2020

Learning-Aided Program Synthesis And Verification

Xujie Si
University of Pennsylvania

Learning-Aided Program Synthesis And Verification

Abstract

The enormous rise in the scale, scope, and complexity of software projects has created a thriving marketplace for program reasoning tools. Despite broad adoption by industry, developing such tools remains challenging. For each project, specialized heuristics or analysis rules have to be carefully designed and customized, which requires non-trivial expertise. Recently machine learning, especially deep learning, achieved remarkable successes in many challenging areas such as image recognition and strategy game playing. Inspired by these successes, this thesis is concerned with the following question: can program reasoning be effectively learned and automatically improved over time?

This thesis demonstrates that learning-based techniques can be a new driving force for tackling fundamental program reasoning challenges, particularly, program synthesis and program verification. First, this thesis presents a scalable inductive logic programming (ILP) framework, Difflog, which can synthesize a rich set of logical rules used in various important domains like program analysis, relational query and knowledge discovery. Unlike classic program synthesis techniques, which heavily rely on manually designed heuristics or symbolic constraint solvers, Difflog leverages efficient gradient-based approaches, which is possible due to a novel numerical relaxation of logical rules. Second, this thesis presents an end-to-end deep learning framework for program verification, Code2Inv, which directly maps a piece of source code to its related proof without requiring any annotations from human experts. Code2Inv is inspired by the recent AI breakthrough, AlphaGo; however, unlike the two-dimensional game board, programs have sophisticated structures and correct proofs are extremely rare, posing unique challenges on representation learning and reinforcement learning. To address these challenges, we leverage advances of graph neural networks and develop a counterexample-based smooth reward mechanism. Code2Inv outperforms state-of-the-art approaches that are based on manually designed heuristics or decision tree learning, and the learned policy by Code2Inv can generalize to unseen programs. Furthermore, Code2Inv can be flexibly customized as a Constrained Horn Clause (CHC) solver as well as a meta-solver for syntax-guided program synthesis tasks.

Degree Type

Dissertation

Degree Name

Doctor of Philosophy (PhD)

Graduate Group

Computer and Information Science

First Advisor

Mayur Naik

Keywords

deep learning, machine learning, program analysis, program synthesis, program verification, reinforcement learning

Subject Categories

Computer Sciences

LEARNING-AIDED PROGRAM SYNTHESIS AND VERIFICATION

Xujie Si

A DISSERTATION
in
Computer and Information Science

Presented to the Faculties of the University of Pennsylvania
in
Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2020

Supervisor of Dissertation

Mayur Naik, Professor, Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor, Computer and Information Science

Dissertation Committee:

Rajeev Alur, Zisman Family Professor of Computer and Information Science

Osbert Bastani, Research Assistant Professor of Computer and Information Science

Steve Zdancewic, Professor of Computer and Information Science

Le Song, Associate Professor in College of Computing, Georgia Institute of Technology

LEARNING-AIDED PROGRAM SYNTHESIS AND VERIFICATION

Dedicated to my parents and brothers.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Mayur Naik, for providing a constant source of wisdom, guidance, and support over the six years of my Ph.D. journey. Were it not for his mentorship, I would not be here today.

My thesis committee consisted of Rajeev Alur, Osbert Bastani, Steve Zdancewic, and Le Song: I thank them for the careful reading of this document and suggestions for improvement. In addition to providing valuable feedback on this dissertation, they have also provided extremely helpful advice over the years on research, communication, networking, and career planning.

I would like to thank all my collaborators, particularly Hanjun Dai, Mukund Raghothaman, and Xin Zhang, with whom I have numerous fruitful discussions, which finally lead to this thesis. I deeply enjoyed each research project during my Ph.D. study, which would not be possible without a group of amazing collaborators including Aws Albarghouthi, Paraschos Koutris from the University of Wisconsin–Madison, Insu Yun, Taesoo Kim, Yuan Yang, Binghong Chen from Georgia Tech, Changwoo Min from Virgina Tech, Yeongjin Jang from Oregon State University, Radu Grigore from the University of Kent, Vasco Manquinho, Mikoláš Janota from the University of Lisbon, and Alexey Ignatiev from Monash University.

I am grateful to the wonderful group of PhDs and postdocs Mayur has assembled. I thank Mukund Raghothaman for great advice on writing and presentation, Kihong Heo and Woosuk Lee for sharing skills on quickly building strong research prototypes, Xin Zhang and Sulekha Kulkarni for kind encouragement during paper rejections, Elizabeth Dinella, Pardis Pashakhanloo, Jiani Huang and Ziyang Li for timely feedback before paper submissions.

I enjoyed many presentations and discussions during the weekly seminars from Penn PLClub and DSL, which also provide invaluable opportunities for me to prac-

tice conference talks. I appreciate the insightful feedback and helpful advice on presentation from Benjamin Pierce, Stephanie Weirich, Steve Zdancewic, Boon Thau Loo, Zack Ives, Susan Davidson and Val Tannen.

I thank all organizers of TGIFs, where I had the the most relaxing moments in a week and enjoyed many interesting conversations with Jennifer Paykin, Leonidas Lampropoulos, Antal Spector-Zabusky, Caleb Stanford, Richard Zhang, Omar Navarro Leija, Li-yao Xia, Paul He and Anton Xue.

I would like to thank my friends at Penn or near Philly: Meng Xu, Yao Li, Yishuai Li, Qizhen Zhang, Yinjun Wu, Yi Zhang, Haoxian Chen, Hui Lyu, Hua Li, and Yiyuan Zhao. Hanging out with them in the weekends for food or movie or hiking is the perfect way to release pressure and stay optimistic when facing various challenges.

I spent an exceptional final year. I am grateful to the wonderful Deep Learning team lead by Oriol Vinyals and Robust AI team lead by Pushmeet Kohli at DeepMind, where I had a fantastic summer working with awesome colleagues Yujia Li, Vinod Nair, and Felix Gimeno. I appreciate the warm support from Arie Gurfinkel in the cold winter, his critical feedback which helped significantly through the tough job search process, and his wonderful students Nham Le and Hari Govind.

It is my tremendous blessing to meet my girlfriend, Ningning Xie, who made the second half of my Ph.D. journey wonderful and worthwhile. I am deeply indebted to Ningning for her unwavering support, heartful companion and wise suggestions during the hardest moment.

Last but not least, I thank my parents and brothers for their love and support during every stage of my life. For them, no words of gratitude will ever be enough. I dedicate this thesis to them.

ABSTRACT

LEARNING-AIDED PROGRAM SYNTHESIS AND VERIFICATION

Xujie Si

Mayur Naik

The enormous rise in the scale, scope, and complexity of software projects has created a thriving marketplace for program reasoning tools. Despite broad adoption by industry, developing such tools remains challenging. For each project, specialized heuristics or analysis rules have to be carefully designed and customized, which requires non-trivial expertise. Recently machine learning, especially deep learning, achieved remarkable successes in many challenging areas such as image recognition and strategy game playing. Inspired by these successes, this thesis is concerned with the following question: *can program reasoning be effectively learned and automatically improved over time?*

This thesis demonstrates that learning-based techniques can be a new driving force for tackling fundamental program reasoning challenges, particularly, program synthesis and program verification. First, this thesis presents a scalable inductive logic programming (ILP) framework, Difflog, which can synthesize a rich set of logical rules used in various important domains like program analysis, relational query and knowledge discovery. Unlike classic program synthesis techniques, which heavily rely on manually designed heuristics or symbolic constraint solvers, Difflog leverages efficient gradient-based approaches, which is possible due to a novel numerical relaxation of logical rules. Second, this thesis presents an end-to-end deep learning framework for program verification, Code2Inv, which directly maps a piece of source code to its related proof without requiring any annotations from human experts. Code2Inv is inspired by the recent AI breakthrough, AlphaGo; however, unlike the two dimensional game board, programs have sophisticated structures and correct proofs are extremely rare, posing unique challenges on representation

learning and reinforcement learning. To address these challenges, we leverage advances of graph neural networks and develop a counterexample-based smooth reward mechanism. Code2Inv outperforms state-of-the-art approaches that are based on manually designed heuristics or decision tree learning, and the learned policy by Code2Inv can generalize to unseen programs. Furthermore, Code2Inv can be flexibly customized as a Constrained Horn Clause (CHC) solver as well as a meta-solver for syntax-guided program synthesis tasks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

1.1 The New Driving Force for Program Reasoning

The enormous rise in the scale, scope, and complexity of software projects has created a thriving marketplace for program reasoning tools, which plays a crucial role in modern software development for high reliability. Program testing, analysis, and verification tools have been widely adopted in many large IT companies. Prominent examples are SLAM [25], which is a software model checker for Windows device drivers from Microsoft, Infer [39], which is a multiple language static analyzer from Facebook, Tricorder [147], which is a static analyzer deployed on 2-billion-line codebase [138] from Google, CBMC [47], which is a bounded model checker from Amazon for verifying boot code in AWS data centers, and many other successful applications [33, 52, 108]. There is also an increasing trend of applying rigorous reasoning tools in popular web services [45] and autonomous vehicles [37].

Despite broad adoption by industry, developing such program reasoning tools remains challenging. Particularly, specialized heuristics or rules have to be carefully designed and customized for each project, which requires non-trivial expertise. For instance, static analysis tools are essentially a large body of abstract domains and logical rules [170, 130, 184] handcrafted by experts. And, due to undecidability, such logical rules have to be specialized from project to project in order to maintain a reasonable accuracy. For program verification tools, the situation is even worse since not only the designers but also the users have to be highly-skilled and spend tremendous effort in supplying non-trivial specifications or annotations [107, 52, 108]. The lack of expertise prevents these reasoning tools from being included in

the development toolbox of average programmers.

To address this issue, many synthesis and automated verification techniques have been developed. Program synthesis has made significant progresses over the past decade [105, 172, 83, 14, 17] and has proven to be successful for helping end-users on tasks like Excel data manipulation [78, 106, 136, 28, 82]. However, how synthesis could help professional programmers or even experts of program analysis and verification has been rarely studied. This is due to a number of fundamental challenges like expressiveness as well as scalability of synthesized programs, adaption to evolving requirements or various domains, and resolving underspecified or even conflicted constraints. Automated verification, especially concerning functional correctness, faces the exact same set of challenges. Automated verification has largely relied on and benefited from symbolic constraint solvers, particularly SAT/SMT solvers [165, 62, 56, 142], which enable efficient validation of a candidate proof; however, proposing a candidate proof is essentially a challenging synthesis problem. Thus, for both synthesis and verification tasks, certain kind of intervention (or creativity) from human experts seems unavoidable. This raises a natural and philosophical question: *can human-level creativity or intelligence be automated?*

Surprisingly, recent advances in machine learning have overturned traditional wisdom in many challenging areas where human expertise is believed to be necessary. In computer vision, automatically learned features [102, 87] turn out to be much more effective than hand-engineered ones that have been dominant for more than two decades [190, 85]. In strategic games like Shogi, Chess and Go, machine learning approaches [166, 167, 168] have successfully defeated human world champions with a fairly large margin. Machine learning has also achieved many other astonishing successes [186, 155, 27, 92, 120] in areas like machine translation, physical simulation, protein folding, radiology, pharmacy, to name a few. All these recent successes indicate that human-level creativity or intelligence

can be automated or even surpassed.

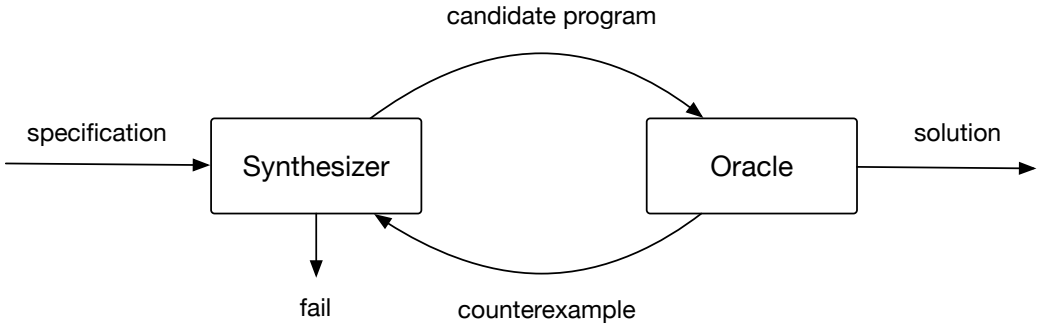
This thesis hypothesizes that learning will be the new driving force for program reasoning tasks, which have heavily relied on either SAT/SMT solving or domain-specific heuristics from human experts, and have only witnessed successes in few places that can afford to enough experts. To have a broad impacts on general software development, the new trend of program reasoning will be learning-driven approaches, which automatically adapt to new settings and discover effective rules, heuristics, or policies over interactions with new environments. This thesis demonstrates promising results of learning-based techniques on tackling fundamental program reasoning challenges, particularly, program synthesis and verification.

This thesis presents a learning-aided framework and highlights two particular instantiations, Difflog and Code2Inv. Difflog is an inductive logic programming (ILP) engine leveraging efficient gradient-based methods and numerical relaxation for learning logical rules, which is fundamentally different from classic Prolog-based ILP engines. Code2Inv leverages deep learning models for loop invariant generation. In the rest of this thesis, we illustrate how various learning techniques can be synergistically integrated with classic synthesis and verification approaches, and show that the learning-aided design enables a general solution to many different problems.

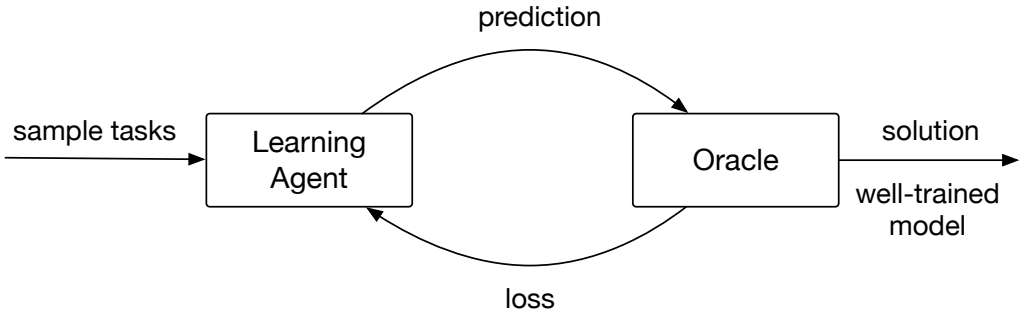
1.2 A Learning-aided Reasoning Framework

Our learning-aided reasoning framework is based on the standard program synthesis framework, *counter-example guided inductive synthesis* (CEGIS) [13, 172]. The key insight is to embed learning techniques into this classic framework, which enables new potentials provided by learning and at the same time leverages the accumulated wisdom in program synthesis community.

Figure 1.1a depicts the CEGIS framework, which consists of a synthesizer and



(a) The counterexample guided program synthesis (CEGIS) framework



(b) General architecture of our learning-aided reasoning framework

Figure 1.1: Comparison of the counterexample guided program synthesis framework and our learning-aided reasoning framework.

an oracle. The synthesizer first takes as input a specification, which can be either a formula in certain background theory specifying desired functional properties formally or a set of input-output example pairs informally demonstrating desired functionality, and then interacts with the oracle — iteratively proposes a new candidate program satisfying accumulated counterexamples (if any) and receives a new counterexample from the oracle. This interaction continues until a solution satisfying the specification is accepted by the oracle. The oracle is usually an SMT solver checking whether the given specification holds on a candidate program and supplying a counterexample if not. Progresses and innovations of CEGIS framework are mainly on how a synthesizer generates new candidate programs, for which

there are three popular approaches: i) enumerative approach [178, 16], which systematically enumerate all possible candidates in increasing order of complexity; ii) constraint-based approach [81, 172], which reduces candidate generation into constraint solving by encoding counterexamples as new constraints; and iii) stochastic approach [151], which generates candidate programs in a stochastic way, e.g. random sampling and mutation via Metropolis-Hastings.

The general architecture of our *Learning-Aided Reasoning frameworK* (LARK) is depicted in Figure 1.1b, which is inspired by the CEGIS framework but has several critical differences. First of all, LARK has a learning agent rather than a synthesizer, which is of course not simply a new name. Unlike a synthesizer in CEGIS, which has a *predefined* algorithm that incorporates counterexamples and proposes new candidates, a learning agent is a *learnable* component, which is trained to make good predictions and minimize the loss. The loss is designed in such a way that a desired solution has zero loss. A prediction can be either a candidate program or some intermediate thing that can be used to generate a candidate program, depending on whether the learning agent is in charge of entire synthesis procedure or a fraction of it. The learning agent is implemented as certain kind of machine learning model. Secondly, LARK eventually produces not only a solution but also a well-trained model, which is a representation of the learning agent. In fact, the solution is simply a byproduct of the LARK framework. After being trained, the agent becomes an executable algorithm, which immediately produces the solution. Besides solving the exact task the agent has been trained on, the agent can handle similar but *unseen* task very efficiently. This is really intriguing because the popular CEGIS approaches cannot benefit from past experiences of solving similar or even exactly the same tasks. Thirdly, instead of one particular specification, LARK can take as input multiple tasks simultaneously. The feedback on one task from oracle could potentially improve the progress on the other task assuming their solutions share some *latent* algorithm. That is, LARK also has some *meta-learning* capability.

Challenges. To instantiate our learning-aided framework, we need to answer three key questions. First, what is a good representation of the learning agent? Should the agent be a program in some domain specific language, or a set of interpretable rules, or a support vector machine, or a probabilistic graphical model, or a deep neural network, or a hybrid combination of these? The representation matters in various aspects, such as interpretability, generalization, and learning efficiency. Ideally, we want the learned agent to be great on all of these aspects, which is, however, beyond the capability of current machine learning or artificial intelligence techniques, especially for the tasks involving symbolic and logical reasonings. This thesis aims to explore trade-offs of various aspects for learning algorithms automatically in the areas of program synthesis and verification.

Second, how to decide the loss of a prediction from the agent? A prediction may or may not be directly checked by the oracle, as generating a candidate program may need a sequence of predictions. Furthermore, the oracle usually provides a counterexample, rather than a numerical score. Either a reasonable way to turn counterexamples into numerical scores is necessary, or we should design a new oracle that directly gives numerical feedback.

Third, how to train the agent? The answer might seem obvious, as gradient descent methods (e.g. backward propagation) are standard for training machine learning models. However, standard automatic differentiation techniques cannot be used to compute gradients in the LARK framework. This is because the mapping from a prediction to a loss involves non-trivial semantics like fixed-point computation and symbolic constraint solving, which is not differentiable. Propagating gradients through a complex logical reasoning process is a unique challenge of instantiating the LARK framework.

1.3 Contributions and Organizations

This thesis proposes various *learning-aided* techniques for program reasoning tasks, specifically program synthesis and verification. The key novelty lies in bridging the gap between discrete logic reasoning and continuous numerical optimization. We summarize contributions of this thesis as follows.

In Chapter 2, we briefly introduce necessary background on numerical relaxation, deep learning and reinforcement learning.

In Chapter 3, we go over applications for which learning logical rules can be very useful. We highlight a particular application — finding API misuse bugs in system software, which is intriguing because it shows that combining the learned rules and statistical information is very effective in discovering security vulnerabilities without a specification in large system software like Linux kernel and OpenSSL library.

In Chapter 4, we present two inductive logic programming (ILP) engines, ALPS, which is an instantiation of the CEGIS framework, and Difflog, which is an instantiation of the LARK framework. ALPS leverages a novel syntax-guided technique, *template augmentation*, for generating candidate logical rules and incorporates counterexamples using an efficient *bi-directional search* technique. Difflog is a synergistic combination of a syntax-guided search-based technique (the core of ALPS) and gradient-based numerical optimization. Difflog uses a novel numerical relaxation technique, which attaches numerical weights to logical rules. The semantics of weights carried with logical rules in fixed point computation naturally forms a Viterbi semi-ring. This design enables a *differentiable* oracle so that efficient gradient-based approaches can be leveraged to learn logical rules.

In Chapter 5, we present a loop invariant generation system, Code2Inv, which is another novel instantiation of LARK framework. Code2Inv is an end-to-end deep learning system that directly maps a piece of source code to its corresponding invariant without requiring any annotations from human experts. Code2Inv uses a graph representation of source code, embeds the graph into high-dimensional vec-

tor space, and reduces loop invariant generation into a multiple step decision process carried out by a neural agent. Instead of making the oracle differentiable as we did for Difflog, we propose a novel counterexample-guided reward mechanism that turns discrete counterexamples from the oracle into continuous numerical reward, and then train Code2Inv using the standard policy gradient algorithm.

