are present in the goal state. This insight helped me develop the counterexample-guided refinement approach presented in Chapter 3. I believe the proposed data-driven angle to lemma synthesis can serve as the basis of future work in the area of proof automation. I have listed two such directions in § 3.6.

Just as software systems took over the world, we are now at the cusp of the stage when Machine Learning is rapidly proliferating in our lives. Machine Learning has been touted as the panacea to many of humanity's current problems. But it remains a black box, with seemingly insurmountable hard challenges such as privacy, verifiability, explainability, and fairness. Similar to software systems, the failure of machine learning systems has produced catastrophic results. For example, autonomous vehicles have been involved in numerous fatal crashes despite very high test and training accuracy.

I was able to appreciate the benefits of machine learning since it can flexibly learn from all sorts of data with minimal inductive bias. But this also means that we lack any semblance of behavioral formal guarantees. To avoid catastrophic outcomes, there is an urgent need for research into model behavior guarantees. I was motivated by this goal and wanted to apply existing formal method techniques to machine learning models as they have been widely successful in providing strong guarantees in software systems. Formal properties of functions are often characterized in terms of their counterexamples. Counterexample-guided algorithms are prevalent in the field of formal methods, for example to verify and synthesize programs. Therefore, I decided to develop a counterexample-guided algorithm that can enforce a given property (e.g. monotonicity) as an inductive bias during training time. While this was straightforward, it took significant effort to provide provable guarantees that a learned model is monotonic for all points in the input domain. These techniques are presented in Chapter 4. Although `COMET` works well for monotonicity constraints, I believe the proposed techniques can serve as the basis for future work for other properties like fairness, submodularity, etc.

REFERENCES

[ACE16] Alex A Alemi, François Chollet, Niklas Eén, Geoffrey Irving, Christian Szegedy, and Josef Urban. "DeepMath-deep sequence models for premise selection." *arXiv preprint arXiv:1606.04442*, 2016.

[add18] "Additions." https://github.com/coq-contribs/additions, 2018.

[AGK13] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. "Recursive program synthesis." In *International Conference on Computer Aided Verification*, pp. 934–950. Springer, 2013.

[AMS19] Angello Astorga, P Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. "Learning stateful preconditions modulo a test generator." In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 775–787, 2019.

[AOS16] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. "Concrete problems in AI safety." *arXiv preprint arXiv:1606.06565*, 2016.

[App15] Andrew W Appel. "Verification of a cryptographic primitive: SHA-256." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **37**(2):1–31, 2015.

[ARD12] Eric E Altendorf, Angelo C Restificar, and Thomas G Dietterich. "Learning from sparse data by exploiting monotonicity constraints." *arXiv preprint arXiv:1207.1364*, 2012.

[Aub76] R Aubin. "Mechanising Structural Induction." 1976.

[AW93] Norman P. Archer and Shouhong Wang. "Learning bias in neural networks and an approach to controlling its effect in monotonic classification." *IEEE transactions on pattern analysis and machine intelligence*, **15**(9):962–966, 1993.

[AXK17] Brandon Amos, Lei Xu, and J Zico Kolter. "Input convex neural networks." In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 146–155. JMLR. org, 2017.

[BB72] Richard E Barlow and Hugh D Brunk. "The isotonic regression problem and its dual." *Journal of the American Statistical Association*, **67**(337):140–147, 1972.

[BBC97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. "The Coq proof assistant reference manual: Version 6.1." 1997.

[BBP11] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. "Extending Sledgehammer with SMT Solvers." In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, pp. 116–130, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[BDD16] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. "End to end learning for self-driving cars." *arXiv preprint arXiv:1604.07316*, 2016.

[BH14] Bernhard Beckert and Reiner Hähnle. "Reasoning and verification: State of the art and current trends." *IEEE Intelligent Systems*, **29**(1):20–29, 2014.

[BKP16] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C Paulson, and Josef Urban. "Hammering towards QED." *Journal of Formalized Reasoning*, **9**(1):101–148, 2016.