In Chapter 6, we formalize the Code2Inv framework and demonstrate two intriguing extensions of Code2Inv on very different tasks — syntax-guided program synthesis (SyGuS) task and constrained Horn clause (CHC) solving. We further outline a number of future extensions in Chapter 7, and conclude in Chapter 8.

CHAPTER 2

BACKGROUND

2.1 Numerical Relaxation

Numerical relaxation is a popular strategy in mathematical optimizations. The idea is to find a close approximation of a difficult problem by relaxing certain constraints. The relaxed problem can usually be solved very efficiently, and the solution provides useful hints to solve the original difficult problem.

For example, integer programming is an NP-hard problem, by relaxing the constraint that a variable has to be an integer, the problem becomes a linear programming problem, which can be solved in polynomial time. Solution of the linear programming problem could serve as a close estimation for the solution of the original integer programming problem. A concrete example is illustrated in Figure 2.1. The relaxed linear programming problem has the optimal solution $Z = 7$ when $x = 4.5$ and $y = 2.5$. While the actual solution for the original integer programming is $Z = 6$ when $x = 4$ and $y = 2$. Though the solution of the relaxed problem is different, the actual solution of the original problem is fairly close to it. Once the approximated solution is available, various efficient heuristics can be used to recover the actual solution. Surprisingly, the same idea is applicable to logical rule synthesis, as we will show in Chapter 4.

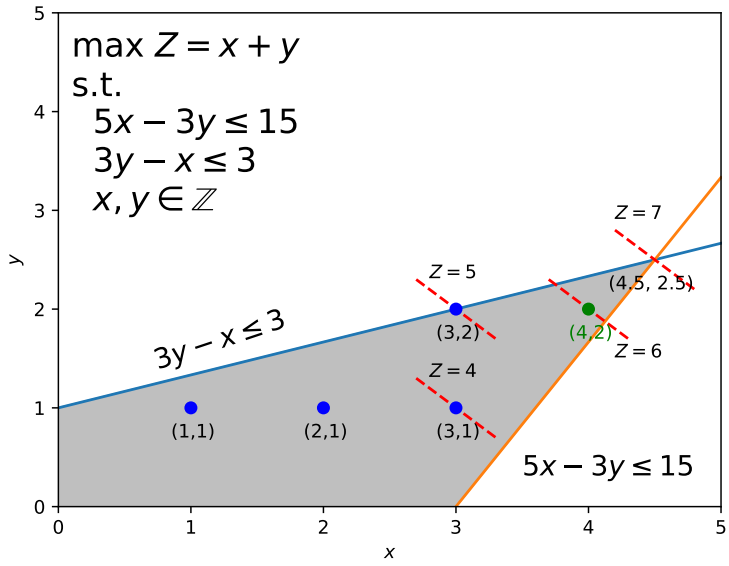
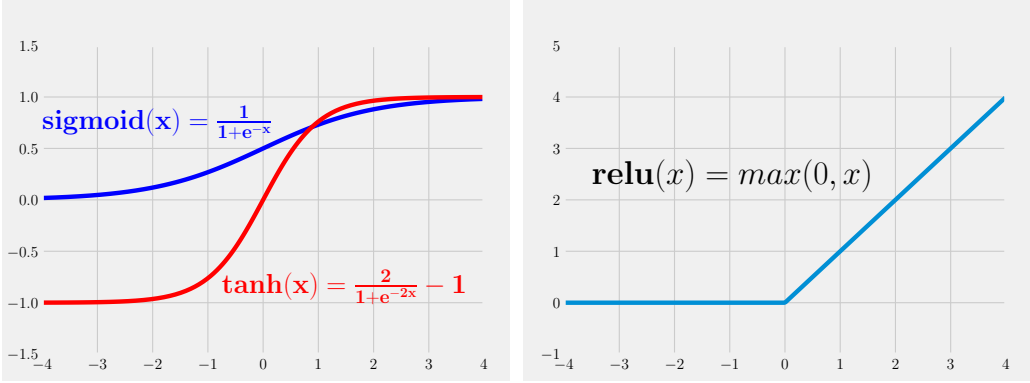


Figure 2.1: An example of numerical relaxation

2.2 Deep Learning

2.2.1 Multi-layer Perceptron

A multi-layer perceptron (MLP) is a basic neural network model, which consists of multiple directed and fully connected layers. The first layer is called input layer and the last layer is called output layer. The layers in between are called hidden layers. Each layer consists of a number of nodes (a.k.a. neurons). Each neuron in the hidden or output layer takes as input values produced by neurons in the previous layer and outputs a value, which is a non-linear transformation of the weighted sum of the input. The weights are parameters, which are usually trained (or learned) using gradient descent methods, and the number of neurons in a layer and associated non-linear transformation are hyper-parameters, which are pre-determined before training. Frequently used non-linear transformations (or activations) are: sigmoid, hyperbolic tangent (TanH), and rectified linear unit (ReLU). Their definitions and plots are depicted in figure 2.2.



(a) Sigmoid and TanH **(b) ReLU**

Figure 2.2: Plots of frequently used non-linear transformation functions.

An MLP can be viewed as a mapping

$$\mathbf{y} = f(\mathbf{x}; \theta) \tag{2.1}$$

where $\mathbf{x}$ and $\mathbf{y}$ are numeric vectors, and θ denotes weights of connections. The MLP model is particularly interesting because it can be used to approximate an arbitrary continuous function $\mathbf{y} = f^*(\mathbf{x})$ due to the *universal approximation theorem* [91].

2.2.2 Recurrent Neural Network

Recurrent neural networks (RNNs) are widely used in natural language processing tasks, especially speech recognition and machine translation. The primary goal of RNNs is to approximate the mapping from a sequence of inputs $\mathbf{x}^{(1)}, ..., \mathbf{x}^{(t)}$ to either a single output $\mathbf{y}$ or a sequence of outputs $\mathbf{y}^{(1)}, ..., \mathbf{y}^{(t)}$. An RNN defines a mapping

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \theta) \tag{2.2}$$

where $\mathbf{h}^{(t)}$ is the hidden state, from which the final output $\mathbf{y}^{(t)}$ can be computed by either a non-linear transformation or an MLP.

Popular RNN models. A simple RNN model can be

$$f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \theta) = tanh(W\mathbf{x}^{(t)} + U\mathbf{h}^{(t-1)} + b) \tag{2.3}$$

where $\theta = [W, U, b]$. The key issue of such a simple form is that the long-term dependencies are hard to capture, which makes training extremely difficult. To address this issue, a commonly used RNN model is the long short-term memory network (LSTM) [89], which introduces a *memory cell with various gates* to preserve state over a long sequence.

LSTM maintains two states — the original hidden state and a newly introduced *context* state (or memory cell). At a high level, LSTM defines a mapping function as follows:

$$\mathbf{h}^{(t)}, \mathbf{c}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{c}^{(t-1)}, \mathbf{x}^{(t)}; \theta) \tag{2.4}$$

where $\mathbf{h}^{(t)}$ is the hidden state and $\mathbf{c}^{(t)}$ is the context state. These two states are updated by three gates — input gate $i^{(t)}$, forget gate $f^{(t)}$ and output gate $o^{(t)}$ as follows:

$$\begin{aligned}
i^{(t)} &= \sigma\left(W_i\mathbf{x}^{(t)} + U_i\mathbf{h}^{(t-1)} + b_i\right) \\
f^{(t)} &= \sigma\left(W_f\mathbf{x}^{(t)} + U_f\mathbf{h}^{(t-1)} + b_f\right) \\
o^{(t)} &= \sigma\left(W_o\mathbf{x}^{(t)} + U_o\mathbf{h}^{(t-1)} + b_o\right) \\
u^{(t)} &= tanh\left(W_u\mathbf{x}^{(t)} + U_u\mathbf{h}^{(t-1)} + b_u\right) \\
c^{(t)} &= i^{(t)} \odot u^{(t)} + f^{(t)} \odot c^{(t-1)} \\
h^{(t)} &= o^{(t)} \odot tanh(c^{(t)})
\end{aligned} \tag{2.5}$$

where $\theta = [W_i, U_i, b_i, W_f, U_f, b_f, W_o, U_o, b_o, W_u, U_u, b_u]$, σ is the sigmoid function, and $\odot$ is the element-wise product.

Two common variants of LSTM are gated recurrent units (GRUs) [41] and tree-structured LSTM (Tree-LSTM) [177]. The former simplifies gates of LSTM for efficiency while the latter extends the modeling ability to tree structures.

2.2.3 Graph Neural Network

In many domains, graphs are used to represent data with rich structure, such as programs, molecules, social networks, and knowledge bases. Graph neural networks (GNNs) [109, 53, 70, 12, 187] are commonly used to learn over graph-structured data. A GNN learns an embedding (i.e. real-valued vector) for each node of the given graph using a recursive neighborhood aggregation (a.k.a. neural message passing [70]) procedure. After training, a node embedding captures the structural information within the node's K-hop neighborhood, where K is a hyper-parameter. A simple aggregation of all node embeddings (a.k.a. pooling) [189] according to the graph structure summarizes the entire graph into an embedding.

GNNs are usually parametrized with other neural network models such as MLPs, which are the learnable non-linear transformations used in message passing, and GRUs, which are used to update the embedding for each node. These MLPs and GRUs in a GNN are shared across different nodes in the graph, thus once trained, they can be applied to different graphs.

2.2.4 Memory and Attention Mechanism

A relatively new improvement of neural network concerns augmenting neural networks with external memory [73, 174, 182, 77], providing great flexibility and generalization ability. The external memory is accessed by neural network through a differentiable *attention mechanism* [23].

The external memory can have various structures, e.g. stack, queue, graphs. The value of a memory cell is usually an embedding that can be directly taken as input by neural networks. Attention mechanism assigns a likelihood (or weight) to each memory cell, which is usually in proportion to the dot product between a neural context and the embedding associated with the current cell. The retrieved value is either an embedding of some memory cell according to the distribution

determined by the attention or an aggregation of memory cells according to the attention weights. Such a design makes it easy to capture long-term dependencies, avoids encoding all information into the internal weights of neural networks, and improves the capability of generalization in practice.

2.3 Reinforcement Learning

In many tasks, the learning goal is beyond predicting some predefined label for a given input. Instead, the learning goal is to reach some beneficial state after a sequence of actions following a set of rules from some initial state. Thus, what is really learned is a policy, which predicts an action given a state and the action is ideally optimal towards the ultimate beneficial state. Reinforcement learning is a systematic methodology of solving this exact problem and has achieved remarkable successes in many challenging problems, particularly games. Prominent examples are strategical games like Chess [93] and Go [166], and video games like Atari [124], StarCraft [179], and Dota [132].

Markov Decision Process. A standard formulation of reinforcement learning is called Markov decision process (MDP). A MDP is a a 4-tuple (S, A, P, R), where S is a set of states, A is a set of actions, P is a function of the type $\mathbb{S} \to \mathbb{S} \to \mathbb{A} \to [0, 1]$, which describes the probability of the transition from one state $s \in \mathbb{S}$ to another state $s' \in \mathbb{S}$ by taking some action $a \in \mathbb{A}$, and similarly, R is a function of the type $\mathbb{S} \to \mathbb{S} \to \mathbb{A} \to \mathbb{R}$, which describes the corresponding reward when a transition happens. For simplicity, we let $P_a(s, s')$ denote $P(s, s', a)$ and $R_a(s, s')$ denote $R(s, s', a)$. The optimization objective of a MDP is to learn a policy π, a function predicting the optimal action or a distribution of actions given an input state, such that the expected gain of trajectories sampled according to π is maximized:

$$\text{Expected Gain} = \mathbb{E}_{\tau\sim\pi,P}\left[G(\tau)\right] = \mathbb{E}_{\tau\sim\pi,P}\left[\sum_{t=0}^{T-1} \gamma^t R_{a_t}(s_t, s_{t+1})\right] \tag{2.6}$$

where $\tau = [s_0, a_0, s_1, a_1, ...s_T]$, $a_i \sim \pi(s_i)$, $s_{i+1} \sim P_{a_i}(s_i, s_{i+1})$, and $\gamma \in [0, 1]$ is the discount factor, which is necessary when $T = \infty$.

In the case the state transition and the policy are deterministic and the trajectory is finite, the optimization objective can be simplified as follows:

$$\sum_{t=0}^{T-1} R_{a_t}(s_t, s_{t+1}) \tag{2.7}$$

where s_0 is the initial state, s_T is the terminal state, $a_i = \pi(s_i)$, and $s_{i+1} = P_{a_i}(s_i)$[1].

Since the reward significantly influences the optimal trajectories of reaching desired terminal states, modeling reward properly plays a crucial role in applications of reinforcement learning. Generally, the sparser the reward is, the more difficult policy learning becomes. Taking the chess as an example, if a positive reward is given only when the checkmate state is reached, and before that, the reward is always zero, learning a good policy would be extremely hard. A better reward might be assigning certain scores whenever a piece of the opponent is captured, which hints progress towards the winning state. However, such domain specific insights are not always easy to provide.

Solving Techniques. Many techniques [175] have been developed over the past three decades. These techniques can be generally categorized into dynamic programming methods and policy gradient methods. The former requires explicitly maintaining a value estimator that approximates optimal gains for possible states or state-action pairs. A policy is indirectly derived from such the value estimator. The latter directly maintains a policy and improves the policy by sampling trajectories. The latter is more feasible when the state space or action space is huge or even infinite.

[1] Since $P_{a_i}(s_i, s_{i+1})$ is always equal to 1, here the notation $P_{a_i}(s_i)$ is overloaded to refer s_{i+1}.

CHAPTER 3

APPLICATIONS OF RULE LEARNING

This chapter presents some promising examples of logical rule learning. In terms of the representation, we consider first order logical rules with least fixed-point semantics but without function symbols. More concretely, we target on a declarative logic programming language, Datalog. A key reason is the emergence of scalable Datalog solvers, including open-source [1, 153, 3, 18, 157] and commercial ones [4, 2, 6, 72]. This makes Datalog popular in a variety of domains, including bioinformatics [98, 149], big-data analytics [157, 164, 86], natural language processing [125], networking [114], program analysis [74, 35], and robotics [137]. Moreover, the concise and declarative nature of Datalog has made it the target of a growing body of meta-reasoning tools. For instance, program analyses written in Datalog are readily extensible with features such as fixed point frameworks [116, 22], abstraction refinement [193], and user interaction [194, 117, 140]. Likewise, software-defined networking (SDN) applications written in Datalog can avail of efficient provenance tracking to help in tasks such as debugging and repairing [185].

We next give a formal description of Datalog and then present a few simple but interesting enough examples illustrating the importance and challenges of learning Datalog programs.

Rules. A *term* t is either a variable $x, y, z, \ldots$, or a constant $a, b, c, \ldots$. A *relation symbol* $p, q, r, \ldots$ is associated with an arity $ar(r)$. An *atom* is an application of a relation symbol to a vector of variables and constants, e.g., $r(x, y, a)$ for a relation r with arity 3. A *ground atom* is an application of a relation symbol to constants, e.g.,

$r(a_1, \ldots, a_n)$, where a_i are constants. A *Datalog rule* C is an *expression* of the form:

$$A \text{ :- } B_1, B_2, \ldots, B_n.$$

where $A, B_1, \ldots, B_n$ are atoms. The atom A is called the *head* of the rule; the set of atoms $\{B_1, \ldots, B_n\}$ is called the *body* of the rule. A Datalog rule can be interpreted as a logical implication: if $B_1, \ldots, B_n$ are true, then so is A.

Datalog Programs. A Datalog program P is a finite set of rules. We divide relation symbols into two categories: the *input relations* whose contents are given, and the *output relations* whose contents are derived from the input relations using the program P. An input relation can never appear in the head of a rule. We use I to denote the set of *facts* (ground atoms) in the input relations. The *Herbrand base* $\mathcal{B}$ denotes all possible applications of the output relations to vectors of constants in I. A Datalog program is *recursive* if a relation symbol appears in both the head and the body of a rule.

Semantically, evaluating P on I yields a *minimal Herbrand model* of $P \cup I$, which is the smallest set of ground atoms that satisfies the rules in P and input I. Given a ground atom e, $P \cup I \models e$ denotes that P with input I derives fact e.

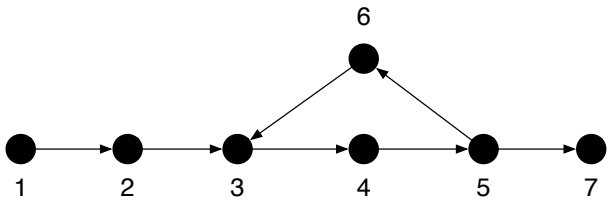
3.1 Promising Examples

Learning logical rules from data has been a long-standing challenge since early days of artificial intelligence [139, 42, 126]. We present motivating examples in the early days like knowledge discovery and motion planing, as well as recent applications in relational queries and program analysis.

Example 3.1.1 (Knowledge discovery)**.** The most widely used knowledge discovery example in the inductive logic programming (ILP) literature is computing the transitive closure of a directed graph. The problem involves one input relation `edge` and one output relation `path` with the following meaning:

- `edge`(x, y): there is an edge from node x to node y.
- `path`(x, y): there is a path from node x to node y.

Suppose the user populates the input relation, `edge`, with the following example graph:



where an edge from node i to node j indicates that `edge`(i, j) appears in the input relation.

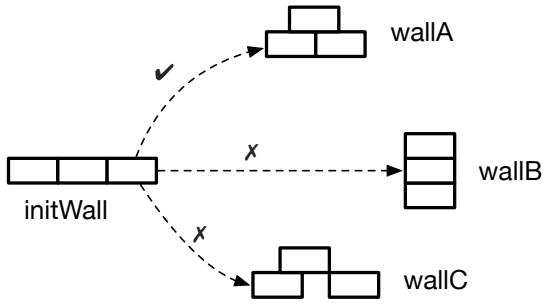
Any reasonable synthesizer should be able to figure out the following recursive program, which computes the transitive closure of a directed graph.

$$\texttt{path}(x, y) \text{ :- } \texttt{edge}(x, y).$$
$$\texttt{path}(x, z) \text{ :- } \texttt{path}(x, y), \texttt{edge}(y, z).$$

Our synthesis engine, ALPS, can efficiently maintain all possible programs, so it also discovers the other symmetric one and the following *non-linear* recursive program:

$$\texttt{path}(x, y) \text{ :- } \texttt{edge}(x, y).$$
$$\texttt{path}(x, z) \text{ :- } \texttt{path}(x, y), \texttt{path}(y, z).$$

Example 3.1.2 (Robot Strategy Learning)**.** In AI, planning is to find a sequence of actions which result in a goal state from an initial state [146], and a strategy is a mapping from a set of initial states to a set of goal states [128].



The above figure shows three examples that are used to learn how to build a stable wall strategy, which corresponds to a simplification of a real-world robotics application [128]. Specifically, the pair of `initWall` and `wallA` is a positive example, which is labeled with a check mark, while the other two pairs labeled with cross mark are negative examples. A wall is modeled as a list of lists, for example, `wallA` is represented as [[2],[1,3]], where each number corresponds to the horizontal position of a brick, which has width 2. The primitive actions can be represented as following input relations between lists of lists:

- `fetch`(x,y) : fetch a brick from wall x and get wall y.
- `putOnTopOf`(x,y) : put a brick on top of wall x and get wall y.
- `offset`(x) : each layer of wall x (except for the ground layer) is supported by an underlying layer with some offset.
- `continuous`(x) : there is no gap in wall x.

And similarly, the strategy to learn is represented as an output relation `buildWall`(x,y), which means y can be constructed by a sequence of primitive actions on x. With a few examples, our synthesizer engine learns the following Datalog program. Note that relations or predicates `p` and `q` are not supplied in the input and output relations but *invented*.

$$\texttt{buildWall}(x, y) \text{ :- } \texttt{p}(x, y), \texttt{offset}(y).$$

$$\texttt{buildWall}(x, y) \text{ :- } \texttt{p}(x, z), \texttt{buildWall}(z, y).$$
$$\texttt{p}(x, y) \text{ :- } \texttt{q}(x, y), \texttt{continuous}(y).$$
$$\texttt{q}(x, y) \text{ :- } \texttt{fetch}(x, z), \texttt{putOnTopOf}(z, y).$$

Example 3.1.3 (Relational queries)**.** In recent years, Datalog has become popular as a relational query language due to its expressiveness and scalable performance [21, 76, 153]. Sophisticated relational queries are relatively easy to learn in Datalog from input-output behaviors.

For instance, the following is an interesting relational query for finding students who take two different classes on the same day. The problem involves three input relations and one output relation with the following meaning:

- `Student`(s,n): Student s is associated with the ID n.
- `Class`(c,d): Class c is held on day d.
- `Enrolled`(n,c): The student having ID n is enrolled in class c.
- `Busy`(s): Student s takes two different classes on the same day.