[BLR19a] Kshitij Bansal, Sarah M Loos, Markus N Rabe, Christian Szegedy, and Stewart Wilcox. "HOList: An environment for machine learning of higher-order theorem proving." *arXiv preprint arXiv:1904.03241*, 2019.

[BLR19b] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. "HOList: An Environment for Machine Learning of Higher-Order Theorem Proving (extended version)." *CoRR*, **abs/1904.03241**, 2019.

[BM79] Robert S Boyer and J Strother Moore. "A Computational Logic." *ACM Monograph Series*, 1979.

[BM98] Catherine L Blake and Christopher J Merz. "UCI repository of machine learning databases, 1998.", 1998.

[BSV93] Alan Bundy, Andrew Stevens, Frank Van Harmelen, Andrew Ireland, and Alan Smaill. "Rippling: A heuristic for guiding inductive proofs." *Artificial intelligence*, **62**(2):185–253, 1993.

[BT18] Clark Barrett and Cesare Tinelli. "Satisfiability modulo theories." In *Handbook of Model Checking*, pp. 305–343. Springer, 2018.

[Cas85] Jacqueline Castaing. "How to Facilitate the Proof of Theorems by Using the Induction-matching, and by Generalization." In *IJCAI*, 1985.

[CCK17] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. "Verifying a high-performance crash-safe file system using a tree specification." In *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 270–286, 2017.

[CDK11] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. "Integrating Testing and Interactive Theorem Proving." In David Hardin and Julien Schmaltz, editors, *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2011, Austin, Texas, USA, November 3-4, 2011*, volume 70 of *EPTCS*, pp. 4–19, 2011.

[CFB20] YooJung Choi, Golnoosh Farnadi, Behrouz Babaki, and Guy Van den Broeck. "Learning Fair Naive Bayes Classifiers by Discovering and Eliminating Discrimination Patterns." In *Proceedings of the 34th AAAI Conference on Artificial Intelligence*, 2020.

[CGJ00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided abstraction refinement." In *International Conference on Computer Aided Verification*, pp. 154–169. Springer, 2000.

[CH00] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs." In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, ICFP, pp. 268–279. ACM, 2000.

[Chl13a] Adam Chlipala. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant.* MIT Press, 2013.

[Chl13b] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press, 2013.

[Cho15] François Chollet et al. "Keras." `https://github.com/keras-team/keras`, 2015.

[cir95] "Circuits." `https://github.com/coq-contribs/circuits`, 1995.

[CJR13] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. "Automating Inductive Proofs Using Theory Exploration." In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pp. 392–406. Springer, 2013.

[CK18a] Łukasz Czajka and Cezary Kaliszyk. "Hammer for Coq: Automation for dependent type theory." *Journal of automated reasoning*, **61**(1-4):423–453, 2018.

[CK18b] Łukasz Czajka and Cezary Kaliszyk. "Hammer for Coq: Automation for Dependent Type Theory." *Journal of Automated Reasoning*, **61**(1):423–453, Jun 2018.

[coq03] "Coq-of-Ocaml." `https://github.com/foobar-land/coq-of-ocaml`, 2003.

[CSH10] Koen Claessen, Nicholas Smallbone, and John Hughes. "QuickSpec: Guessing Formal Specifications Using Testing." In Gordon Fraser 0001 and Angelo Gargantini, editors, *TAP@TOOLS*, volume 6143 of *Lecture Notes in Computer Science*, pp. 6–21. Springer, 2010.

[DBB01] Charles Dugas, Yoshua Bengio, François Bélisle, Claude Nadeau, and René Garcia. "Incorporating second-order functional knowledge for better option pricing." In *Advances in neural information processing systems*, pp. 472–478, 2001.

[DBB09] Charles Dugas, Yoshua Bengio, François Bélisle, Claude Nadeau, and René Garcia. "Incorporating functional knowledge in neural networks." *Journal of Machine Learning Research*, **10**(Jun):1239–1262, 2009.

[DF03] Lucas Dixon and Jacques Fleuriot. "IsaPlanner: A prototype proof planner in Isabelle." In *International Conference on Automated Deduction*, pp. 279–283. Springer, 2003.

[DJS18] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. "Output Range Analysis for Deep Feedforward Neural Networks." In *NASA Formal Methods Symposium*, pp. 121–138. Springer, 2018.

[DK99] Hennie Daniels and B Kamp. "Application of MLP networks to bond rating and house pricing." *Neural Computing & Applications*, **8**(3):226–234, 1999.

[DKW08] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. "A survey of automated techniques for formal software verification." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **27**(7):1165–1178, 2008.

[DLK14] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. "The matter of heartbleed." In *Proceedings of the 2014 conference on internet measurement conference*, pp. 475–488, 2014.

[DV10] Hennie Daniels and Marina Velikova. "Monotone and partially monotone neural networks." *IEEE Transactions on Neural Networks*, **21**(6):906–917, 2010.

[Ehl17] Ruediger Ehlers. "Formal verification of piece-wise linear feed-forward neural networks." In *International Symposium on Automated Technology for Verification and Analysis*, pp. 269–286. Springer, 2017.

[END18] P Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P Madhusudan. "Horn-ICE learning for synthesizing invariants and contracts." *Proceedings of the ACM on Programming Languages*, **2**(OOPSLA):1–25, 2018.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[FBG20a] Emily First, Yuriy Brun, and Arjun Guha. "TacTok: semantics-aware proof synthesis." *Proceedings of the ACM on Programming Languages*, **4**(OOPSLA):1–31, 2020.

[FBG20b] Emily First, Yuriy Brun, and Arjun Guha. "TacTok: Semantics-Aware Proof Synthesis." In *Object-oriented Programming, Systems, Languages, and Applications*, 10 2020.

[FCC16] Mahdi Milani Fard, Kevin Canini, Andrew Cotter, Jan Pfeifer, and Maya Gupta. "Fast and flexible monotonic functions with ensembles of lattices." In *Advances in Neural Information Processing Systems*, pp. 2919–2927, 2016.

[FCD15] John K. Feser, Swarat Chaudhuri, and Isil Dillig. "Synthesizing Data Structure Transformations from Input-Output Examples." *SIGPLAN Not.*, **50**(6):229–239, jun 2015.

[FHB97] Jean-Christophe Filliâtre, Hugo Herbelin, Bruno Barras, Bruno Barras, Samuel Boutin, Eduardo Giménez, Samuel Boutin, Gérard Huet, César Muñoz, Cristina Cornes, Cristina Cornes, Judicaël Courant, Judicael Courant, Chetan Murthy, Chetan Murthy, Catherine Parent, Catherine Parent, Christine Paulin-mohring, Christine Paulin-mohring, Amokrane Saibi, Amokrane Saibi, Benjamin Werner, and Benjamin Werner. "The Coq Proof Assistant - Reference Manual Version 6.1." Technical report, 1997.

[FNP18] Zhe Feng, Harikrishna Narasimhan, and David C Parkes. "Deep learning for revenue-optimal auctions with budgets." In *Proceedings of the 17th International Conference on Autonomous Agents and Multiagent Systems*, pp. 354–362. International Foundation for Autonomous Agents and Multiagent Systems, 2018.

[FOW16] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. "Example-Directed Synthesis: A Type-Theoretic Interpretation." *SIGPLAN Not.*, **51**(1):802–815, jan 2016.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks." In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323, 2011.

[GBC16] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[GCP16] Maya Gupta, Andrew Cotter, Jan Pfeifer, Konstantin Voevodski, Kevin Canini, Alexander Mangylov, Wojciech Moczydlowski, and Alexander Van Esbroeck. "Monotonic calibrated interpolated look-up tables." *The Journal of Machine Learning Research*, **17**(1):3790–3836, 2016.

[GDS18] Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy Mann, and Pushmeet Kohli. "On the effectiveness of interval bound propagation for training verifiably robust models." *arXiv preprint arXiv:1810.12715*, 2018.