It is natural in a programming-by-example setting for the user to provide an instance specifying the input-output behavior of the desired query. Using such an instance comprising input relations regarding 14 students and 6 classes, and 5 examples in the output relation `Busy`, ALPS synthesizes the following Datalog program within 18 seconds:

$$\texttt{EnrollClass}(n, c, l) \text{ :- } \texttt{Enrolled}(n, c), \texttt{Class}(c, l).$$
$$\texttt{Busy}(s) \text{ :- } \texttt{Student}(s, n), \texttt{EnrollClass}(n, c1, l),$$
$$\texttt{EnrollClass}(n, c2, l), c1\texttt{!=}c2.$$

where `EnrollClass` is an invented predicate. While ostensibly simple, the above

query is non-trivial to synthesize since it is semantically equivalent to the following complex SQL query:[2]

```
SELECT S.s FROM Student S
WHERE S.n IN (SELECT E1.n
                FROM Enrolled E1, Enrolled E2, Class C1, Class C2
                WHERE E1.n = E2.n AND E1.c <> E2.c
                AND E1.c = C1.c AND E2.c = C2.c AND C1.d = C2.d))
```

In contrast, a state-of-the art tool Scythe [180] for synthesizing SQL queries fails to generate the above SQL query within 3 hours.

Example 3.1.4 (Pointer Analysis)**.** The following problem concerns synthesizing a basic *inclusion-based* pointer analysis for C programs, namely, Andersen's classic analysis [19]. It involves four input relations, encoding different instruction types, and one output relation, encoding *points-to* information. The input relations are:

- `addr(x,y)` : there is an assignment `x := &y` in a given input program.
- `copy(x,y)` : there is an assignment `x := y`.
- `load(x,y)` : there is a load statement `y := *x`.
- `store(x,y)` : there is a store statement `*x := y`.

The output relation is `pt(x,y)`, specifying that x may point to y. Suppose the input relations are populated with a simple C program exhibiting the four kinds of instructions, e.g.:

```
1: v2 = &v1;     3: v4 = &v3;     5: v5 = v7;
2: v3 = &v2;     4: v7 = &v4;     6: v6 = *v4;
                                  7: *v5 = v2;
```

[2]Datalog can in fact be viewed as augmenting relational algebra, which is widely used in the form of SQL, with recursion.

In a few minutes, our synthesis engine learns the following recursive program, where each clause encodes an over-approximation of the semantics of one of the instruction types.

$$\texttt{pt}(x, y) \text{ :- } \texttt{addr}(x, y).$$
$$\texttt{pt}(x, z) \text{ :- } \texttt{copy}(x, y), \texttt{pt}(y, z).$$
$$\texttt{pt}(w, z) \text{ :- } \texttt{store}(x, y), \texttt{pt}(y, z), \texttt{pt}(x, w).$$
$$\texttt{pt}(x, w) \text{ :- } -\texttt{load}(x, y), \texttt{pt}(y, z), \texttt{pt}(z, w).$$

This is an exciting example of the possibilities of synthesizing declarative programs. For instance, we envision a future in which developers will be able to automatically synthesize custom static analyses by interacting with a synthesizer embedded in their development environment. We next illustrate such an initial attempt.

Example 3.1.5 (Static analyzer)**.** We demonstrate how Datalog synthesis engine like ALPS can be used to learn a static analysis to detect API misuses—a common source of bugs in today's world of complex and evolving APIs. For a given example program with known API misuses, we populate input relations representing the syntax of the program and output relations representing the bugs. Then, the synthesizer learns Datalog rules that can be used for detecting similar API misuses.

Consider the C program shown in Figure 3.1 using the OpenSSL API. Functions `ssl_socket_open1-4` establish a SSL socket and return a constant `OK` if they succeed. Two functions `ssl_socket_open{2,4}` contain API misuses in that they incorrectly return `OK` when a SSL socket is not properly established.

Our goal is to learn a Datalog program that detects functions that misuse the OpenSSL API, whose behavior is defined as follows:

- `SSL_get_peer_certificate` returns a pointer to the X509 certificate the peer presented. If the peer did not present a certificate, `NULL` is returned.

- `SSL_get_verify_result` returns the result of the verification of the X509 certificate presented by the peer, if any. It returns a constant named `X509_V_OK` if the verification succeeded or if no peer certificate was presented.

Functions should return `OK` only if (*i*) `SSL_get_peer_certificate` returns a non-null pointer, and (*ii*) `SSL_get_verify_result` returns the constant named `X509_V_OK`.

```
1   int ssl_socket_open1(SSL* ssl) {
2     X509* cert = SSL_get_peer_certificate(ssl);
3     long err = SSL_get_verify_result(ssl);
4     if (!cert) {...}
5     if (err == X509_V_OK) { ... }
6     return OK;  // correct
7   }
8
9   int ssl_socket_open2(SSL* ssl) {
10    X509* cert = SSL_get_peer_certificate(ssl);
11    if (cert == NULL) {...}
12    long err = SSL_get_verify_result(ssl);
13    ...
14    return OK;   // incorrect (missing check on err)
15  }
16
17  int ssl_socket_open3(SSL* ssl) {
18    long err = SSL_get_verify_result(ssl);
19    if (err != X509_V_OK) {...}
20    X509* cert = SSL_get_peer_certificate(ssl);
21    if (cert) {...}
22    return OK;   // correct
23  }
24
25  int ssl_socket_open4(SSL* ssl) {
26    long err = SSL_get_verify_result(ssl);
27    switch (err) {
28      case X509_V_OK:
29        cert = SSL_get_peer_certificate(ssl);
30    }
31    return OK;   // incorrect (missing check on cert)
32  }
```

Figure 3.1: An example of misusing OpenSSL APIs

The problem involves four input relations and one output relation with the following meaning:

- `OpSucc(l1,l2)`: Program control may flow from line `l1` to `l2`.
- `Check(x,l)`: The value of variable x is compared to a specific value at line `l`.

- `Certify(x,l)`: Variable x at line `l` is assigned the return value of `SSL_get_peer_certificate()`.
- `Verify(x,l)`: Variable x at line `l` is assigned the return value of `SSL_get_verify_result()`.
- `Ok(l)`: The function that returns `OK` at line `l` correctly uses the OpenSSL API.

Relations `OpSucc` and `Check` are pre-defined as part of the program's intermediate representation while relations `Certify` and `Verify` can be automatically extracted from a given API, in this case OpenSSL. We provide an instance of these relations encoding the analyzed C program to our synthesis engine, namely

`Certify(cert,2), Verify(err,3), Check(cert,4), Check(err,5), ...`

along with `Ok(6)` and `Ok(22)` as positive examples and `Ok(14)` and `Ok(31)` as negative examples in the output relation. Our synthesis engine generates the following program in 6 minutes.

$$\texttt{CertFlow}(x, l2) \coloneq \texttt{Certify}(x, l1), \texttt{OpSucc}(l1, l2).$$
$$\texttt{VeriFlow}(x, l2) \coloneq \texttt{Verify}(x, l1), \texttt{OpSucc}(l1, l2).$$
$$\texttt{CertCheck}(l2) \coloneq \texttt{CertFlow}(x, l1), \texttt{Check}(x, l1), \texttt{OpSucc}(l1, l2).$$
$$\texttt{VeriCheck}(l2) \coloneq \texttt{VeriFlow}(x, l1), \texttt{Check}(x, l1), \texttt{OpSucc}(l1, l2).$$
$$\texttt{Ok}(l) \coloneq \texttt{CertCheck}(l), \texttt{VeriCheck}(l).$$

Note that predicates `CertFlow(x,l)`, `VeriFlow(x,l)`, `CertCheck(l)`, and `VeriCheck(l)` are not specified among the input or output relations; they are actually invented,
highlighting the rich space of programs it explores.[3] The relation `CertFlow(x,l)` (`VeriFlow(x,l)` resp.) indicates the return value of `SSL_get_peer_-certificate` (`SSL_get_verify_result` resp.) flows to line `l`. The relation `CertCheck(l)` (`VeriCheck(l)`

[3]For readability, we provide intuitive names for invented predicates instead of mechanically generated names.

resp.) means the return value of `SSL_get_peer_certificate` (`SSL_get_verify_result` resp.) is compared to a specific value and control flows to line `1`.

The Datalog program correctly captures an important portion of the proper use of the OpenSSL API. This example illustrates that our synthesis engine represents a promising step towards synthesizing usable program analyzers.

3.2 A Case Study on Detecting API Misuses

In this section, we present a case study on detecting API misuses in system software like the Linux kernel and the OpenSSL library. The goal is to demonstrate that rule templates designed by experts and statistical information are quite powerful in finding security vulnerabilities without any specifications. This indicates a great real-world potential of combining rule-learning and numerical reasoning. Also, it suggests that expert knowledge should not be completely ignored when designing learning-aided systems.

We next present the design and primary results of our prototype, APISAN, for finding API misuse bugs in large system software. The key challenge of detecting API misuses is the lack of specifications, which is generally assumed to be available by any analyzer or verifier in the first place. Also, very often only distributed binaries rather than source code implementation are available. To address these challenges, our key idea is to infer correct specifications by observing how APIs are used (rather than implemented) across large codebases. The dominant usage patterns are assumed to be the correct specifications.

Figure 3.2 illustrates APISAN's workflow, which consists of three steps. First, APISAN infers possible execution traces by using symbolic execution. These execution traces witness how a particular API is being used during particular runs. Given that symbolic execution is expensive and is not yet scalable to large system like the Linux kernel, APISAN performs *under-constrained* symbolic execution [141]. Sec-

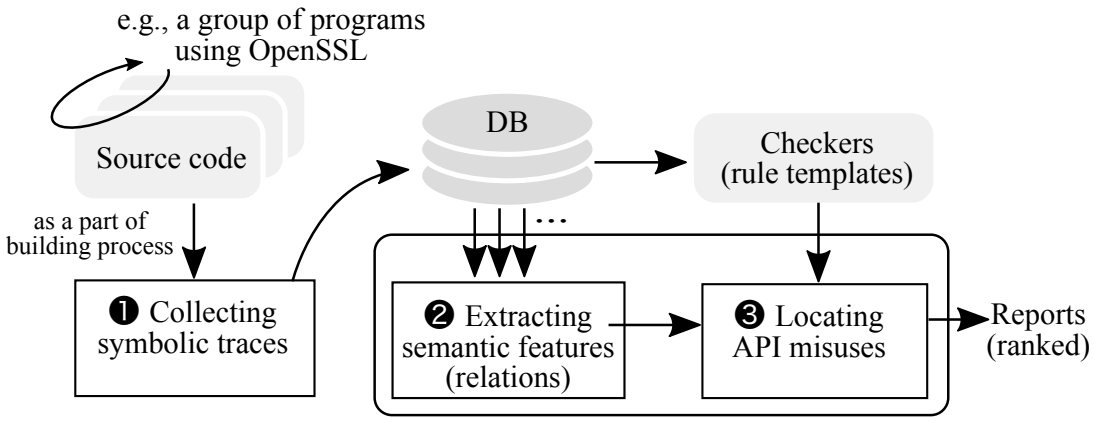


Figure 3.2: Overview of APISAN's architecture and workflow.

ond, APISAN extracts *semantic features* from symbolic execution traces. These semantic features are essentially relations, which captures how an API interacts with other elements (e.g. arguments, return values, guarding conditions, other function calls) within the current execution and can be viewed as further abstractions of execution traces. Third, APISAN applies checkers from domain experts to extracted semantic features to locate potential API misuses, which will be further ranked according to relevant usage frequencies. Checkers are implemented in Python but are essentially *rule templates*. The checking process over semantic features is equivalent to finding a subset of relations that match rule templates. Since under-constrained symbolic execution is standard (see [141]), we next elaborate on semantic features and checkers.

Semantic features. Semantic features are designed to capture the surrounding context for a given API (or function) call. APISAN considers four types of features as follows.

1. *Return value.* In system software, not only does a function return the result of its computation, but it often explicates the status of the computation through the return value; for example, non-zero value in `glibc` and `PTR_ERR()` in the Linux kernel indicates certain types of errors. Any usage of a return value in

a trace is extracted as a specific relation.

2. *Argument.* Arguments of an API can be semantically inter-related. Typical examples are memory copy APIs, such as `strncpy(d,s,n)` and `memcpy(d,s,n)`; for correct operation without buffer overrun, the size of the destination buffer `d` should be larger or equal to the copy length `n`. Thus, we consider pairwise correlations among arguments as an important feature.

3. *Causality.* Two APIs can be causally related; for example, an acquired lock should be released at the end of critical section. Besides such "direct" causal relationships, there are many *constrained* causal relationships as well. One popular example is the conditional synchronization primitives; there is a causal relationship between `mutex_trylock()` and `mutex_unlock()` only when the former returns a non-zero value.

4. *Conditions.* In many cases, there are hidden assumptions *before* or *after* calling APIs, namely, implicit pre- and post-conditions. For example, the memory allocation APIs assume that there is no integer overflow on the argument passed as allocation size, which implies that there should be a proper check before the call.

Checkers. With these semantic features, we are ready to design many interesting checkers for detecting API misuses. For example, a simple lock checker could be to use causality feature if the name of an API contains the keyword "lock", which does help to find a missing unlock bug in Linux kernel as shown in Figure 3.3. Another interesting example is to use both causality and conditions features, which helps us to find a memory leak vulnerability in OpenSSL as shown in Figure 3.4.

We have implemented many other checkers for SSL/TLS APIs, return value validation, broken argument relation, format string, etc. The primary results are summarized in Table 3.1. Overall, we applied APISAN to 92 million lines of code, including Linux Kernel, and OpenSSL, found 76 previously unknown bugs, and provided

```
// @drivers/clk/clk.c:2672
// in Linux v4.5-rc4
void clk_unregister(struct clk *clk) {
  clk_prepare_lock();
  if (clk->core->ops == &clk_nodrv_ops) {
    pr_err("%s: unregistered clock: %s\n", __func__,
        clk->core->name);
    // APISan: Missing clk_prepare_unlock()
    // @FUNC: clk_prepare_lock
    // @CONS: None
    // @POST: clk_prepare_unlock
    return;
  }
  clk_prepare_unlock();
}
```

Figure 3.3: A missing unlock bug in Linux found by APISAN.

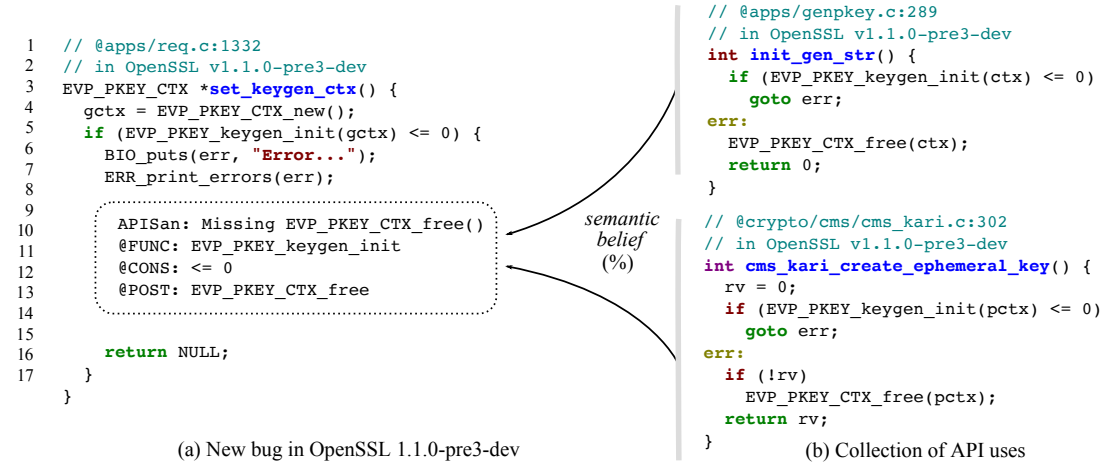


Figure 3.4: A memory leak vulnerability found by APISAN in OpenSSL 1.1.0-pre3-dev. When a crypto key fails to initialize, the allocated context (i.e., `gctx`) should be freed. Otherwise, a memory leak will occur. APISAN infers correct semantic usage of the API from (b) other uses of the API.

patches for all the bugs.

Program	Module	API misuse	Impact	Checker	#bugs	S.
Linux	cifs/cifs_dfs_ref.c	heap overflow	code execution	`args`	1	✓
	xenbus/xenbus_dev_frontend.c	missing integer overflow check	code execution	`intovfl`	1	✓
	ext4/resize.c	incorrect integer overflow check	code execution	`intovfl`	1	✓
	tipc/link.c	missing `tipc_bcast_unlock()`	deadlock	`cpair`	1	✓
	clk/clk.c	missing `clk_prepare_unlock()`	deadlock	`cpair`	1	✓
	hotplug/acpiphp_glue.c	missing `pci_unlock_rescan_remove()`	deadlock	`cpair`	1	✓
	usbvision/usbvision-video.c	missing `mutex_unlock()`	deadlock	`cpair`	1	✓
	drm/drm_dp_mst_topology.c	missing `drm_dp_put_port()`	DoS	`cpair`	1	✓
	affs/file.c	missing `kunmap()`	DoS	`cpair`	1	✓
	acpi/sysfs.c	missing `kobject_create_and_add()` check	system crash	`rvchk`	1	✓
	cx231xx/cx231xx-417.c	missing `kmalloc()` check	system crash	`rvchk`	1	✓
	qxl/qxl_kms.c	missing `kmalloc()` check	system crash	`rvchk`	1	P
	chips/cfi_cmdset_0001.c	missing `kmalloc()` check	system crash	`rvchk`	1	✓
	ata/sata_sx4.c	missing `kzalloc()` check	system crash	`rvchk`	1	✓
	hsi/hsi.c	missing `kzalloc()` check	system crash	`rvchk`	2	✓
	mwifiex/sdio.c	missing `kzalloc()` check	system crash	`rvchk`	2	✓
	usbtv/usbtv-video.c	missing `kzalloc()` check	system crash	`rvchk`	1	✓
	cxgb4/clip_tbl.c	missing `t4_alloc_mem()` check	system crash	`rvchk`	1	✓
	devfreq/devfreq.c	missing `devm_kzalloc()` check	system crash	`rvchk`	2	✓
	i915/intel_dsi_panel_vbt.c	missing `devm_kzalloc()` check	system crash	`rvchk`	1	✓
	gpio/gpio-mcp23s08.c	missing `devm_kzalloc()` check	system crash	`rvchk`	1	✓
	drm/drm_crtc.c	missing `drm_property_create_range()` check	system crash	`rvchk`	13	✓
	gma500/framebuffer.c	missing `drm_property_create_range()` check	system crash	`rvchk`	1	✓
	emu10k1/emu10k1_main.c	missing `kthread_create()` check	system crash	`rvchk`	1	✓
	m5602/m5602_s5k83a.c	missing `kthread_create()` check	system crash	`rvchk`	1	✓
	hisax/isdnl2.c	missing `skb_clone()` check	system crash	`rvchk`	1	✓
	qlcnic/qlcnic_ctx.c	missing `qlcnic_alloc_mbx_args()` check	system crash	`rvchk`	1	✓
	xen-netback/xenbus.c	missing `vzalloc()` check	system crash	`rvchk`	1	✓
	i2c/ch7006_drv.c	missing `drm_property_create_range()` check	system crash	`rvchk`	1	✓
	fmc/fmc-fakedev.c	missing `kmemdup()` check	system crash	`rvchk`	1	P
	rc/igorplugusb.c	missing `rc_allocate_device()` check	system crash	`rvchk`	1	✓
	s5p-mfc/s5p_mfc.c	missing `create_singlethread_workqueue()` check	system crash	`rvchk`	1	P
	fusion/mptbase.c	missing `create_singlethread_workqueue()` check	system crash	`rvchk`	1	P
	nes/nes_cm.c	missing `create_singlethread_workqueue()` check	system crash	`rvchk`	1	✓
	dvb-usb-v2/mxl111sf.c	missing `mxl111sf_enable_usb_output()` check	malfunction	`rvchk`	2	✓
	misc/xen-kbdfront.c	missing `xenbus_printf()` check	malfunction	`rvchk`	1	✓
	pvrusb2/pvrusb2-context.c	incorrect `kthread_run()` check	malfunction	`rvchk`	1	P
	agere/et131x.c	incorrect `drm_alloc_coherent()` check	malfunction	`rvchk`	1	✓
	drbd/drbd_receiver.c	incorrect `crypto_alloc_hash()` check	malfunction	`rvchk`	1	✓
	mlx4/mr.c	incorrect `mlx4_alloc_cmd_mailbox()` check	maintanence	`rvchk`	1	✓
	usnic/usnic_ib_qp_grp.c	incorrect `kzalloc()` check	maintanence	`rvchk`	2	✓
	aoe/aoecmd.c	incorrect `kthread_run()` check	maintanence	`rvchk`	1	✓
	ipv4/tcp.c	incorrect `crypto_alloc_hash()` check	maintanence	`rvchk`	1	✓
	mfd/bcm590xx.c	incorrect `i2c_new_dummy()` check	maintanence	`rvchk`	1	P
	usnic/usnic_ib_main.c	incorrect `ib_alloc_device()` check	maintanence	`rvchk`	1	✓
	usnic/usnic_ib_qp_grp.c	incorrect `usnic_fwd_dev_alloc()` check	maintanence	`rvchk`	1	✓
OpenSSL	dsa/dsa_gen.c	missing `BN_CTX_end()`	DoS	`cpair`	1	✓
	apps/req.c	missing `EVP_PKEY_CTX_free()`	DoS	`cpair`	1	✓
	dh/dh_pmeth.c	missing `OPENSSL_memdup()` check	system crash	`rvchk`	1	✓
PHP	standard/string.c	missing integer overflow check	code execution	`intovfl`	3	✓
	phpdbg/phpdbg_prompt.c	format string bug	code execution	`args`	1	✓
Python	Modules/zipimport.c	missing integer overflow check	code execution	`intovfl`	1	✓
rabbitmq	librabbitmq/amqp_openssl.c	incorrect `SSL_get_verify_result()` use	MITM	`cond`	1	✓
hexchat	common/server.c	incorrect `SSL_get_verify_result()` use	MITM	`cond`	1	✓
lprng	auth/ssl_auth.c	incorrect `SSL_get_verify_result()` use	MITM	`cond`	1	P
afflib	lib/aftest.cpp	missing `BIO_new_file()` check	system crash	`rvchk`	1	✓
	tools/aff_bom.cpp	missing `BIO_new_file()` check	system crash	`rvchk`	1	✓

Table 3.1: List of new bugs discovered by APISAN. We sent patches of all 76 new bugs; 69 bugs have been already confirmed by corresponding developers (marked ✓in the rightmost column); 7 bugs (marked P in the rightmost column) have not been confirmed yet. APISAN analyzed 92 million LoC and found one bug per 1.2 million LoC.

3.3 Discussion

In this chapter, we have presented many interesting applications of learning logical rules in various domains. We also demonstrate its great potential in finding security vulnerabilities in large system software due to API misuses. We will show how to effectively learn a set of rich logical rules from data in the next chapter.

Illustrative examples and experiment results presented in this chapter are from the following published papers:

- ✎ Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paris Koutris, Mayur Naik. Syntax-Guided Synthesis of Datalog Programs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 515–527, ACM 2018.

- ✎ Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, Mayur Naik. APISan: Sanitizing API Usages through Semantic Cross-checking. In *Proceedings of the 25th USENIX Conference on Security Symposium*, page 363–378, USA 2016. USENIX Association.

CHAPTER 4

LEARNING-AIDED RULE SYNTHESIS

4.1 Introduction

As a result of its rich expressive power and efficient implementations, the logic programming language Datalog has witnessed applications in diverse domains such as bioinformatics [156], big-data analytics [164], robotics [137], networking [113], and formal verification [35]. Users on the other hand are often unfamiliar with logic programming. The programming-by-example (PBE) paradigm aims to bridge this gap by providing an intuitive interface for non-expert users [78].

Typically, a PBE system is given a set of input tuples and sets of desirable and undesirable output tuples. The central computational problem is that of synthesizing a Datalog program, i.e., a set of logical inference rules which produces, from the input tuples, a set of conclusions which is compatible with the output specification. Previous approaches to this problem focus on optimizing the combinatorial exploration of the search space. For example, Zaatar encodes the derivation of output tuples as a SAT formula for subsequent solving by a constraint solver [10], and inductive logic programming (ILP) systems employ sophisticated pruning algorithms based on ideas such as inverse entailment [126]. Given the computational complexity of the search problem, however, these systems are hindered by large or difficult problem instances. Furthermore, these systems have difficulty coping with minor user errors or noise in the training data.

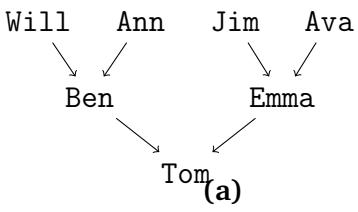
We take a fundamentally different approach to the problem of synthesizing Datalog programs. Inspired by the success of numerical methods in machine learning and other large scale optimization problems, and of the strategy of relaxation in

solving combinatorial problems such as integer linear programming, we extend the classical discrete semantics of Datalog to a continuous setting named Difflog, where each rule is annotated with a real-valued weight, and the program computes a numerical value for each output tuple. This step can be viewed as an instantiation of the general K-relation framework for database provenance [75] with the Viterbi semi-ring[4] being chosen as the underlying space K of provenance tokens. We then formalize the program synthesis problem as that of selecting a subset of target rules from a large set of candidate rules, and thereby uniformly capture various methods of inducing syntactic bias, including syntax-guided synthesis (SyGuS) [14], and template rules in meta-interpretive learning [128].

The synthesis problem thus reduces to that of finding the values of the rule weights which result in the best agreement between the computed values of the output tuples and their specified values (1 for desirable and 0 for undesirable tuples). The fundamental NP-hardness of the underlying decision problem manifests as a complex search surface, with local minima and saddle points. To overcome these challenges, we devise a hybrid optimization algorithm which combines Newton's root-finding method with periodic invocations of a simulated annealing search. Finally, when the optimum value is reached, connections between the semantics of Difflog and Datalog enable the recovery of a classical discrete-valued Datalog program from the continuous-valued optimum produced by the optimization algorithm.

A particularly appealing aspect of relaxation-based synthesis is the randomness caused by the choice of the starting position and of subsequent Monte Carlo iterations. This manifests both as a variety of different solutions to the same problem, and as a variation in running times. Running many search instances in parallel therefore enables stochastic speedup of the synthesis process, and allows us to leverage compute clusters in a way that is fundamentally impossible with deterministic

[4] A semi-ring defined over the base set [0, 1], where $\oplus$ is the max function and $\otimes$ is the usual multiplication of real numbers.



(a)

Input tuples (EDB)
parent (Will, Ben)
parent (Ann, Ben)
parent (Jim, Emma)
parent (Ava, Emma)
parent (Ben, Tom)
parent (Emma, Tom)

(b)

Output tuples (IDB)
samegen (Ben, Emma)
samegen (Ann, Will)
samegen (Jim, Ann)
...

(c)

Figure 4.1: Example of a family tree (a), and its representation as a set of input tuples (b). An edge from x to y indicates that x is a parent of y, and is represented symbolically as the tuple $\texttt{parent}(x, y)$. The user wishes to realize the relation $\texttt{samegen}(x, y)$, indicating the fact that x and y occur are from the same generation of the family (c).

approaches. We have implemented Difflog and evaluate it on a suite of 34 benchmark programs from recent literature. We demonstrate significant improvements over the state-of-the-art, even while synthesizing complex programs with recursion, invented predicates, and relations of arbitrary arity.

4.2 The Datalog Synthesis Problem

In this section, we concretely describe the Datalog synthesis problem, and establish some basic complexity results. We use the family tree shown in Figure 4.1 as a running example. In Section 4.2.1, we briefly describe how one may compute $\texttt{samegen}(x, y)$ from $\texttt{parent}(x, y)$ using a Datalog program. In Section 4.2.2, we formalize the query synthesis problem as that of rule selection.

4.2.1 Overview of Datalog

The set of tuples inhabiting relation $\texttt{samegen}(x, y)$ can be computed using the following pair of *inference rules*, r_1 and r_2:

$$r_1 : \ \texttt{samegen}(x, y) \coloneq \texttt{parent}(x, z), \texttt{parent}(y, z).$$

$$r_2 : \texttt{samegen}(x, u) :- \texttt{parent}(x, y), \texttt{parent}(u, v), \texttt{samegen}(y, v).$$

Rule r_1 describes the fact that for all persons x, y, and z, if both x and y are parents of z, then x and y occur at the same level of the family tree. Informally, this rule forms the base of the inductive definition. Rule r_2 forms the inductive step of the definition, and provides that x and u occur in the same generation whenever they have children y and v who themselves occur in the same generation.

By convention, the relations which are explicitly provided as part of the input are called the EDB, $\mathcal{I} = \{\texttt{parent}\}$, and those which need to be computed as the output of the program are called the IDB, $\mathcal{O} = \{\texttt{samegen}\}$. To evaluate this program, one starts with the set of input tuples, and repeatedly applies rules r_1 and r_2 to derive new output tuples. Note that because of the appearance of the literal $\texttt{samegen}(y, v)$ on the right side of rule r_2, discovering a single output tuple may recursively result in the further discovery of additional output tuples. The derivation process ends when no additional output tuples can be derived, i.e., when the set of conclusions reaches a *fixpoint*.

More generally, we assume a collection of *relations*, $\{P, Q, \dots\}$. Each relation P has an arity $k \in \mathbb{N}$, and is a set of *tuples*, each of which is of the form $P(c_1, c_2, \dots, c_k)$, for some *constants* c_1, c_2, …, c_k. The Datalog program is a collection of rules, where each rule r is of the form:

$$P_h(\boldsymbol{u}_h) :- P_1(\boldsymbol{u}_1), P_2(\boldsymbol{u}_2), \dots, P_k(\boldsymbol{u}_k),$$

where P_h is an output relation, and $\boldsymbol{u}_h$, $\boldsymbol{u}_1$, $\boldsymbol{u}_2$, …, $\boldsymbol{u}_k$ are vectors of *variables* of appropriate length. The variables $\boldsymbol{u}_1$, $\boldsymbol{u}_2$, …, $\boldsymbol{u}_k$, $\boldsymbol{u}_h$ appearing in the rule are implicitly universally quantified, and instantiating them with appropriate constants $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, …, $\boldsymbol{v}_k$, $\boldsymbol{v}_h$ yields a grounded constraint g of the form $P_1(\boldsymbol{v}_1) \wedge P_2(\boldsymbol{v}_2) \wedge \dots \wedge P_k(\boldsymbol{v}_k) \implies P_h(\boldsymbol{v}_h)$: *"If all of the antecedent tuples $A_g = \{P_1(\boldsymbol{v}_1), P_2(\boldsymbol{v}_2), \dots, P_k(\boldsymbol{v}_k)\}$ are derivable, then the conclusion $c_g = P_h(\boldsymbol{v}_h)$ is also derivable."*

4.2.2 Synthesis as Rule Selection

The input-output examples, I, O_+, and O_-. Instead of explicitly providing rules r_1 and r_2, the user provides an example instance of the EDB I, and labels a few tuples of the output relation as "desirable" or "undesirable" respectively:

$$O_+ = \{\texttt{samegen}(\texttt{Ann}, \texttt{Jim})\}, \text{ and}$$

$$O_- = \{\texttt{samegen}(\texttt{Ava}, \texttt{Tom}), \texttt{samegen}(\texttt{Jim}, \texttt{Emma})\},$$

indicating that `Ann` and `Jim` are from the same generation, but `Ava` and `Tom` and `Jim` and `Emma` are not. Note that the user is free to label as many potential output tuples as they wish, and the provided labels $O_+ \cup O_-$ need not be exhaustive. The goal of the program synthesizer is to find a set of rules R_s which produce all of the desired output tuples, i.e., $O_+ \subseteq R_s(I)$, and none of the undesired tuples, i.e., $O_- \cap R_s(I) = \emptyset$. We assume that labels from the user are *noise-free*[5], that is, O_+ and O_- should be disjoint.

The set of candidate rules, R. The user often possesses additional information about the problem instance and the concept being targeted. This information can be provided to the synthesizer through various forms of *bias*, which direct the search towards desired parts of the search space. A particularly common form in the recent literature on program synthesis is syntactic: for example, SyGuS requires a description of the space of potential solution programs as a context-free grammar [14], and recent ILP systems such as Metagol [128] require the user to provide a set of higher-order rule templates ("*meta-rules*") and order constraints over predicates and variables that appear in clauses. In the next section, we elaborate how to systematically collect a rich set of candidate rules via a technique called *metarule augumentation*. For now let's assume that a large set of candidate rules R are available

[5] An approach to relaxing this assumption is presented in section 4.5.

and that the target concept R_s is a subset of these rules: $R_s \subseteq R$.

These candidate rules can express various patterns that could conceivably discharge the problem instance. For example, R can include the candidate rule r_s, "$\texttt{samegen}(x, y) :- \texttt{samegen}(y, x)$", which indicates that the output relation is symmetric, and the candidate rule r_t, "$\texttt{samegen}(x, z) :- \texttt{samegen}(x, y), \texttt{samegen}(y, z)$", which indicates that the relation is transitive. Note that the assumption of the candidate rule set R uniformly subsumes many previous forms of syntactic bias, including those in SyGuS and Metagol.

Problem 4.2.1 (Rule Selection)**.** Let the following be given: (*a*) a set of input relations, $\mathcal{I}$ and output relations, $\mathcal{O}$, (*b*) the set of input tuples I, (*c*) a set of positive output tuples O_+, (*d*) a set of negative output tuples O_-, and (*e*) a set of candidate rules R which map the input relations $\mathcal{I}$ to the output relations $\mathcal{O}$. Find a set of target rules $R_s \subseteq R$ such that:

$$O_+ \subseteq R_s(I), \text{ and } O_- \cap R_s(I) = \emptyset.$$

Finally, we note that the rule selection problem is NP-hard: this is because multiple rules in the target program R_s may interact in non-compositional ways. The proof proceeds through a straightforward encoding of the satisfiability of a 3-CNF formula, and is provided in the Appendix.

Theorem 4.2.2. *Determining whether an instance of the rule selection problem, ($\mathcal{I}$, $\mathcal{O}$, I, O_+, O_-, R), admits a solution is NP-hard.*

4.3 Systematic Candidate Rule Generation

A straightforward way for rule generation is to enumerate all possible rules according to the rule syntax usually in the increasing order of size. However, candidate rule generation is just the first phase of synthesis, and the second phase is an ex-

pensive rule selection process. On one hand, the set of generated candidate rules should be small enough so that rule selection process can finish in a reasonable amount of time. On the other hand, candidate rules should be rich enough so that there exists at least one valid program. To have a reasonable balance between these two factors, Metagol [128] relies on a proper set of meta-rules provided by the user. Even though the user may have a rough idea about the syntactic structure of rules to be synthesized, it is challenging to determine the structure exactly in advance.

We next present a technique called *meta-rule augmentation*, which is a systematic way to generate all possible candidate rules even if the user cannot provide any meta-rules. If an initial structure bias from the user is possible, our technique can leverage that and significantly reduce the subsequent rule selection space.

Meta-rules. A *meta-rule* is a *second-order rule*. Multiple rules can be instantiated from a meta-rule. We shall use V_1 and V_2 to denote first- and second-order variables, respectively. A meta-rule takes the following form:

$$R_1(x_1, \ldots, x_{m_1}) \text{ :- } R_2(y_1, \ldots, y_{m_2}), \ldots, R_n(z_1, \ldots, z_{m_n}).$$

where $x_i, y_i, z_i \in V_1$ and $R_i \in V_2$.

A meta-rule can be instantiated by substituting second-order variables with relation symbols. For example, the rules from the running example are generated by the following meta-rules:

$$T_1 : R_0(x, y) \text{ :- } R_1(x, z), R_2(y, z).$$
$$T_2 : R_0(x, u) \text{ :- } R_1(x, y), R_2(u, v), R_3(y, v).$$

Meta-rule augmentation. The choice of meta-rules dictates the effectiveness of any rule selection algorithm in the following stage. If the set of meta-rules is too

large, then scalability might be an issue, since the search space will be huge. On the other hand, the meta-rules must be sufficiently rich to capture the desired program. Simply reusing meta-rules that are either provided by the end-user or mined from existing code repositories is usually insufficient. To solve this problem, we start with a very small set of intuitive meta-rules that are supplied by default (i.e. the chain meta-rule) or given by the user, and then extend these using *augmentation*, a process that slightly modifies each meta-rule.

An augmentation T' of a meta-rule T is a meta-rule where each atom $R(x_1, \ldots, x_k)$ in T is replaced by another atom $R(y_1, \ldots, y_\ell)$. However, we must take care to limit how much the sequence of variables changes. Denote by $d_R(T, T')$ the *edit distance* between the strings $x_1 \ldots x_k$ and $y_1 \ldots y_\ell$. Then, the *augmentation distance* between T, T' is defined as

$$AD(T, T') = \sum_R d_R(T, T')$$

where R ranges over all atoms in T. Because the edit distance $d_R(\cdot, \cdot)$ is symmetric, it is straightforward to see that the augmentation distance $AD(\cdot, \cdot)$ is also symmetric.

Our key idea is to consider all the augmentations of T that are within a bounded augmentation distance from T. The smaller this bound, the fewer meta-rules will be generated from T.

As an example of augmentation, consider these two meta-rules:

$$\begin{array}{lllll} T_1: & R_0(y) & :- & R_1(z), & R_2(y, z). \\ T_2: & R_0(y, z) & :- & R_1(z, x), & R_2(y, z). \end{array}$$

Then, T_2 is an augmentation of T_1 with distance 2.

The augmentation distance required for synthesizing a program P from an initial set of meta-rules $\mathbf{T}$ is:

$$AD(P, \mathbf{T}) = \max_{T_1} \min_{T_2 \in \mathbf{T}} AD(T_2, T_1)$$

where T_1 ranges over all meta-rules that can be instantiated to at least one rule in P. In our experiments, we could synthesize almost all of the programs using an augmentation distance of 5 from three simple chain meta-rules as follows.

$$\begin{aligned} R_0(v_1, v_2) &:- R_1(v_1, v_2). \\ R_0(v_1, v_3) &:- R_1(v_1, v_2), \quad R_2(v_2, v_3). \\ R_0(v_1, v_4) &:- R_1(v_1, v_2), \quad R_2(v_2, v_3), \quad R_3(v_3, v_4). \end{aligned}$$

Note that with a sufficient large augmentation distance, chain meta-rules can be used to generate any meta-rules.

Predicate invention. Another orthogonal way to improve the richness of candidate rules is predicate invention. Predicate invention helps to break a complex rule into simpler ones, and thereby enables to reuse existing meta-rules. More importantly, it is unavoidable for Datalog programs with recursion. For instance, consider the following program which computes strongly connected components (SCC) in a directed graph:

$$\begin{aligned} \texttt{path}(x, y) &:- \texttt{edge}(x, y). \\ \texttt{path}(x, z) &:- \texttt{path}(x, y), \texttt{edge}(y, z). \\ \texttt{scc}(x, y) &:- \texttt{path}(x, y), \texttt{path}(y, x). \end{aligned}$$

Here, the input and output relations are `edge` and `scc`, respectively. Given that `scc` cannot be derived by any set of clauses in terms of only the input relation `edge`, a new predicate `path` must be invented. The difficulty with predicate invention lies in determining what form the invented predicates should take. Without meta-rules, we have no way to effectively constrain the syntax of such predicates. With meta-rules, we can easily support predicate invention: the rules that define the potential invented predicates are exactly the instantiations of meta-rules with concrete relations.

4.4 Rule Selection by Bi-directional Search

Any combination of candidate rules forms a candidate program, however, explicitly enumerating these combinations will be prohibitive. In this section, we present a bi-directional synthesis algorithm, which maintains succinct over- and under-approximations of candidate programs that are consistent with currently observed examples. These two approximations are much smaller than the size of the search space due to the structure defined through logical entailment. We next present the search space structure and then illustrate how bi-directional synthesis algorithm efficiently explore the space.

4.4.1 Structure of the Search Space

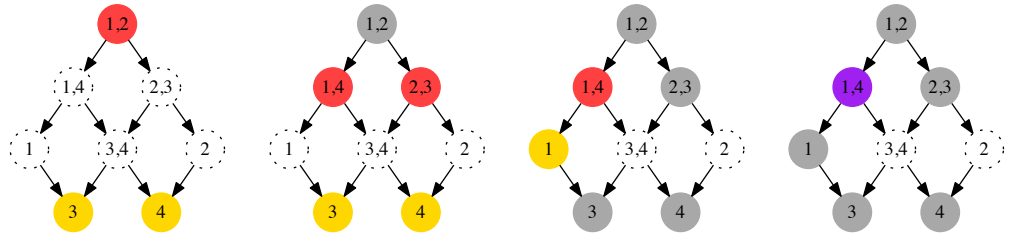
The hypothesis space $\mathcal{H}$ consists of a finite set of Datalog programs over the same input and output relations. For our running example (Example 3.1.1), we consider a simple hypothesis space where all programs use a subset of the following four rules:

$$r_1: \texttt{path}(x, y) \text{ :- } \texttt{edge}(x, y).$$
$$r_2: \texttt{path}(x, z) \text{ :- } \texttt{path}(y, z).$$
$$r_3: \texttt{path}(x, x) \text{ :- } \texttt{edge}(x, x).$$
$$r_4: \texttt{path}(x, y) \text{ :- } \texttt{path}(x, z), \texttt{path}(z, y).$$

We denote the Datalog program consisting of rules r_i, r_j, r_k as P_{ijk}.