[GKU17] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. “TacticToe: Learning to Reason with HOL4 Tactics.” In Thomas Eiter and David Sands, editors, *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pp. 125–143. EasyChair, 2017.

[GKU18] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. “Learning to Reason with HOL4 tactics.” *arXiv preprint arXiv:1804.00595*, 2018.

[GLF89] John H. Gennari, Pat Langley, and Douglas H. Fisher. “Models of Incremental Concept Formation.” *Artif. Intell.*, **40**(1-3):11–61, 1989.

[GLM14] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. “ICE: A robust framework for learning invariants.” In *International Conference on Computer Aided Verification*, pp. 69–87. Springer, 2014.

[GMD18] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. “Ai2: Safety and robustness certification of neural networks with abstract interpretation.” In *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 3–18. IEEE, 2018.

[GNM16] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. “Learning invariants using decision trees and implication counterexamples.” In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pp. 499–512. ACM, 2016.

[GPP20] Emilio Jesús Gallego Arias, Karl Palmskog, and Vasily Pestun. “SerAPI:Machine-Friendly, Data-Centric Serialization for Coq.” https://github.com/ejgallego/coq-serapi, 2020.

[GR14] Shixiang Gu and Luca Rigazio. “Towards deep neural network architectures robust to adversarial examples.” *arXiv preprint arXiv:1412.5068*, 2014.

[GSM19] Akhil Gupta, Naman Shukla, Lavanya Marla, Arinbjörn Kolbeinsson, and Kartik Yellepeddi. “How to Incorporate Monotonicity in Deep Networks While Preserving Flexibility?” *arXiv preprint arXiv:1909.10662*, 2019.

[GSS14] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. “Explaining and harnessing adversarial examples.” *arXiv preprint arXiv:1412.6572*, 2014.

[GSS18] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. “Explaining and harnessing adversarial examples. arXiv.”, 2018.

[HA17] Matthias Hein and Maksym Andriushchenko. "Formal guarantees on the robustness of a classifier against adversarial manipulation." In *Advances in Neural Information Processing Systems*, pp. 2266–2276, 2017.

[har96] "Hardware." `https://github.com/coq-contribs/hardware`, 1996.

[HDS19] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. "GamePad: A Learning Environment for Theorem Proving." In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[Hes92] Jane Thurmann Hesketh. *Using Middle-Out Reasoning to Guide Inductive Theorem Proving*. PhD thesis, University of Edinburgh, 1992.

[HHK15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. "IronFleet: proving practical distributed systems correct." In *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 1–17, 2015.

[HKW17] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. "Safety verification of deep neural networks." In *International Conference on Computer Aided Verification*, pp. 3–29. Springer, 2017.

[HPS16] Moritz Hardt, Eric Price, and Nati Srebro. "Equality of opportunity in supervised learning." In *Advances in neural information processing systems*, pp. 3315–3323, 2016.

[Hum90] B Hummel. "Generation of induction axioms and generalisation." 1990.

[IB96] Andrew Ireland and Alan Bundy. "Productive use of failure in inductive proof." In *Automated Mathematical Induction*, pp. 79–111. Springer, 1996.

[JDB10] Moa Johansson, Lucas Dixon, and Alan Bundy. "Dynamic Rippling, Middle-Out Reasoning and Lemma Discovery." In Simon Siegler and Nathan Wasser, editors, *Verification, Induction, Termination Analysis - Festschrift for Christoph Walther on the Occasion of His 60th Birthday*, volume 6463 of *Lecture Notes in Computer Science*, pp. 102–116. Springer, 2010.

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980*, 2014.

[KBD17] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. "Reluplex: An efficient SMT solver for verifying deep neural networks." In *International Conference on Computer Aided Verification*, pp. 97–117. Springer, 2017.

[KEH09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. "seL4: Formal verification of an OS kernel." In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 207–220, 2009.

[KGB16] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. "Adversarial machine learning at scale." *arXiv preprint arXiv:1611.01236*, 2016.