Generality order. We structure the search by imposing a generality order on the space of Datalog programs. To define this order, we use θ-subsumption [135], which is a syntactic approach for deciding whether one rule subsumes (is more general than) another rule.

Formally, a rule C *subsumes* another rule D iff there is a variable substitution θ



(a) Initialization path(4,2)? No **(b) Iteration 1 path(1,2)? Yes** **(c) Iteration 2 path(1,4)? Yes** **(d) Iteration 3 converged**

Figure 4.2: Version space in each iteration (red/yellow nodes represent most-general/specific programs in the current iteration; purple nodes represent programs that are both most general and most specific in the current iteration; and grey nodes represent programs that have been evaluated). An arrow from u to v means that program u is more general than program v.

such that $C\theta$ has the same head as D, and all atoms in the body of $C\theta$ appear in the body of D.[6] For example, r_2 subsumes r_4 with $\theta = \{\, \mathtt{z/y}, \mathtt{y/z} \,\}$, and r_1 subsumes r_3 with $\theta = \{\, \mathtt{y/x} \,\}$.

Subsumption can be naturally extended from rules to programs. For any two Datalog programs P and Q, P subsumes Q, denoted $Q \sqsubseteq P$, iff for every rule in Q there exists a rule in P that subsumes it. For instance, in our running example, P_{13} subsumes P_{24}.

Given the hypothesis space $\mathcal{H}$, and a generality ordering $\sqsubseteq$, every subset $\mathbf{P}$ of $\mathcal{H}$ forms a quasi-ordered set w.r.t. $\sqsubseteq$. We can now construct a partial order on the quotient set of the equivalence relation (two programs P, Q are equivalent if $P \sqsubseteq Q$ and $Q \sqsubseteq P$).

In our running example, the following equivalence classes are formed w.r.t. θ-subsumption: $\{P_{1234}, P_{123}, P_{124}, P_{12}\}$, $\{P_{143}, P_{14}\}$, $\{P_{13}, P_1\}$, $\{P_{324}, P_{32}\}$, $\{P_{24}, P_2\}$, $\{P_{34}\}$, $\{P_3\}$, and $\{P_4\}$. We restrict the hypothesis space such that it has one representative from each class (any of the programs with the fewest rules) and define

[6] A substitution θ is a set $\{\, v_1/t_1, \cdots, v_n/t_n \,\}$ where the v_i are distinct variables and t_i are terms. Notation $C\theta$ denotes the rule obtained by applying substitution θ on rule C, i.e., for each $v_i/t_i \in \theta$, we replace each occurrence of v_i in C by t_i.

Algorithm 1: The ALPS synthesis algorithm

$(E^+, E^-) \leftarrow (\emptyset, \emptyset)$
$\overline{\mathbf{P}} \leftarrow \mathit{MostGeneral}()$
$\underline{\mathbf{P}} \leftarrow \mathit{MostSpecific}()$
while *true* **do**
  $\mathbf{P} \quad \overline{\mathbf{P}} \cup \underline{\mathbf{P}}$ `// construct committee`
  if $\forall e \in \mathcal{B}.\, D(e, \mathbf{P}) = 0$ **then return** $\mathbf{P}$
  $e^\star \quad \operatorname{argmax}_{e \in \mathcal{B}} D(e, \mathbf{P})$ `// most controversial example`
  $\diamond \quad \mathcal{O}(e^\star)$ `// where` $\diamond \in \{+, -\}$
  $E^\diamond \quad E^\diamond \cup \{e^\star\}$
  $\overline{\mathbf{P}} \quad F^\downarrow(\overline{\mathbf{P}}, E^+, E^-)$ `// top-down refinement`
  $\underline{\mathbf{P}} \quad F^\uparrow(\underline{\mathbf{P}}, E^+, E^-)$ `// bottom-up refinement`

a partial order directly on these representatives instead of the equivalence classes. We can achieve this without any loss of generality since we are discarding only semantically equivalent programs. For our running example, the hypothesis space can now be reformulated as $\{P_{12}, P_{14}, P_1, P_{32}, P_2, P_{34}, P_3, P_4\}$.

Since the generality order is a partial order, there may exist multiple maximal and minimal elements. The set of maximal elements is denoted $\max(\mathbf{P}) = \{\, P \in \mathbf{P} \mid \nexists P' \in \mathbf{P}.\ P \sqsubset P' \,\}$, and we call these the *most-general* programs. Similarly, the set of minimal elements is denoted $\min(\mathbf{P}) = \{\, P \in \mathbf{P} \mid \nexists P' \in \mathbf{P}.\ P' \sqsubset P \,\}$, and we call these the *most-specific* programs. Figure 4.2a shows the initial version space for our running example, where the most-specific and most-general programs are colored yellow and red, respectively.

4.4.2 The Bi-directional Synthesis Algorithm

Our bi-directional synthesis algorithm has an extra advantage since its refinement operation only considers one example during each iteration, which make it suitable for an interactive setting. Thus, a set of positive or negative examples are not required upfront. In what follows, we present our bi-directional synthesis algorithm in the interactive setting. In the case examples are available at the beginning, an

interactive setup can be easily simulated.

Algorithm 1 summarizes our bi-directional synthesis algorithm. It is a *fixpoint* algorithm that maintains a pair $E = (E^+, E^-)$ of positive and negative examples, and a set of most-general programs $\overline{\mathbf{P}}$ and most-specific programs $\underline{\mathbf{P}}$ that are always consistent with E. The examples are initially empty, and $\overline{\mathbf{P}}, \underline{\mathbf{P}}$ are initialized to be the most general and most specific programs respectively (we define this initialization in Section 4.4.3). At every iteration, it adds a (positive or negative) example by querying the oracle $\mathcal{O}$. Then, it invokes two refinement operators $F^{\uparrow}, F^{\downarrow}$ which recalculate the most-general programs and the most-specific programs that agree with the new example (we define the refinement operators in Section 4.4.3). The algorithm stops when no new examples can be added.

The crux of the algorithm is the way we choose the example to query the oracle. The union of two sets of programs $\overline{\mathbf{P}}, \underline{\mathbf{P}}$ forms the *committee* $\mathbf{P}$. The committee then picks the most controversial example $e^\star$. If $\mathcal{O}(e^\star) = +$, then $e^\star$ is added to E^+; otherwise, $e^\star$ is added to E^-. If no controversial example exists, then everyone in the committee agrees; the algorithm terminates and returns set $\mathbf{P}$, which contains all the most-general and most-specific solutions.

In order to determine the most controversial example, we use the metric of *vote entropy*. It is inspired by query-by-committee [159, 66], a greedy yet effective strategy commonly used in active learning [158]. Since there are only two possible labels for an example, we use a simplified definition, which is essentially equivalent to disagreement count.

Definition 4.4.1 (Vote entropy)**.** *For an example e and set of committee members K, the* normalized vote entropy *is:*

$$D(e, K) = 1 - \frac{2}{|K|}\left|p - \frac{|K|}{2}\right|$$

where p is the number of committee members that assign a positive label to the example

e.

When the vote entropy of an example is zero, all programs in the committee agree on its label. Figure 4.2 shows the version space and the query posed in each iteration for our running example.

4.4.3 Refinement with Meta-Rules

We now give concrete definitions of the initialization functions and refinement operators, $F^{\uparrow}$ and $F^{\downarrow}$ in Algorithm 1. The design of the refinement operators is motivated by a practical insight: the synthesis search should be biased towards patterns that are frequently used in practice.

Similar to rules, a generality order between meta-rules can be established using θ-subsumption by allowing substitution for second-order variables as well as first-order variables. Using this generality order, a set of meta-rules forms a partially ordered set.

Initialization. The initialization function *MostGeneral*() collects all rules instantiated from the most general meta-rules and combines them as the most general program. The initialization function *MostSpecific*() makes each individual rule instantiated from the most specific meta-rules as a single rule program, and all of these programs form the initial set of most specific programs.

Meta-rule-guided refinement. Algorithm 2 describes our refinement operations, $F^{\downarrow}$ and $F^{\uparrow}$, which are parameterized by a set of meta-rules $\mathbf{T}$. We explain only top-down refinement $F^{\downarrow}$ in detail, since bottom-up refinement $F^{\uparrow}$ works in a symmetrical manner.

The algorithm begins with the given set of programs $\overline{\mathbf{P}}$. Then, it iteratively *specializes* the programs by applying the specialization operator $\rho^{\downarrow}$, which is guided by $\mathbf{T}$ (line 3–6). In each iteration, the condition $\overline{\mathbf{P}} \not\subseteq \mathcal{V}_E$ checks whether the current

Algorithm 2: Meta-rule-guided refinement

1 **Function** $F^{\downarrow}(\overline{\mathbf{P}}, E^+, E^-)$
2 $E \quad (E^+, E^-)$
3 **while** $\underline{\mathbf{P}} \not\subseteq \mathcal{V}_E$ **do**
4 $\Delta\underline{\mathbf{P}} \quad (\underline{\mathbf{P}} \cap \mathcal{V}_{E^-}) \backslash \mathcal{V}_{E^+}$
5 $\Delta\underline{\mathbf{P}} \leftarrow \rho^{\uparrow}(\Delta\underline{\mathbf{P}}, \mathbf{T}) \cap \mathcal{V}_{E^-}$
6 $\underline{\mathbf{P}} \quad (\underline{\mathbf{P}} \cap \mathcal{V}_E) \cup \Delta\underline{\mathbf{P}}$
7 **return** $\underline{\mathbf{P}}$

8 **Function** $F^{\uparrow}(\underline{\mathbf{P}}, E^+, E^-)$
9 $E \quad (E^+, E^-)$
10 **while** $\underline{\mathbf{P}} \not\subseteq \mathcal{V}_E$ **do**
11 $\Delta\underline{\mathbf{P}} \quad (\underline{\mathbf{P}} \cap \mathcal{V}_{E^-}) \backslash \mathcal{V}_{E^+}$
12 $\Delta\underline{\mathbf{P}} \leftarrow \rho^{\uparrow}(\Delta\underline{\mathbf{P}}, \mathbf{T}) \cap \mathcal{V}_{E^-}$
13 $\underline{\mathbf{P}} \quad (\underline{\mathbf{P}} \cap \mathcal{V}_E) \cup \Delta\underline{\mathbf{P}}$
14 **return** $\underline{\mathbf{P}}$

programs are consistent with the examples. If there is no violation, the algorithm terminates. Otherwise, line 4 first eliminates programs violating positive examples, and then selects programs violating negative examples to specialize. In the former case, programs fail to derive a positive example, and more specific programs will also fail to derive it. This process removes not only inconsistent programs but also any programs more specific than them. The elimination happens in the third iteration of our running example shown in Figure 4.2c: when P_{23} is eliminated due to the positive example `path(1,2)`, all the more specific programs P_{34}, P_2, P_3, P_4 are eliminated from consideration as well.

Next, line 5 specializes programs violating negative examples by calling $\rho^{\downarrow}$, and eliminates any generated programs that fail to derive a positive example. Finally, line 6 updates $\overline{\mathbf{P}}$ by including the new specialized programs.

The final piece of the puzzle is the *specialization operator* $\rho^{\downarrow}$. Here, $\rho^{\downarrow}$ can specialize a program in two ways: (1) replace a rule with a more specific one; for instance, in our running example shown in Figure 4.2b, program P_{12} is specialized to P_{14} and P_{23}; (2) remove a rule that cannot be further specialized; for instance, P_{23} could

potentially be specialized to P_2. Finding all more specific rules for a given rule r can be efficiently done by consulting the generality order of the meta-rules $\mathbf{T}$: first, find the meta-rule T_r used to instantiate r; then, find all more specific meta-rules $\mathbf{T_s}$ with respect to T_r; finally examine all rules instantiated from a meta-rule in $\mathbf{T_s}$ and keep the ones more specific than r.

4.4.4 Properties of ALPS

The ALPS synthesis algorithm (Algorithm 1) always makes progress: each iteration, we resolve a controversial example. Since the set of possible examples is finite, the algorithm always terminates. It also guarantees that a solution is found if there are no controversial examples left in the committee. To ensure this property, it is critical that the algorithm tracks both the most-general and most-specific programs at every iteration. The following theorem succinctly captures these properties. We provide its proof in the Appendix.

Theorem 4.4.2. *Let $S = (\mathcal{I}, \mathcal{O}, I, O_+, O_-, R)$ be a synthesis problem such that there exists a solution to S. Let $\mathbf{P}$ be the output of ALPS. Then:*

1. *(Soundness) Every $P \in \mathbf{P}$ is a solution to S.*
2. *(Completeness) For every solution $P \in \mathcal{H}$ to S, there exist programs $P_l, P_u \in \mathbf{P}$ such that $P_l \sqsubseteq P \sqsubseteq P_u$. An immediate corollary is that if there exists a program P that is a solution to S, then $\mathbf{P}$ is nonempty.*
3. *(Termination) ALPS terminates.*

4.5 A Smoothed Interpretation for Datalog

We next present a dramatically different way of searching through the candidate program space. Instead of carefully maintaining combinations of rules, our new

approach considers all candidate rules simultaneously. The key insight is to turn combinatorial search into numerical optimization. Specifically, we attach numerical weights to candidate rules and design a proper loss function so that rules with high weights are the desired ones in the final solution and the weights can be adjusted by gradient-based methods. This requires us to design a new semantics for Datalog programs whose rules are associated with numerical weights, which we call Difflog.

In this section, we describe the semantics of Difflog, and present an algorithm to evaluate and automatically differentiate this continuous-valued extension.

4.5.1 Relaxing Rule Selection

The idea motivating Difflog is to generalize the concept of rule selection: instead of a set of binary decisions, we associate each rule r with a numerical weight $w_r \in [0, 1]$. One possible way to visualize these weights is as the extent to which they are present in the current candidate program. The central challenge, which we will now address, is in specifying how the vector of rule weights $\boldsymbol{w}$ determines the numerical values $v_t^{R,I}(\boldsymbol{w})$ for the output tuples t of the program. We will simply write $v_t(\boldsymbol{w})$ when the set of rules R and the set of input tuples I are evident from context.

Every output tuple of a Datalog program is associated with a set of derivation trees, such as those shown in Figure 4.3. Let r_g be the rule associated with each instantiated clause g that appears in the derivation tree τ. We define the value of τ, $v_\tau(\boldsymbol{w})$, as the product of the weights of all clauses appearing in τ, and the value of an output tuple t as being the supremum of the values of all derivation trees of which it is the conclusion:

$$v_\tau(\boldsymbol{w}) = \prod_{\text{clause } g \in \tau} w_{r_g}, \text{ and} \tag{4.1}$$

$$v_t(\boldsymbol{w}) = \sup_{\tau \text{ with conclusion } t} v_\tau(\boldsymbol{w}), \tag{4.2}$$

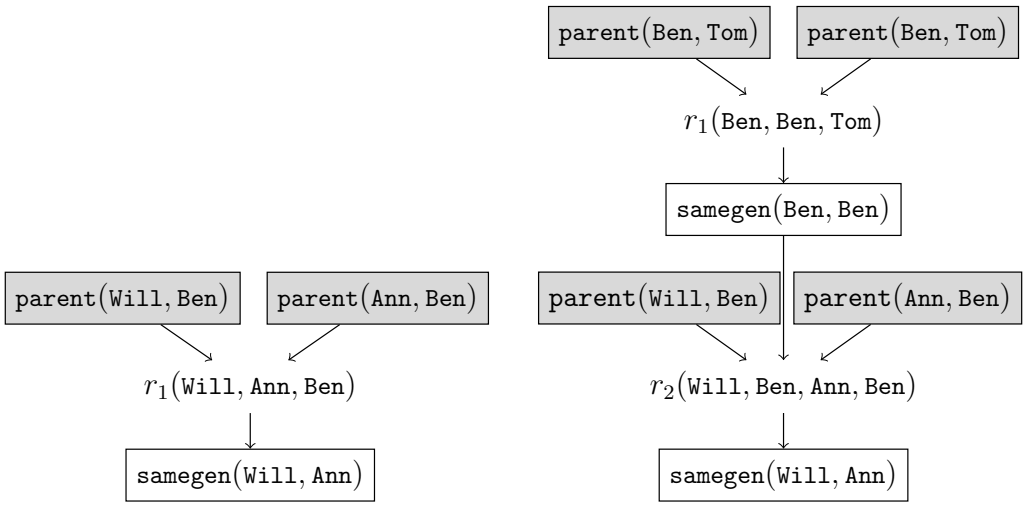


Figure 4.3: Examples of derivation trees, τ_1 (a) and τ_2 (b) induced by various combinations of candidate rules, applied to the EDB of familial relationships from Figure 4.1. The input tuples are shaded in grey. We present two derivation trees for the conclusion $\texttt{samegen}(\texttt{Will}, \texttt{Ann})$ using rules r_1 and r_2 in Section 4.2.1.

with the convention that $\sup(\emptyset) = 0$. For example, if $w_{r_1} = 0.8$ and $w_{r_2} = 0.6$, then the weight of the trees τ_1 and τ_2 from Figure 4.3 are respectively $v_{\tau_1}(\boldsymbol{w}) = w_{r_1} = 0.8$ and $v_{\tau_2}(\boldsymbol{w}) = w_{r_1} w_{r_2} = 0.48$.

Since $0 \leq w_r \leq 1$, it follows that $v_\tau(\boldsymbol{w}) \leq 1$. Also note that a single output tuple may be the conclusion of infinitely many proof trees (see the derivation structure in Figure 4.4), leading to the deliberate choice of the supremum in Equation 4.2.

One way to consider Equations 4.1 and 4.2 is as replacing the traditional operations $(\wedge, \vee)$ and values $\{\text{true}, \text{false}\}$ of the Boolean semiring with the corresponding operations $(\times, \max)$ and values $[0, 1]$ of the Viterbi semiring. The study of various semiring interpretations of database query formalisms has a rich history motivated by the idea of *data provenance*. The following result follows from Prop. 5.7 in [75], and concretizes the idea that Difflog is a refinement of Datalog:

Theorem 4.5.1. *Let R be a set of candidate rules, and $\boldsymbol{w}$ be an assignment of weights $w_r \in [0, 1]$ to each of them, $r \in R$. Define $R_s = \{r \mid w_r \gneq 0\}$, and consider a potential*

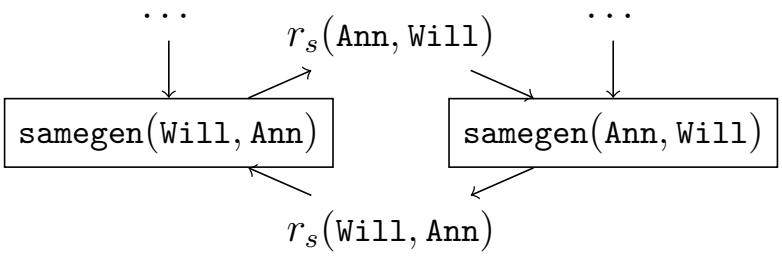


Figure 4.4: The rule r_s, "$\texttt{someone}(x, y) \coloneq \texttt{samegen}(y, x)$", induces cycles in the clauses obtained at fixpoint. When unrolled into derivation trees such as those in Figure 4.3, these cycles result in the production of infinitely many derivation trees for a single output tuple.

output tuple t. Then, $v_t^{R,I}(\boldsymbol{w}) \gtrsim 0$ iff $t \in R_s(I)$.

Furthermore, in the Appendix, we show that the output values $v_t(\boldsymbol{w})$ is well-behaved in its domain of definition:

Theorem 4.5.2. *The value of the output tuples, $v_t(\boldsymbol{w})$, varies monotonically with the rule weights $\boldsymbol{w}$, and is continuous in the region $0 < w_r < 1$.*

Note that Difflog is discontinuous at boundary points when $w_r = 0$ or $w_r = 1$, and undefined outside the unit interval. To prevent this from causing problems during learning with gradient descent, we clamp the rule weights to the interval $[0.01, 0.99]$ in our implementation.