[KM97] Matt Kaufmann and J S. Moore. "An Industrial Strength Theorem Prover for a Logic Based on Common Lisp." *IEEE Transactions on Software Engineering*, **23**(4):203–213, April 1997.

[ACL2 is a successor to the Boyer-Moore prover. 42 references.]

[KS96] Deepak Kapur and Mahadevan Subramaniam. "Lemma Discovery in Automated Induction." In Michael A. McRobbie and John K. Slaney, editors, *Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings*, volume 1104 of *Lecture Notes in Computer Science*, pp. 538–552. Springer, 1996.

[KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks." *Communications of the ACM*, **60**(6):84–90, 2017.

[KU15a] Cezary Kaliszyk and Josef Urban. "HOL (y) Hammer: Online ATP service for HOL Light." *Mathematics in Computer Science*, **9**(1):5–22, 2015.

[KU15b] Cezary Kaliszyk and Josef Urban. "HOL(y)Hammer: Online ATP Service for HOL Light." *Mathematics in Computer Science*, **9**(1):5–22, Mar 2015.

[KU15c] Cezary Kaliszyk and Josef Urban. "MizAR 40 for Mizar 40." *Journal of Automated Reasoning*, **55**(3):245–256, Oct 2015.

[KUM18] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. "Reinforcement learning of theorem proving." *Advances in Neural Information Processing Systems*, **31**, 2018.

[LCO20] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. "Program Sketching with Live Bidirectional Evaluation." *Proc. ACM Program. Lang.*, **4**(ICFP), aug 2020.

[Ler09] Xavier Leroy. "Formal verification of a realistic compiler." *Communications of the ACM*, **52**(7):107–115, 2009.

[LHZ20] Xingchao Liu, Xing Han, Na Zhang, and Qiang Liu. "Certified Monotonic Neural Networks." In *Advances in Neural Information Processing Systems*, 2020.

[log] "Log4j – Apache Log4j Security Vulnerabilities." https://logging.apache.org/log4j/2.x/security.html.

[MKA15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. "The Lean Theorem Prover (System Description)." In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pp. 378–388, Cham, 2015. Springer International Publishing.

[MMS10] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. "Toward a verified relational database management system." In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 237–248, 2010.

[MMS17] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. "Towards deep learning models resistant to adversarial attacks." *arXiv preprint arXiv:1706.06083*, 2017.

[MNB22] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. "Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution." *Proc. ACM Program. Lang.*, **6**(POPL), jan 2022.

[MPW20] Anders Miltner, Saswat Padhi, David Walker, and Todd Millstein. "Data-driven inference of representation invariants." In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2020.

[MSF22] Kiarash Mohammadi, Aishwarya Sivaraman, and Golnoosh Farnadi. "FETA: Fairness Enforced Verifying, Training, and Predicting Algorithms for Neural Networks." *arXiv preprint arXiv:2206.00553*, 2022.

[MVL10] Alexey Minin, Marina Velikova, Bernhard Lang, and Hennie Daniels. "Comparison of universal approximators incorporating partial monotonicity by structure." *Neural Networks*, **23**(4):471–475, 2010.

[ON18] Phillip Odom and Sriraam Natarajan. "Human-guided learning for probabilistic logic models." *Frontiers in Robotics and AI*, **5**:56, 2018.

[OZ15] Peter-Michael Osera and Steve Zdancewic. "Type-and-example-directed program synthesis." *ACM SIGPLAN Notices*, **50**(6):619–630, 2015.

[Pau86] Lawrence C Paulson. "Natural deduction as higher-order resolution." *The Journal of Logic Programming*, **3**(3):237–258, 1986.

[Pau93] Lawrence C. Paulson. "Natural Deduction as Higher-Order Resolution." *CoRR*, **cs.LO/9301104**, 1993.

[PHD15] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin Pierce. "Foundational Property-Based Testing." volume 9236, 08 2015.

[PLR20] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. "Graph Representations for Higher-Order Logic and Theorem Proving." In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pp. 2967–2974. AAAI Press, 2020.