We could conceivably have chosen a different semiring in our definitions in Equations 4.1 and 4.2. One alternative would be to choose a space of events, corresponding to the inclusion of individual rules, and choosing the union and intersection of events as the semiring operations. This choice would make the system coincide with ProbLog [57]. However, the #P-completeness of inference in probabilistic logics would make the learning process computationally expensive. Other possibilities, such as the arithmetic semiring $(\mathbb{R}, +, \times, 0, 1)$, would lead to unbounded values for output tuples in the presence of infinitely many derivation trees.

4.5.2 Evaluation and Automatic Differentiation

Because the set of derivation trees for an individual tuple t may be infinite, note that Equation 4.2 is merely *definitional*, and does not prescribe an algorithm to *compute* $v_t(\boldsymbol{w})$. Furthermore, numerical optimization requires the ability to automatically differentiate these values, i.e., to compute $\nabla_{\boldsymbol{w}} v_t$.

The key to automatic differentiation is tracking the *provenance* of each output tuple [75]. Pick an output tuple t, and let τ be its derivation tree with the greatest value. Note that τ may not be unique and would be chosen randomly when that is the case. We model the provenance of t as a map, $l_t = \{r \mapsto \#r \text{ in } \tau \mid r \in R\}$, which maps each rule r to the number of times it appears in τ. Given the provenance l_t of a tuple, observe that $v_t(\boldsymbol{w}) = \prod_{r \in R} w_r^{l_t(r)}$, so that the derivative of $v_t(\boldsymbol{w})$ can be readily computed as follows:

$$\frac{\partial v_t(\boldsymbol{w})}{\partial w_r} = \frac{l_t(r) v_t(\boldsymbol{w})}{w_r}. \tag{4.3}$$

In Algorithm 3, we present an algorithm to compute the output values $v_t(\boldsymbol{w})$ and provenance l_t, given R, $\boldsymbol{w}$, and the input tuples I. The algorithm is essentially an instrumented version of the "naive" Datalog evaluator [7]. We outline the proof of the following correctness and complexity claims in the Appendix.

Theorem 4.5.3. *Fix a set of input relations $\mathcal{I}$, output relations $\mathcal{O}$, and candidate rules R. Let* Evaluate$(R, \boldsymbol{w}, I) = (F, \boldsymbol{u}, \boldsymbol{l})$. *Then: (a) $F = R(I)$, and (b) $\boldsymbol{u}(t) = v_t(\boldsymbol{w})$. Furthermore,* Evaluate$(R, \boldsymbol{w}, I)$ *returns in time $poly(|I|)$.*

4.6 Formulating the Optimization Problem

We formulate the Difflog synthesis problem as finding the value of the rule weights $\boldsymbol{w}$ which minimizes the difference between the output values of tuples, $v_t(\boldsymbol{w})$, and their expected values, 1 if $t \in O_+$, and 0 if $t \in O_-$. Specifically, we seek to minimize

Algorithm 3: EVALUATE$(R, \boldsymbol{w}, I)$, where R is a set of rules, $\boldsymbol{w}$ is an assignment of weight to each rule in R, and I is a set of input tuples.

1. Initialize the set of tuples in each relation, $F_P \coloneqq \emptyset$, their valuations, $u(t) \coloneqq 0$, and their provenance $l(t) = \{r \mapsto \infty \mid r \in R\}$.
2. For each input relation P, update $F_P \coloneqq I_P$, and for each $t \in I_P$, update $u(t) \coloneqq 1$ and $l(t) = \{r \mapsto 0 \mid r \in R\}$.
3. Until $(F, \boldsymbol{u})$ reach fixpoint,
   (a) Compute the immediate consequence of each rule, r, "$P_h(\boldsymbol{u}_h) \coloneq P_1(\boldsymbol{u}_1), P_2(\boldsymbol{u}_2), \ldots, P_k(\boldsymbol{u}_k)$":
   $$F'_{P_h} = \pi_{\boldsymbol{u}_h}(F_{P_1}(\boldsymbol{u}_1) \bowtie F_{P_2}(\boldsymbol{u}_2) \bowtie \cdots \bowtie F_{P_k}(\boldsymbol{u}_k)).$$
   Furthermore, for each tuple $t \in F'_{P_h}$, determine all sets of antecedent tuples, $A_g(t) = \{P_1(\boldsymbol{v}_1), P_2(\boldsymbol{v}_2), \ldots, P_k(\boldsymbol{v}_k)\}$, which result in its production.
   (b) Update $F_{P_h} \coloneqq F_{P_h} \cup F'_{P_h}$.
   (c) For each tuple $t \in F'_{P_h}$ and each $A_g(t)$: (*i*) compute $u'_t = w_r \prod_{i=1}^{k} u(P_i(\boldsymbol{v}_i))$, and (*ii*) if $u(t) < u'_t$, update:
   $$u(t) \coloneqq u'_t, \text{ and } l(t) \coloneqq \{r \mapsto 1\} + \sum_{i=1}^{k} l(P_i(\boldsymbol{v}_i)),$$
   where addition of provenance values corresponds to the element-wise sum.
4. Return $(F, \boldsymbol{u}, \boldsymbol{l})$.

the L2 loss,

$$L(\boldsymbol{w}) = \sum_{t \in O_+} (1 - v_t(\boldsymbol{w}))^2 + \sum_{t \in O_-} v_t(\boldsymbol{w})^2. \tag{4.4}$$

At the optimum point, Theorem 4.5.1 enables the recovery of a classical Datalog program from the optimum value $\boldsymbol{w}^*$.

Hybrid optimization procedure. In program synthesis, the goal is often to ensure exact compatibility with the provided positive and negative examples. We therefore seek zeros of the loss function $L(\boldsymbol{w})$, and solve for this using Newton's root-finding

algorithm: $\boldsymbol{w}^{(i+1)} \coloneqq \boldsymbol{w}^{(i)} - L(\boldsymbol{w})\nabla_{\boldsymbol{w}} L(\boldsymbol{w})/\|\nabla_{\boldsymbol{w}} L(\boldsymbol{w})\|^2$. To escape from local minima and points of slow convergence, we periodically intersperse iterations of the MCMC sampling, specifically simulated annealing.

Forbidden rules. If a single rule $r \in R$ is seen to independently derive an undesirable tuple $t \in O_-$, i.e., if $l_t(r) \geq 1$ and $l_t(r') = 0$ for all $r \neq r$, then it is marked as a *forbidden* rule, and its weight is immediately clamped to 0: $w_r^{(i+1)} \coloneqq 0$.

Learning details. We initialize $\boldsymbol{w}$ by uniformly sampling weights $w_r \in [0.25, 0.75]$. We apply MCMC sampling after every 30 iterations of Newton's root-finding method, and sample new weights as follows:

$$X \sim U(0,1)$$

$$w_{new} = \begin{cases} w_{old}\sqrt{2X} & \text{if } X < 0.5 \\ 1 - (1 - w_{old})\sqrt{2(1-X)} & \text{otherwise.} \end{cases}$$

The temperature T used in simulated annealing is as follows:

$$T = \frac{1.0}{C * log(5 + \#iter)}$$

where C is initially 0.0001 and $\#iter$ is the number of iterations. We accept the newly proposed sample with probability

$$p_{acc} = \min(1, \pi_{new}/\pi_{curr}),$$

where $\pi_{curr} = \exp(-L_2(\boldsymbol{w}_{curr})/T)$ and $\pi_{new} = \exp(-L_2(\boldsymbol{w}_{new})/T)$.

Separation-guided search termination. After computing each subsequent $\boldsymbol{w}^{(i)}$, we examine the provenance values for each output tuple to determine whether the current position can directly lead to a solution to the rule selection problem. In

particular, we compute the sets of desirable—$R_+ = \{r \in l(t) \mid t \in O_+\}$—and undesirable rules—$R_- = \{r \in l(t) \mid t \in O_-\}$, and check whether $R_+ \cap R_- = \emptyset$. If these sets are separate, then we examine the candidate solution R_+, and return if it satisfies the output specification.

4.7 Empirical Evaluation

We evaluate ALPS and Difflog on a variety of synthesis tasks from different domains.

Implementation and setup. ALPS[7] comprises about 8,000 lines of C++ code and uses the fixpoint engine of the Z3 SMT solver [90] for Datalog evaluation. Difflog[8] comprises 4K lines of Scala code. We use Newton's root-finding method for continuous optimization and apply MCMC-based random sampling every 30 iterations. All experiments were conducted on Linux machines with Intel Xeon 3GHz processors and 64GB memory.

Our experiments address the following aspects:

1. effectiveness of our baseline ALPS compared with existing state-of-the-art synthesis tools [10, 128];
2. effectiveness of Difflog in constrast to ALPS;
3. the benefit of employing MCMC search compared to a purely gradient-based method; and
4. scaling with number of training labels and rule templates.

4.7.1 Benchmark Suite

We collected 34 synthesis tasks from three different application domains: (*i*) knowledge discovery, (*ii*) program analysis and (*iii*) relational queries. Table 4.1 presents

[7]The artifacts of ALPS are available at: https://github.com/XujieSi/fse18-artifact-183.
[8]The artifacts of Difflog are available at: https://github.com/petablox/difflog.

Benchmark	Brief description	#Rel.	#Rule	Rec.?
Knowledge Discovery				
`inflammation`	diagnosis of bladder inflammation	7	2	
`abduce`	grandparent of given father/mother [126]	4	3	
`animals`	distinguishing classes of animals [126]	13	4	
`ancestor`	ancestor in a family tree [128]	4	4	✓
`buildWall`	learn a stable wall strategy [128]	5	4	✓
`samegen`	same generation in a family tree [7]	3	3	✓
`path`	all-pairs reachability in directed graph	2	2	✓
`scc`	compute SCCs in directed graph	3	3	✓
Program Analysis				
`polysite`	polymorphic call-site inference for Java	6	3	
`downcast`	downcast safety checker for Java	9	4	
`rv-check`	return-value-checker in APISan [191]	5	5	
`andersen`	inclusion-based pointer analysis for C [19]	5	4	✓
`1-call-site`	1-call-site pointer analysis for Java [183]	9	4	✓
`2-call-site`	2-call-site pointer analysis for java [183]	9	4	✓
`1-object`	1-object-sensitive pointer analysis [122]	11	4	✓
`1-type`	1-type-sensitive pointer analysis [171]	12	4	✓
`1-obj-type`	1-type-1-object sensitive analysis [171]	13	5	✓
`escape`	escape analysis for Java	10	6	✓
`modref`	mod-ref analysis for Java	13	10	✓
Relational Queries				
`sql-1` $\sim$ `15`	15 SQL queries [180]	≤ 7	≤ 4	

Table 4.1: Benchmark characteristics.

useful characteristics of these benchmarks. The last three columns show the number of input–output relations, the number of rules of the smallest desired program, and whether the desired program is recursive or not, respectively.

Knowledge discovery. The knowledge discovery benchmarks comprise 8 tasks of synthesizing Datalog programs frequently used in the artificial intelligence and database literature. The goal of the first benchmark `inflammation` is to discover interesting correlations between patient risk factors and a disease called acute inflammations of urinary bladder. We used a dataset created by a medical expert to enable

expert systems that perform presumptive diagnosis of the disease [51].[9] The next four benchmarks (`abduce`, `ancestor`, `animals`, and `buildWall`) are widely used in the field of inductive logic programming [126, 128]. The `samegen` benchmark is a standard Datalog program in the database literature [7]. The `path` benchmark is the problem described in Example 3.1.1 and the `scc` benchmark is the problem of computing strongly connected components in a directed graph.

Program analysis. The program analysis benchmarks comprise 11 tasks of synthesizing static analyzers written in Datalog:

- `polysite` is a polymorphic call-site inference analysis for Java;
- `downcast` is a downcast safety checker for Java;
- `rv-check` is the static API misuse detector described in Example 3.1.5, which is motivated from a return value checker used in a tool called APIsan [191]. APIsan identifies API misuses by detecting inconsistent uses of the return values of API functions. However, the tool is neither sound nor complete due to the limitation of its statistical method. This observation motivated our rule-based approach for static API misuse detection.
- `andersen` is a classic pointer analysis for C [19];
- The next five benchmarks are pointer analyses for Java with various context abstractions [183, 122, 171].
- `modref` is a mod-ref analysis for Java and `escape` is an escape analysis for Java. Both benchmarks originated from a programming assignment in an online course on program analysis [5].

[9] Available at http://archive.ics.uci.edu/ml/datasets/Acute+Inflammations.

Relational queries. These benchmarks comprise 15 synthesis tasks from Stack Overflow posts and textbook examples [180]. We chose the 15 tasks of synthesizing SQL queries that can be expressed in Datalog. Each task involves up to 6 input tables and one output table. The desired Datalog programs comprise up to four rules.

4.7.2 Effectiveness of ALPS

We first compare ALPS with two state-of-the-art ILP tools: Metagol [49] and Zaatar [10]. We supply both of these tools with all ground facts upfront since they are non-interactive.

Table 4.2 presents the overall evaluation results of ALPS. One unique advantage of ALPS is its capability of synthesizing all programs in the search space that are consistent with given positive and negative examples. The second column (**#syn. programs**) shows the number of correct programs synthesized by ALPS. The third and fourth columns show the number of candidate programs that are evaluated by ALPS and the number of all candidate programs in the search space, respectively. As we can see, only an extremely small fraction of candidate programs are visited during the bi-directional synthesis process. The last four columns show the running times taken by ALPS, Metagol and Zaatar. In general, ALPS can synthesize most benchmarks in a few minutes.

	#syn. programs	#eval. programs	search space	ALPS time (sec.)	Metagol time same	Metagol time ideal	Zaatar time (sec.)
inflammation	4	2327	10^6	4.3	0.51	0.47	***timeout***
abduce	1	4613	10^6	3.36	***timeout***	0.43	***timeout***
animals	2	45152	10^6	75.8	0.46	0.42	***timeout***
ancestor	3	24280	10^{10}	24.6	***timeout***	0.43	***timeout***
buildWall	13	61654	10^{10}	128.7	***timeout***	35.1	***timeout***
samegen	2	110338	10^9	22.3	***timeout***	***timeout***	4.77
path	3	384	10^4	0.26	***timeout***	0.43	26.43
scc	4	57013	10^6	88.7	***timeout***	***timeout***	***timeout***
polysite	5	27432	10^{22}	130.0	***timeout***	0.43	***timeout***
downcast	1	56489	10^{28}	299.8	***timeout***	0.43	***timeout***
rv-check	1	393740	10^{29}	361.5	***timeout***	***timeout***	***timeout***
andersen	1	100345	10^{20}	148.0	***timeout***	***timeout***	295.31
1-call-site	3	99697	10^{32}	178.3	***timeout***	***timeout***	***timeout***
2-call-site	1	184824	10^{53}	601.8	***timeout***	***timeout***	***timeout***
1-object	1	93362	10^{48}	705.1	***timeout***	***timeout***	***timeout***
1-type	2	10038	10^{30}	21.6	***timeout***	***timeout***	***timeout***
1-obj-type	-	-	10^{51}	***timeout***	***timeout***	***timeout***	***timeout***
escape	12	5706	10^{34}	9.9	***timeout***	***timeout***	***timeout***
modref	1	1346754	10^{45}	5307	***timeout***	***timeout***	***timeout***
sql-1	1	30	10^6	0.07	0.01	0.01	43.65
sql-2	1	7	10^6	0.02	0.01	0.01	***timeout***
sql-3	1	1	10^1	0.03	0.01	0.01	***timeout***
sql-4	1	19	10^2	0.02	0.01	0.01	***timeout***
sql-5	1	1	10^2	0.01	0.01	0.01	***timeout***
sql-6	1	44	10^2	0.03	0.01	0.01	***timeout***
sql-7	1	1	10^1	0.01	0.01	0.01	***timeout***
sql-8	2	230	10^{16}	1.60	0.02	0.01	***timeout***
sql-9	1	9	10^{16}	0.30	***timeout***	0.01	6260
sql-10	1	778	10^{23}	63.2	***timeout***	0.01	***timeout***
sql-11	6	1192	10^{18}	1.86	***timeout***	0.04	8320
sql-12	1	117	10^{15}	0.20	***timeout***	***timeout***	2417
sql-13	1	4	10^3	0.01	***timeout***	***timeout***	***timeout***
sql-14	1	13	10^{25}	90.9	***timeout***	***timeout***	***timeout***
sql-15	1	344	10^{15}	17.7	***timeout***	***timeout***	***timeout***

Table 4.2: The performance results of ALPS, Metagol and Zaatar; the timeout limit is 3 hours.

Metagol is an ILP tool that is an instance of the *meta-interpretive learning* framework [128], which is also parameterized by meta-rules. We run Metagol with two settings: the ALPS setting, which uses the same set of meta-rules that ALPS uses after it performs augmentation, and the ideal setting, which consists of the minimal set of meta-rules that are sufficient for synthesizing a correct program. Using

ALPS's setting, Metagol cannot finish most knowledge discovery benchmarks and all program analysis benchmarks. Using the ideal setting, Metagol still fails on two knowledge discovery benchmarks and most of program analysis benchmarks. Metagol also fails on four of the SQL benchmarks despite their lack of recursion. It is important to note that Metagol employs meta-interpretive learning, which is not a complete technique, so it is not guaranteed to terminate, despite finiteness of the search space.

Zaatar [10] is a constraint-based Datalog program synthesis tool. It fails on most of our benchmarks because it is very sensitive to the size of the input data, since the size of the encoding is polynomial in the input data. In contrast, ALPS has much better scalability in terms of input size, as ALPS only evaluates candidate programs on input data instead of encoding the input as symbolic constraints.

4.7.3 Effectiveness of Difflog

We just show that ALPS significantly outperforms two state-of-the-art ILP tools. We next compare Difflog with ALPS and show that numerical relaxation could further improve the performance dramatically.

The running time and solution of Difflog depends on the random choice of initial weights. Difflog exploits this characteristic by running multiple synthesis processes for each problem in parallel. The solution is returned once any one of the parallel processes successfully synthesizes a Datalog program which is consistent with the specifications. We populated 32 processes in parallel and measured the running time until the first solution was found. The timeout is set to 1 hour for each problem.

Table 4.3 shows the running of Difflog and ALPS. Of the 34 benchmarks, we excluded 14 benchmarks where either both Difflog and ALPS find solutions within a second (13 benchmarks) or both solvers time-out (1 benchmark). Difflog outperforms ALPS on 19 of the remaining 20 benchmarks in Table 4.3. In particular, Difflog is orders of magnitude faster than ALPS on most of the program analysis

Benchmark	Rel	Rule		Tuple		Difflog			ALPS
		Exp	Cnd	In	Out	Iter	Smpl	Time	Time
inflamation	7	2	134	640	49	1	0	**1**	2
abduce	4	3	80	12	20	1	0	$<$ **1**	2
animals	13	4	336	50	64	1	0	**1**	40
ancestor	4	4	80	8	27	1	0	$<$ **1**	14
buildWall	5	4	472	30	4	5	1	**7**	67
samegen	3	3	188	7	22	1	0	**2**	12
scc	3	3	384	9	68	6	1	**28**	56
polysite	6	3	552	97	27	17	1	**27**	84
downcast	9	4	1,267	89	175	5	1	**30**	1,646
rv-check	5	5	335	74	2	1,205	41	**22**	195
andersen	5	4	175	7	7	1	0	**4**	27
1-call-site	9	4	173	28	16	4	1	**4**	106
2-call-site	9	4	122	30	15	25	1	**53**	676
1-object	11	4	46	40	13	3	1	**3**	345
1-type	12	4	70	48	22	3	1	**4**	13
escape	10	6	140	13	19	2	1	**1**	5
modref	13	10	129	18	34	1	0	**1**	2,836
sql-10	3	2	734	10	2	7	1	**11**	41
sql-14	4	3	23	11	6	1	0	$<$ **1**	54
sql-15	4	2	186	50	7	902	31	875	**11**

Table 4.3: Characteristics of benchmarks and performance of Difflog compared to ALPS. Rel shows the number of relations. The columns titled Rule represent the number of expected and candidate rules. Tuple shows the number of input and output tuples. Iter and Smpl report the number of iterations and MCMC samplings. Time shows the running time of Difflog and ALPS in seconds.

benchmarks. Meanwhile, the continuous optimization may not be efficient when the problem has many local minimas and the space is not convex. For example, `sql-15` has a lot of sub-optimal solutions that generate not only all positive output tuples but also some negative ones.

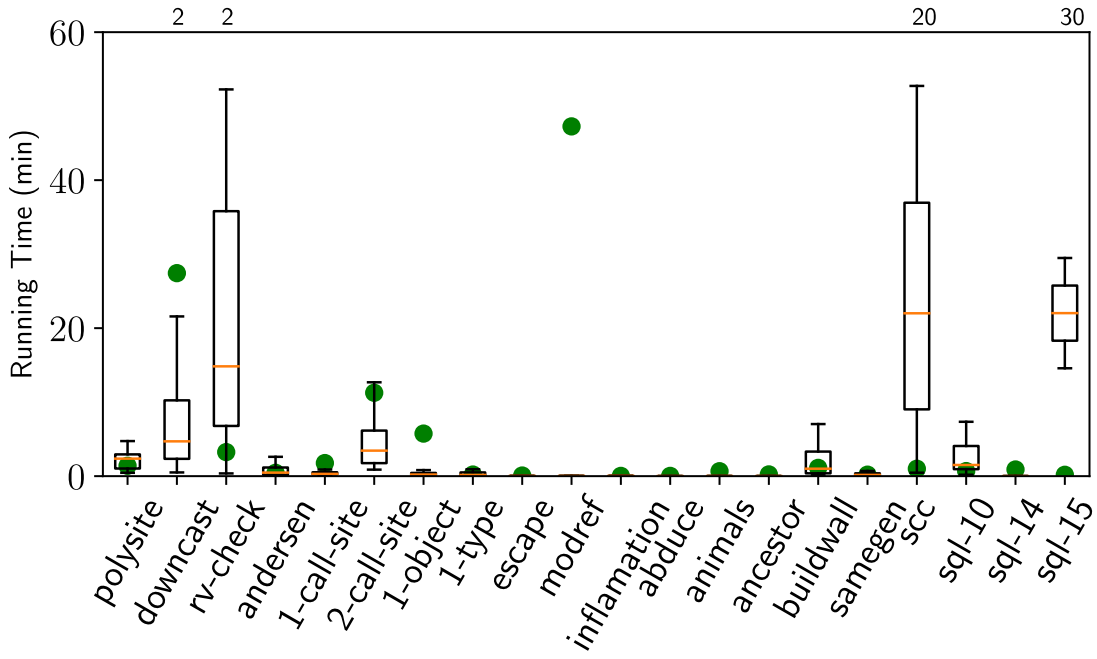


Figure 4.5: Distribution of Difflog's running time from 32 parallel runs. The numbers on top represents the number of timeouts. Green circles represent the running time of ALPS.

Figure 4.5 depicts the distribution of running time on the benchmarks. The results show that Difflog is always able to find solutions for all the benchmarks except for occasional timeouts on `downcast`, `rv-check`, `scc`, and `sql-15`. Also note that even the median running time of Difflog is smaller than the running time of ALPS for 13 out of 20 benchmarks.

4.7.4 Impact of MCMC-based Sampling

Next, we evaluate the impact of our MCMC-based sampling by comparing the performance of three variants of Difflog: (*a*) a version that uses both Newton's method and the MCMC-based technique (**Hybrid**), which is the same as in Section 4.7.3, (*b*) a version that uses only Newton's method (**Newton**), and (*c*) a version that uses only the MCMC-based technique (**MCMC**). Table 4.4 shows the running time of the best run and the number of timeouts among 32 parallel runs for these three variants. The table shows that our hybrid approach strikes a good balance between

Benchmark	Hybrid			Newton			MCMC		
	Best	Median	Timeout	B	M	T	B	M	T
`polysite`	27s	142s	0	10s	72s	0	12s	76s	0
`downcast`	30s	310s	2	16s	252s	9	70s	268s	7
`rv-check`	22s	948s	2	N/A	N/A	32	N/A	N/A	32
`andersen`	4s	29s	0	3s	15s	10	4s	17s	9
`1-call-site`	4s	18s	0	8s	18s	1	N/A	N/A	32
`2-call-site`	53s	225s	0	27s	N/A	17	42s	94s	9
`1-object`	3s	17s	0	3s	N/A	17	N/A	N/A	32
`1-type`	4s	12s	0	3s	N/A	18	N/A	N/A	32
`escape`	1s	2s	0	1s	N/A	17	N/A	N/A	32
`modref`	1s	2s	0	1s	1s	4	N/A	N/A	32
Total			4			125			217

Table 4.4: Effectiveness of MCMC sampling in terms of the best and median running times and the number of timeouts observed over 32 independent runs.

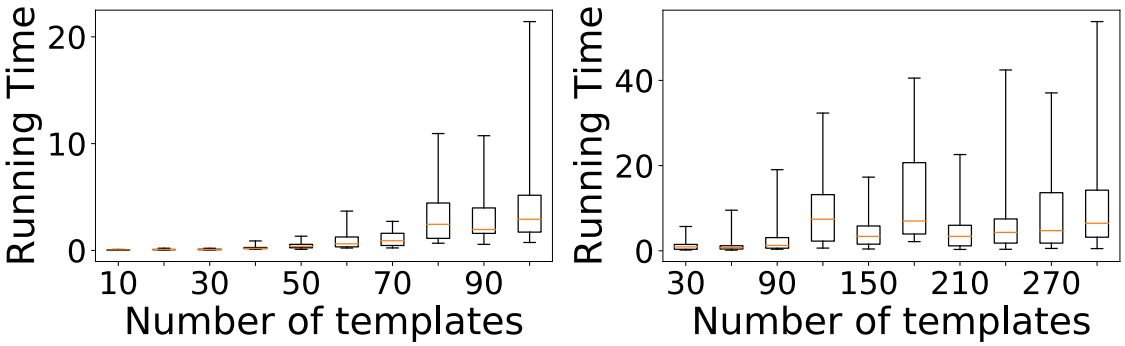
exploitation and exploration. In many cases, **Newton** gets stuck in local minima; for example, it cannot find any solution for `rv-check` within one hour. **MCMC** cannot find any solution for 6 out of 10 benchmarks. Overall, **Hybrid** outperforms both **Newton** and **MCMC** by reporting 31× and 54× fewer timeouts, respectively.

4.7.5 Scalability

Finally, we evaluate the scalability of Difflog-based synthesis, which is affected by two factors: the number of templates and the size of training data. Our general observation is that increasing either of these does not significantly increase the effective running time (i.e., the best of 32 parallel runs).

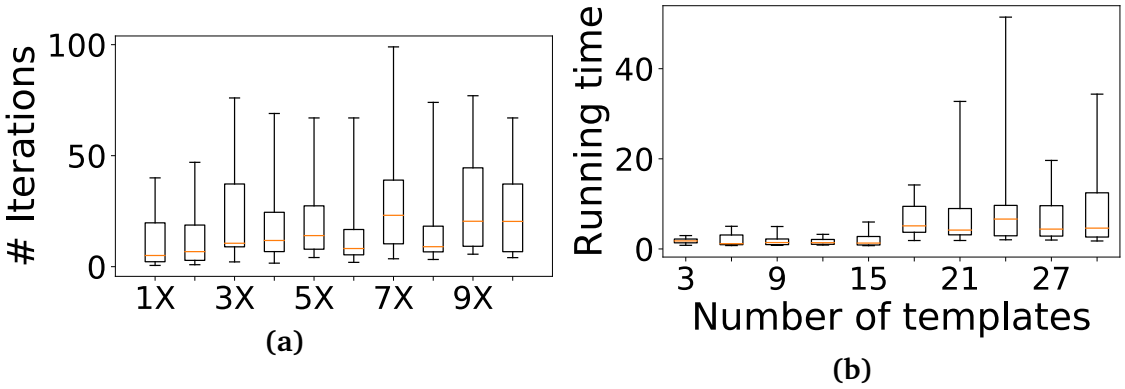
Figure 4.6 shows how running time increases with the number of templates.[10] As shown in Figure 4.6a, the running time distribution for `2-call-site` tends to have larger variance when the number of templates increases, but the best running time (out of 32 i.i.d samples) only increases modestly. The running time distribution for `downcast`, shown in Figure 4.6b, has a similar trend except that smaller num-

[10]We ensure that all candidate rules in a set are also present in subsequent larger sets.



(a) `2-call-site` **(b)** `downcast`

Figure 4.6: Running time distributions (in minutes) for `downcast` and `2-call-site` with different number of templates.



(a) **(b)**

Figure 4.7: Performance of Difflog on `andersen` with different sizes of data: (a) the distribution of number of iterations, (b) the distribution of running time (in seconds).

ber of templates does not always lead to smaller variance or faster running time. For instance, the distribution in the setting with 180 templates has larger variance and median than distributions in the subsequent settings with larger number of templates. This indicates that the actual combination of templates also matters. In general, approximately half the benchmarks follow a trend similar to Figure 4.6a, with monotonically increasing variance in running times, while the remaining benchmarks are similar to Figure 4.6b.

The size of training data is another important factor affecting the performance of Difflog. Figure 4.7a shows the distribution of the number of iterations for `andersen` with different sizes of training data. According to the results, the size of training data does not necessarily affect the number of iterations of Difflog. Meanwhile,

Figure 4.7b shows that the end-to-end running time increases with more training data. This is mainly because more training data imposes more cost on the Difflog evaluator. However, the statistics show that the running time increases linearly with the size of data.

4.8 Related Work

Template-guided synthesis. Templates are commonly used to guide the search in program synthesis [173, 49, 169, 172]. At a high-level, meta-rules can also be seen as program sketches [172], where the holes are the relation symbols. One advantage of meta-rules over sketches is that Datalog programs in various domains share the exact same meta-rules. For instance, meta-rules used for graph manipulation are also used in program analyses. Another advantage is that with predicate invention simple meta-rules can be *composed* together to express complex rules.

Weighted logical inference. The idea of extending logical inference with weights has been studied by the community in statistical relational learning. [160] proposes *quantitative logic programming* to measure the uncertainty of expert systems by associating logical rules with uncertainty scores. Markov Logic Networks [144, 99] view a first order formula as a template for generating a Markov random field, where the weight attached to the formula specifies the likelihood of its grounded clauses. ProbLog [57] extends logic programing languages with probabilistic rules and reduces the inference problem to weighted model counting. DeepProbLog [118] further extends ProbLog with neural predicates (e.g., input data which can be images). In another direction, aProbLog [96, 97] generalizes ProbLog by associating logical rules with elements from a semiring, instead of just probability values. These frameworks could conceivably serve as the underlying inference engine of our framework but we use the Viterbi semiring because: (*a*) inference in these frame-

works is #P-complete and only requires polynomial time in the Viterbi semiring; and (*b*) automatic differentiation is either inefficient or simply not available.

Structure learning for probabilistic logics. Weight learning has also been used as a means to structure learning [127, 181, 63]; however, our work has two significant differences: First, the values we assign to tuples do not have natural interpretations as probabilities, so that exact inference can be performed just as efficiently as solving Datalog programs. Furthermore, while the search trajectory itself proceeds through smoothed programs with non-zero loss, our termination criterion ensures that the final result is still a classical Datalog program which is consistent with the provided examples.

Inductive logic programming (ILP). The Datalog synthesis problem can also be seen as an instance of the classic ILP problem. [43] show that learning a single rule that is consistent with labelled examples is NP-hard: this is similar to our motivating result in Theorem 4.2.2, where we demonstrate NP-hardness even if candidate rules are explicitly specified. Metagol [128] supports higher-order dyadic Datalog synthesis but the synthesized program can only consist of binary relations. Metagol is built on top of Prolog which makes the system very expressive but also introduces difficult issues with non-terminating programs. Recent works such as NeuralLP [188] and ∂ILP [64] cast logic program synthesis as a differentiable end-to-end learning problem and model relation joins as a form of matrix multiplication, which also limits them to binary relations. NTP [145] constructs a neural network as a learnable proof (or derivation) for each output tuple up to a predefined depth (e.g. ≤ 2) with a few (e.g. ≤ 4) templates, where the neural network could be exponentially large when either the depth or the number of templates grows. The predefined depth and a small number of templates could significantly limit the class of learned programs. Our work seeks to synthesize Datalog programs consisting of relations of arbitrary arity and support rich features like recursion and predicate

invention.

MCMC methods for program synthesis. Markov chain Monte-Carlo (MCMC) methods have also been used for program synthesis. For example, in STOKE, [152] apply the Metropolis-Hastings algorithm to synthesize efficient loop free programs. Similarly, [112] show that program transformations can be efficiently learned from demonstrations by MCMC inference.

4.9 Conclusion

We have presented a technique to synthesize Datalog programs using numerical optimization. The central idea is to formulate the problem as an instance of rule selection, and then relax classical Datalog to a refinement named Difflog. In a comprehensive set of experiments, we show that by learning a Difflog program and then recovering a classical Datalog program, we can achieve significant speedups over the state-of-the-art Datalog synthesis systems. In future, we plan to extend the approach to other synthesis problems such as SyGuS and to applications in differentiable programming.

Technical and experimental results presented in this chapter are from the following published papers:

- ✎ Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. Synthesizing Datalog Programs using Numerical Relaxation. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China*, pages 6117–6124.

- ✎ Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paris Koutris, Mayur Naik. Syntax-Guided Synthesis of Datalog Programs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 515–527, ACM 2018.

CHAPTER 5

DEEP REINFORCEMENT LEARNING FOR PROGRAM VERIFICATION

5.1 Introduction

The growing ubiquity and complexity of software has led to a dramatic increase in software bugs and security vulnerabilities that pose enormous costs and risks. Program verification technology enables programmers to prove the absence of such problems at compile-time before deploying their program. One of the main activities underlying this technology involves inferring a *loop invariant*—a logical formula that constitutes an abstract specification of a loop—for each loop in the program. Obtaining loop invariants enables a broad and deep range of correctness and security properties to be proven automatically by a variety of program verification tools spanning type checkers, static analyzers, and theorem provers. Notable examples include Microsoft Code Contracts for .NET programs [65] and the Verified Software Toolchain spanning C source code to machine language [20].

Many different approaches have been proposed in the literature to infer loop invariants. The problem is undecidable, however, and even practical instances are challenging, which greatly limits the benefits of program verification technology. Existing approaches suffer from key drawbacks: they are purely search-based, or they use hand-crafted features, or they are based on supervised learning. The performance of search-based approaches is greatly hindered by their inability to learn from past mistakes. Hand-crafted features limit the space of possible invariants, e.g., Garg et al. [68] is limited to features of the form $x \pm y \leq c$ where c is a constant, and thus cannot handle invariants that involve $x+y \leq z$ for program variables

x, y, z. Finally, obtaining ground truth solutions needed by supervised learning is hindered by the undecidability of the loop invariant generation problem.

We propose Code2Inv, an end-to-end learning-based approach to inferring loop invariants. Code2Inv has the ability to automatically learn rich latent representations of desirable invariants, and can avoid repeating similar mistakes. Furthermore, it leverages reinforcement learning to discover invariants by partial feedback from trial-and-error, without needing ground truth solutions for training.

The design of Code2Inv is inspired by the reasoning exercised by human experts. Given a program, a human expert first maps the program to a well-organized structural representation, and then composes the loop invariant step by step. Based on such reasoning, different parts of the representation get highlighted at each step. To mimic this procedure, we utilize a graph neural network model (GNN) to construct the structural external memory representation of the program. The multi-step decision making is implemented by an autoregressive model, which queries the external memory using an attention mechanism. The decision at each step is a syntax- and semantics-guided decoder which generates subparts of the loop invariant.

Code2Inv employs a reinforcement learning approach since it is computationally intensive to obtain ground truth solutions. Although reinforcement learning algorithms have shown remarkable success in domains like combinatorial optimization [29, 54] (see Section 5.6 for more discussion on related work), our setting differs in two crucial ways: first, it has a non-continuous objective function (i.e., a proposed loop invariant is correct or not); and second, the positive reward is extremely sparse and given only after the correct loop invariant is proposed, by an automated theorem prover [56]. We therefore model the policy learning as a multi-step decision making process: it provides a fine-grained reward at each step of building the loop invariant, followed by continuous feedback in the last step based on counterexamples collected by the agent itself during trial-and-error learning.

We evaluate Code2Inv on a suite of 133 benchmark problems from recent works [60,

133, 68] and the 2017 SyGuS program synthesis competition [13]. We also compare it to three state-of-the-art systems: a stochastic search-based system C2I [161], a heuristic search-based system LoopInvGen [133], and and a decision tree learning-based system ICE-DT [68]. Code2Inv solves 106 problems, versus 73 by C2I, 77 by LoopInvGen, and 100 by ICE-DT. Moreover, Code2Inv exhibits better learning, making orders-of-magnitude fewer calls to the theorem prover than these systems.

5.2 Problem Formulation

We formally define the loop invariant inference and learning problems by briefly introducing Hoare logic [88], which comprises a set of axioms and inference rules for proving program correctness assertions. Let P and Q denote predicates over program variables and let S denote a program. We say that *Hoare triple* $\{P\}\ S\ \{Q\}$ is valid if whenever S begins executing in a state that satisfies P and finishes executing, then the resulting state satisfies Q. We call P and Q the *pre-condition* and *post-condition* respectively of S. Hoare rules allow to derive such triples inductively over the structure of S. The rule most relevant for our purpose is that for loops:

$$\frac{P \Rightarrow I \quad (\textit{pre}) \qquad \{I \wedge B\}\ S\ \{I\} \quad (\textit{inv}) \qquad (I \wedge \neg B) \Rightarrow Q \quad (\textit{post})}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{Q\}}$$

Predicate I is called a *loop invariant*, an assertion that holds before and after each iteration, as shown in the premise of the rule. We can now formally state the loop invariant inference problem:

Problem 5.2.1 (Loop Invariant Inference)**.** Given a pre-condition P, a post-condition Q and a program S containing a single loop, can we find a predicate I such that $\{P\}\ S\ \{Q\}$ is valid?

Given a candidate loop invariant, it is straightforward for an automated theorem prover such as Z3 [56] to check whether the three conditions denoted *pre*, *inv*, and

post in the premise of the above rule hold, and thereby prove the property asserted in the conclusion of the rule. If any of the three conditions fails to hold, the theorem prover returns a concrete counterexample witnessing the failure.

The loop invariant inference problem is undecidable. Moreover, even seemingly simple instances are challenging, as we illustrate next using the program in Figure 5.1(a). The goal is to prove that assertion $(y > 0)$ holds at the end of the program, for every input value of integer variable y. In this case, the pre-condition P is `true` since the input value of y is unconstrained, and the post-condition Q is $(y > 0)$, the assertion to be proven. Using predicate $(x < 0 \ \vee \ y > 0)$ as the loop invariant I suffices to prove the assertion, as shown in Figure 5.1(b). Notation $\phi[e/x]$ denotes the predicate ϕ with each occurrence of variable x replaced by expression e. This loop invariant is non-trivial to infer. The reasoning is simple in the case when the input value of y is non-negative, but far more subtle in the case when it is negative: regardless of how negative it is at the beginning, the loop will iterate at least as many times as to make it positive, thereby ensuring the desired assertion upon finishing. Indeed, a state-of-the-art loop invariant generator LoopInvGen [133] crashes on this problem instance after making 1,119 calls to Z3, whereas Code2Inv successfully generates it after only 26 such calls.

The central role played by loop invariants in program verification has led to a large body of work to automatically infer them. Many previous approaches are based on exhaustive bounded search using domain-specific heuristics and are thereby limited in applicability and scalability [44, 148, 80, 163, 162, 8, 60, 67]. A different strategy is followed by data-driven approaches proposed in recent years [161, 68, 133]. These methods speculatively guess likely invariants from program executions and check their validity. In [68], decision trees are used to learn loop invariants with simple linear features, e.g. $a * x + b * y < c$, where $a, b \in \{-1, 0, 1\}, c \in Z$. In [133], these features are generalized by systematic enumeration. In [161], stochastic search is performed over a set of constraint templates. While such features or tem-

$x := -50;$
while $(x < 0)$ {
  $x := x + y;$
  $y := y + 1$
}
assert$(y > 0)$

(a) An example program.

(b) A desirable loop invariant I is a predicate over x, y such that:

$$\forall x, y : \begin{cases} \texttt{true} \Rightarrow I[-50/x] & (\textit{pre}) \\ I \wedge x < 0 \Rightarrow I[(y+1)/y, (x+y)/x] & (\textit{inv}) \\ I \wedge x \geq 0 \Rightarrow y > 0 & (\textit{post}) \end{cases}$$

(c) The desired loop invariant is $(x < 0 \vee y > 0)$.

Figure 5.1: A program with a correctness assertion and a loop invariant that suffices to prove it.

plates perform well in specific domains, however, they may fail to adapt to new domains. Moreover, even in the same domain, they do not benefit from past experiences: successfully inferring the loop invariant for one program does not speed up the process for other similar ones. We hereby formulate the second problem we aim to address:

Problem 5.2.2 (Loop Invariant Learning)**.** Given a set of programs $\{S_i\} \sim \mathcal{P}$ that are sampled from some unknown distribution $\mathcal{P}$, can we learn from them and generalize the strategy we learned to other programs $\{\tilde{S}_i\}$ that are from the same distribution?