[PMG17] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. "Practical black-box attacks against machine learning." In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 506–519. ACM, 2017.

[PMJ16] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. "The limitations of deep learning in adversarial settings." In *2016 IEEE European symposium on security and privacy (EuroS&P)*, pp. 372–387. IEEE, 2016.

[PSM16] Saswat Padhi, Rahul Sharma, and Todd Millstein. "Data-driven precondition inference with learned features." In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 42–56. ACM, 2016.

[Rad11] Jerome Radcliffe. "Hacking medical devices for fun and insulin: Breaking the human SCADA system." In *Black Hat Conference presentation slides*, volume 2011, 2011.

[RDT15] Stuart Russell, Daniel Dewey, and Max Tegmark. "Research priorities for robust and beneficial artificial intelligence." *Ai Magazine*, **36**(4):105–114, 2015.

[RK15] Andrew Reynolds and Viktor Kuncak. "Induction for SMT Solvers." In Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen, editors, *VMCAI*, volume 8931 of *Lecture Notes in Computer Science*, pp. 80–98. Springer, 2015.

[RSL18] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. "Certified defenses against adversarial examples." *arXiv preprint arXiv:1801.09344*, 2018.

[SA97] Joseph Sill and Yaser S Abu-Mostafa. "Monotonicity hints." In *Advances in neural information processing systems*, pp. 634–640, 1997.

[SAS20a] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. "Generating correctness proofs with neural networks." In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 1–10, 2020.

[SAS20b] Alex Sanchez-Stern, Yousef Alhessi, Lawrence K. Saul, and Sorin Lerner. "Generating correctness proofs with neural networks." In Koushik Sen and Mayur Naik, editors, *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2020, London, UK, June 15, 2020*, pp. 1–10. ACM, 2020.

[SDE12] William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. "Zeno: An automated prover for properties of recursive data structures." In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 407–421. Springer, 2012.

[SGM18] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. "Fast and effective robustness certification." In *Advances in Neural Information Processing Systems*, pp. 10802–10813, 2018.

[Sil98] Joseph Sill. "Monotonic networks." In *Advances in neural information processing systems*, pp. 661–667, 1998.

[SIS17] Taro Sekiyama, Akifumi Imanishi, and Kohei Suenaga. "Towards Proof Synthesis Guided by Neural Machine Translation for Intuitionistic Propositional Logic." *CoRR*, **abs/1706.06462**, 2017.

[SLR19] Hadi Salman, Jerry Li, Ilya Razenshteyn, Pengchuan Zhang, Huan Zhang, Sebastien Bubeck, and Greg Yang. "Provably robust deep learning via adversarially trained smoothed classifiers." In *Advances in Neural Information Processing Systems*, pp. 11289–11300, 2019.

[SND18] Aman Sinha, Hongseok Namkoong, and John Duchi. "Certifiable distributional robustness with principled adversarial training." *In International Conference on Learning Representations*, 2018.

[SNV17] Aman Sinha, Hongseok Namkoong, Riccardo Volpi, and John Duchi. "Certifying some distributional robustness with principled adversarial training." *arXiv preprint arXiv:1710.10571*, 2017.

[Sol09] Armando Solar-Lezama. "The Sketching Approach to Program Synthesis." In Zhenjiang Hu, editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pp. 4–13. Springer, 2009.

[SPD19] Marcelo Carvalho dos Santos, Victor Henrique Cabral Pinheiro, Filipe Santana Moreira do Desterro, Renato Koga de Avellar, Roberto Schirru, Andressa dos Santos Nicolau, and Alan Miranda Monteiro de Lima. "Deep rectifier neural network applied to the accident identification problem in a PWR nuclear power plant." *Annals of Nuclear Energy*, **133**:400–408, 2019.

[SS97] Michael J Schell and Bahadur Singh. "The reduced monotonic regression method." *Journal of the American Statistical Association*, **92**(437):128–135, 1997.

[ST15] Roberto Sebastiani and Silvia Tomasi. "Optimization modulo theories with linear rational costs." *ACM Transactions on Computational Logic (TOCL)*, **16**(2):1–43, 2015.

[ST18] Roberto Sebastiani and Patrick Trentin. "OptiMathSAT: A Tool for Optimization Modulo Theories." *Journal of Automated Reasoning*, Dec 2018.

[STB06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. "Combinatorial sketching for finite programs." In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pp. 404–415, 2006.

[Tar98] Alfred Tarski. "A decision method for elementary algebra and geometry." In *Quantifier elimination and cylindrical algebraic decomposition*, pp. 24–84. Springer, 1998.

[Tel80] Paul Teller. "Computer proof." *The Journal of Philosophy*, **77**(12):797–803, 1980.

[Tur89] Alan Turing. "Checking a large routine." In *The early British computer conferences*, pp. 70–72, 1989.

[Wan94] Shouhong Wang. "A neural network method of density estimation for univariate unimodal data." *Neural Computing & Applications*, **2**(3):160–167, 1994.

[Wha16a] Daniel Whalen. "Holophrasm: a neural automated theorem prover for higher-order logic." *arXiv preprint arXiv:1608.02644*, 2016.

[Wha16b] Daniel Whalen. "Holophrasm: a neural Automated Theorem Prover for higher-order logic.", 2016.

[WK18] Eric Wong and Zico Kolter. "Provable defenses against adversarial examples via the convex outer adversarial polytope." In *International Conference on Machine Learning*, pp. 5286–5295. PMLR, 2018.

[WL19] Antoine Wehenkel and Gilles Louppe. "Unconstrained monotonic neural networks." In *Advances in Neural Information Processing Systems*, pp. 1543–1553, 2019.

[WPW18] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. "Formal security analysis of neural networks using symbolic intervals." In *27th USENIX Security Symposium (USENIX Security 18)*, pp. 1599–1614, 2018.

[XCL16] Lie Xu, Chiu-sing Choy, and Yi-Wen Li. "Deep sparse rectifier neural networks for speech denoising." In *2016 IEEE International Workshop on Acoustic Signal Enhancement (IWAENC)*, pp. 1–5. IEEE, 2016.

[XTJ17] Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. "Reachable set computation and safety verification for neural networks with ReLU activations." *arXiv preprint arXiv:1712.08163*, 2017.

[XTJ18] Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. "Output reachable set estimation and verification for multilayer neural networks." *IEEE transactions on neural networks and learning systems*, **29**(11):5777–5783, 2018.

[YD19a] Kaiyu Yang and Jia Deng. "Learning to prove theorems via interacting with proof assistants." In *International Conference on Machine Learning*, pp. 6984–6994. PMLR, 2019.

[YD19b] Kaiyu Yang and Jia Deng. "Learning to Prove Theorems via Interacting with Proof Assistants." In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pp. 6984–6994. PMLR, 2019.

[YDC17] Seungil You, David Ding, Kevin Robert Canini, Jan Pfeifer, and Maya R. Gupta. "Deep Lattice Networks and Partial Monotonic Functions." In *NIPS*, pp. 2981–2989, 2017.

[YFG19] Weikun Yang, Grigory Fedyukovich, and Aarti Gupta. "Lemma synthesis for automating induction over algebraic data types." In *International Conference on Principles and Practice of Constraint Programming*, pp. 600–617. Springer, 2019.

[YN13] Shuo Yang and Sriraam Natarajan. "Knowledge intensive learning: Combining qualitative constraints with causal independence for parameter learning in probabilistic models." In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 580–595. Springer, 2013.

[ZHL20] Albert Zhao, Tong He, Yitao Liang, Haibin Huang, Guy Van den Broeck, and Stefano Soatto. "LaTeS: Latent Space Distillation for Teacher-Student Driving Policy Learning." In *Conference on Robot Learning (CoRL)*, 2020.

[ZMJ18] He Zhu, Stephen Magill, and Suresh Jagannathan. "A data-driven CHC solver." *ACM SIGPLAN Notices*, **53**(4):707–721, 2018.