5.3 End-to-End Reasoning Framework

5.3.1 Reasoning Process of a Human Expert

We start out by illustrating how a human expert might typically accomplish the task of inferring a loop invariant. Consider the example in Figure 5.2 chosen from our benchmarks.

An expert usually starts by reading the assertion (line 15), which contains variables `x` and `y`, then determines the locations where these two variables are initialized, and then focuses on the locations where they are updated in the loop. Instead of reasoning about the entire assertion at once, an expert is likely to focus on updates

to one variable at a time. This reasoning yields the observation that x is initialized to zero (line 2) and may get incremented in each iteration (line 5,9). Thus, the sub goal "x $<$ 4" may not always hold, given that the loop iterates non-deterministically. This in turn forces the other part "y $>$ 2" to be true when "x $>=$ 4". The only way x can equal or exceed 4 is to execute the first if branch 4 times (line 4-6), during which y is set to 100. Now, a natural guess for the loop invariant is "x $<$ 4 || y $>=$ 100". The reason for guessing "y $>=$ 100" instead of "y $<=$ 100" is because part of the proof goal is "y $>$ 2". However, this guess will be rejected by the theorem prover. This is because y might be decreased by an arbitrary number of times in the third if-branch (line 12), which happens when x is less than zero; to avoid that situation, "x $>= 0$" should also be part of the loop invariant. Finally, we have the correct loop invariant: "(x $>=$ 0) && (x $<$ 4 || y $>=$ 100)", which suffices to prove the assertion.

```
1 int main() {
2   int x = 0, y = 0;
3   while (*) {
4      if (*) {
5         x++;
6         y = 100;
7      } else if (*) {
8         if (x >= 4) {
9            x++;
10           y++;
11        }
12        if (x < 0) y--;
13     }
14  }
15  assert( x < 4 || y > 2);
16}
```

Figure 5.2: An example from our benchmarks. "*" denotes non-deterministic choice.

We observe that the entire reasoning process consists of three key components: 1) organize the program in a hierarchical-structured way rather than a sequence of tokens; 2) compose the loop invariant step by step; and 3) focus on a different part of the program at each step, depending on the inference logic, e.g., abduction and induction.

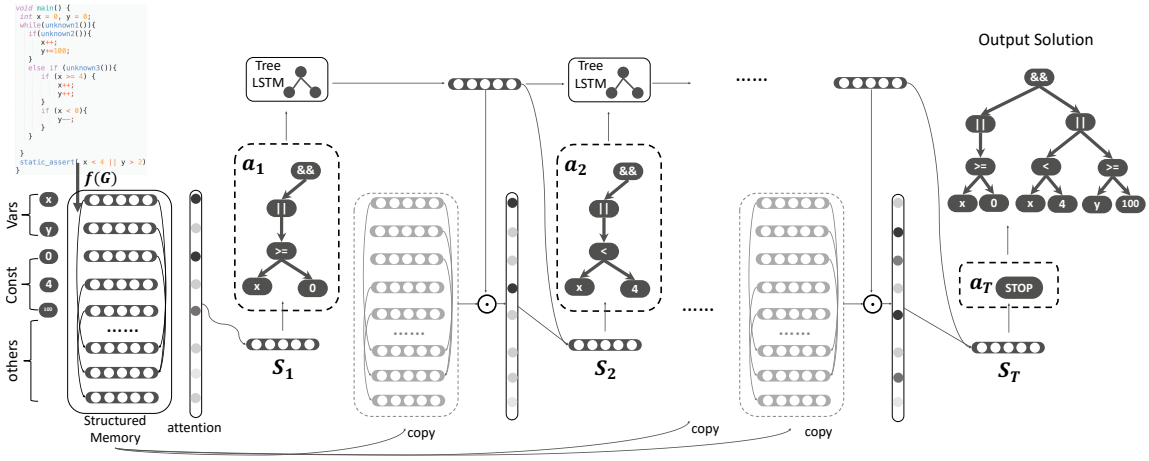


Figure 5.3: Overall framework of neuralizing loop invariant inference.

5.3.2 Reasoning with Neural Networks

We propose to use a neural network to mimic the reasoning used by human experts as described above. The key idea is to replace the above three components with corresponding differentiable modules:

- a structured external memory representation which encodes the program;
- a multi-step autoregressive model for incremental loop invariant construction;
- an attention component that mimics the varying focus in each step.

As shown in Figure 5.3, these modules together build up the network that constructs loop invariants from programs, while being jointly trained with reinforcement learning described in Section 5.4. At each step, the neural network generates a predicate. Then, given the current generated partial tree, a Tree-LSTM module summarizes what have been generated so far, and the summarization is used to read the memory using attention. Lastly, the summarization together with the read memory is fed into next time step. We next elaborate upon each of these three components.

Structured external memory

The loop invariant is built within the given context of a program. Thus it is natural to encode the program as an external memory module. However, in contrast to traditional memory networks [174, 123], where the memory slots are organized as a linear array, the information contained in a program has rich structure. A chain LSTM over program tokens can in principle capture such information but it is challenging for neural networks to understand with limited data. Inspired by Allamanis et al. [11], we instead use a graph-structured memory representation. Such a representation allows to capture rich semantic knowledge about the program such as its control-flow and data-flow.

More concretely, we first convert a given program into static single assignment (SSA) form [50], and construct a control flow graph, each of whose nodes represents a single program statement. We then transform each node into an abstract syntax tree (AST) representing the corresponding statement. Thus a program can be represented by a graph $G = (V, E)$, where V contains terminals and nonterminals of the ASTs, and $E = \{(e_x^{(i)}, e_y^{(i)}, e_t^{(i)})\}_{i=1}^{|E|}$ is the set of edges. The directed edge $(e_x^{(i)}, e_y^{(i)}, e_t^{(i)})$ starts from node $e_x^{(i)}$ to $e_y^{(i)}$, with $e_t^{(i)} \in \{1, 2, \ldots, K\}$ representing edge type. In our construction, the program graph contains 3 different edge types (and 6 after adding reversed edges).

To convert the graph into a vector representation, we follow the general message passing operator introduced in graph neural network (GNN) [150] and its variants [61, 53, 11]. Specifically, the graph network will associate each node $v \in V$ with an embedding vector $\mu_v \in \mathbb{R}^d$. The embedding is updated iteratively using the general neighborhood embedding as follows:

$$\mu_v^{(l+1)} = h(\{\mu_u^{(l)}\}_{u \in \mathcal{N}^k(v), k \in \{1,2,\ldots,K\}}) \tag{5.1}$$

Here $h(\cdot)$ is a nonlinear function that aggregates the neighborhood information to

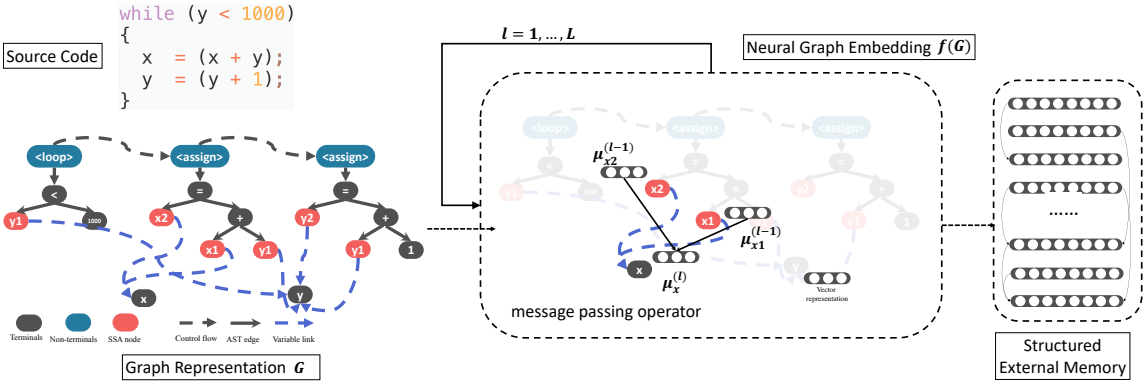


Figure 5.4: Diagram for source code graph as external structured memory. We convert a given program into a graph G, where nodes correspond to syntax elements, and edges indicate the control flow, syntax tree structure, or variable linking. We use embedding neural network to get structured memory $f(G)$.

update the embedding. $\mathcal{N}^k(v)$ is the set of neighbor nodes connected to v with edge type k, *i.e.*, $\mathcal{N}^k(v) = \{u|(u, v, k) \in E\}$. This process is repeated for L steps and the node embedding μ_v is set to $\mu_v^{(L)}, \forall v \in V$. Our parameterization takes the edge types into account. The specific parameterization used is shown below:

$$\mu_v^{(l+1),k} = \sigma(\textstyle\sum_{u \in \mathcal{N}^k(v)} \mathbf{W}_2 \mu_u^{(l)}), \forall k \in \{1, 2, \ldots, K\} \tag{5.2}$$

$$\mu_v^{(l+1)} = \sigma(\mathbf{W}_3[\mu_v^{(l+1),1}, \mu_v^{(l+1),2}, \ldots, \mu_v^{(l+1),K}]) \tag{5.3}$$

with the boundary case $\mu_v^{(0)} = \mathbf{W}_1 \mathbf{x}_v$. Here $\mathbf{x}_v$ represents the syntax information of node v, such as token or constant value in the program. Matrices $\mathbf{W}_{1,2,3}$ are learnable model parameters, and σ is some nonlinear activation function. Figure 5.4 shows the construction of graph structured memory using the iterative message passing operator in Eq (5.1). $f(G) = \{\mu_v\}_{v \in V}$ denotes the structured memory.

Multi-step decision making process

A loop invariant itself is a mini-program that contains expressions and logical operations. Without loss of generality, we define the loop invariant to be a tree $\mathcal{T}$, in

a form with conjunctions of disjunctions:

$$\mathcal{T} = (\mathcal{T}_1 \,||\, \mathcal{T}_2 \ldots) \,\&\&\, (\mathcal{T}_{t+1} \,||\, \mathcal{T}_{t+2} \ldots) \,\&\&\, \ldots\, (\ldots \mathcal{T}_{T-1} || \,\mathcal{T}_T) \tag{5.4}$$

Each subtree $\mathcal{T}_t$ is a simple logic expression (*i.e.*, `x < y * 2 + 10 - z`). Given this representation form, it is natural to use Markov decision process (MDP) to model this problem, where the corresponding T-step finite horizon MDP is defined as $\mathcal{M}^G = (s_1, a_1, r_1, s_2, a_2, \ldots, s_T)$. Here s_t, a_t, r_t represent the state, action and reward at time step $t = 1, \ldots, T-1$, respectively. Here we describe the state and action used in the inference model, and describe the design of reward and termination in Section 5.4.

action: As defined in Eq (5.4), a loop invariant tree $\mathcal{T}$ consists of multiple subtrees $\{\mathcal{T}_t\}$. Thus we model the action at time step t as $a_t = (op_t, \mathcal{T}_t)$, where op_t can either be $||$ or $\&\&$. That is to say, at each time step, the agent first decides whether to attach the subexpression $\mathcal{T}_t$ to an existing disjunction, or create a new disjunction and add it to the list of conjunctions. We use $\mathcal{T}^{(<t)}$ to denote the partial tree generated by time t so far. So the policy $\pi(\mathcal{T}|G)$ is decomposed into:

$$\pi(\mathcal{T}|G) = \prod_{t=1}^{T} \pi(a_t|\mathcal{T}^{(<t)}, G) = \prod_{t=1}^{T} \pi(op_t, \mathcal{T}_t|\mathcal{T}^{(<t)}, G) \tag{5.5}$$

where $\mathcal{T}^{(<1)}$ is empty at the first step. The generation process of subtree $\mathcal{T}_t$ is also an autoregressive model implemented by LSTM. However, generating a valid program is nontrivial, since strong syntax and semantics constraints should be enforced. Recent advances in neural program synthesis [134, 104] utilize formal language information to help the generation process. Here we use the Syntax-Directed decoder proposed in [55] to guarantee both the syntax and semantics validity. Specifically,

- Syntax constraints: The AST generation follows the grammar of loop invariants described in Eq 5.4. Operators such as `+`, `-`, `*` are non-terminal nodes in the

AST while operands such as constants or variables are leaf nodes.

- Semantic constraints: We regulate the generated loop invariant to be meaningful. For example, a valid loop invariant must contains all the variables that appear in the given assertion. Otherwise, the missing variables can take arbitrary values, causing the assertion to be violated. In contrast to offline checking which discards invalid programs after generation, such online regulation restricts the output space of the program generative model, which in turn makes learning efficient.

state: At time step $t = 1$, the state is simply the weighted average of structured memory $f(G)$. At each later time step $t > 1$, the action a_t should be conditioned on graph memory, as well as the partial tree generated so far. Thus $s_t = (G, \mathcal{T}^{(<t)})$.

Memory query with attention

A program is encoded as an external memory to an agent, i.e. a deep learning model used to construct candidate loop invariants. A memory query from the agent intuitively means looking into certain part of the program, which resembles human focus or attention on the source code. Note that the attention evolves along with the invariant construction. To mimic such process, we model the attention as a mapping from partially generated invariant to some region of the external memory. More concretely, we use TreeLSTM [177] to represent this mapping. At time step t, we compute an embedding $\mathbf{v}_{\mathcal{T}^{(<t)}}$ of the partially generated invariant $\mathcal{T}^{(<t)}$

$$\mathbf{v}_{\mathcal{T}^{(<t)}} = \text{TreeLSTM}(\mathcal{T}^{(<t)}) \tag{5.6}$$

which can be viewed a "context" that is subsequently used to determine the attention

$$\alpha_v = \frac{\exp \mu_v^\top \mathbf{v}_{\mathcal{T}^{(<t)}}}{\sum_{v \in V} \exp \mu_v^\top \mathbf{v}_{\mathcal{T}^{(<t)}}} \tag{5.7}$$

With attention α_v, the agent eventually read out $\sum_{v \in \mathcal{V}} \alpha_v \mu_v$.

5.4 Reinforcement Learning

The undecidability of the loop invariant generation problem hinders the ability to obtain ground truth solutions as supervisions for training. Inspired by recent advances in combinatorial optimization [29, 54], where the agent learns a good policy by trial-and-error, we employ reinforcement learning to learn to propose loop invariants. Ideally, we seek to learn a policy $\pi(\mathcal{T}|G)$ that proposes a correct loop invariant $\mathcal{T}$ for a program graph G. However, directly solving such a model is practically not feasible, since:

- In contrast to problems tackled by existing work, where the objective function is relatively continuous (*e.g.*, tour length of traveling salesman problem), the proposed loop invariant only has binary objective (*i.e.*, correct or not). This makes the loss surface of the objective function highly non-smooth.
- Finding the loop invariant is a bandit problem where the binary reward is given only after the invariant is proposed. Also, in contrast to two player games [167] where a default policy (e.g., random rollout) can be used to estimate the reward, it is a single player game with an extremely sparse reward.

To tackle the above two challenges, the multi-step decision making model proposed in Section 5.3.2 is used, where a fine-grained reward is also designed for each step. In the last step, a continuous feedback is provided based on the counterexamples collected by the agent itself.

5.4.1 Reward Design

Section 5.3.2 defines the state and action representation used for inference. We next describe our reward design which is important to properly train a reinforcement learning agent.

reward: In each intermediate step $t \in 1, \ldots, T-1$, an intermediate reward r_t is given to regulate the generation process. For example, a subexpression should be non-trivial, and it should not contradict $\mathcal{T}^{(<t)}$. In the last step, the generated

loop invariant $\mathcal{T}$ is given to a theorem prover, which returns success or failure. In the latter case, the theorem prover also tells which step (*pre*, *inv*, *post*) failed, and provides a counterexample. The failure step can be viewed as a "milestone" of the verification process, providing a coarse granularity feedback. To achieve continuous (i.e. fine granularity) reward within each step, we exploit the counterexamples collected so far. For instance, the ratio of passed examples is a good indicator of the learning progress.

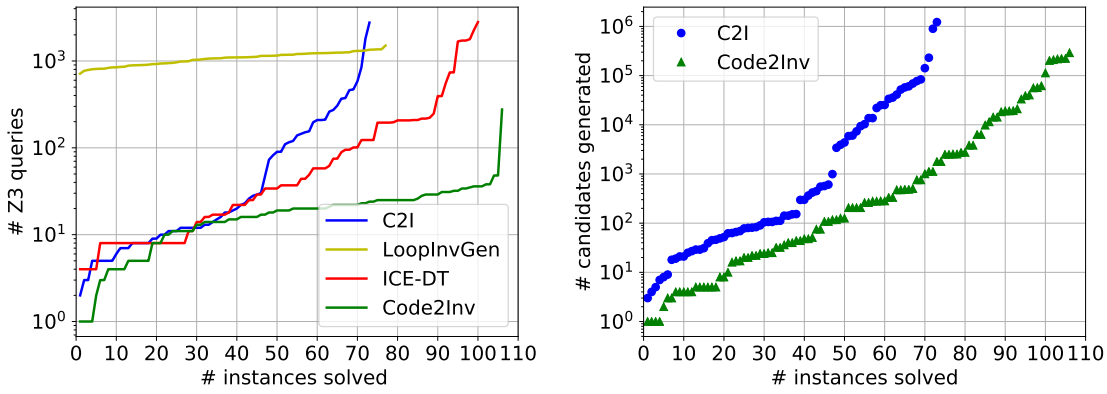
termination: There are several conditions that may trigger the termination of tree generation: (1) the agent executes the "stop" action, as illustrated in Figure 5.3; (2) the generated tree has the maximum number of branches allowed; or (3) the agent generates an invalid action.

5.4.2 Training

We use the advantage actor critic (A2C) algorithm [101] to train the above reinforcement learning policy. Specifically, let $\theta = \{\mathbf{W}_i\}$ be the parameters in graph memory representation $f(\cdot;\theta)$, and ϕ be the parameter used in $\pi(a_t|\mathcal{T}^{(<t)}, G; \phi)$, our objective is to maximize the expected policy reward:

$$\max_{\theta,\phi} \mathbb{E}_{\pi(op_t,\mathcal{T}_t|\mathcal{T}^{(<t)},G;\phi)}(\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} - b(\mathcal{T}^{(<t)}, G; \psi)) \tag{5.8}$$

To reduce the variance of policy gradient, we use the baseline function $b(\mathcal{T}^{(<t)}, G; \psi)$ modeled as a two-layer MLP parameterized by ψ, which is trained to minimize $\mathbb{E}_{\pi,t}\| \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} - b(\mathcal{T}^{(<t)}, G; \psi)\|$. Given that the MDP has a finite horizon, we set the discounting factor γ to 1 to encourage the long-term gain.



(a) verification cost by each solver **(b) sample complexity**

Figure 5.5: Comparison of Code2Inv with state-of-the-art solvers on benchmark dataset.

5.5 Experiments

We evaluate Code2Inv on a suite of 133 benchmark programs from recent works [60, 133, 68] and the 2017 SyGuS competition [176].[11] Each program consists of three parts: a number of assumption or assignment statements, one loop which contains nested if-else statements with arithmetic operations, and one assertion statement.

We first evaluate Code2Inv as an out-of-the-box solver, i.e., without any training or fine-tuning with respect to the dataset. We then conduct an ablation study to justify various design choices. Finally, we evaluate the impact of training Code2Inv on a similar dataset.

5.5.1 Learning Loop Invariants from Scratch

In this section, we study the capability of Code2Inv with no training, that is, using it as an out-of-the-box solver. We compare Code2Inv with three state-of-the-art solvers: C2I [161], which is based on stochastic search; LoopInvGen [133], which searches a conjunctive normal form over predicates synthesized by an underlying engine, Escher [8]; and ICE-DT [68], which learns a decision tree over manually designed features (e.g. predicate templates). The last two solvers are the winners of

[11] Our code and data are publicly available from https://github.com/PL-ML/code2inv

the invariant synthesis track of the SyGuS 2017 and 2016 competitions, respectively.

A uniform metric is needed to compare the different solvers since they can leverage diverse performance optimizations. For instance, Code2Inv can take advantage of GPUs and TPUs, and C2I can benefit from massive parallelization. Instead of comparing absolute running times, we observe that all four solvers are based on the Z3 theorem prover [56] and rely on the counterexamples from Z3 to adjust their search strategy. Therefore, we compare the number of queries to Z3, which is usually the performance bottleneck for verification tasks. We run all solvers on a single 2.4 GHz AMD CPU core up to 12 hours and using up to 4 GB memory for each program.

Figure 5.5a shows the number of instances solved by each solver and the corresponding number of queries to Z3. Code2Inv solves the largest number of instances, which is **106**. In contrast, ICE-DT, LoopInvGen and C2I solve **100**, **77** and **74** instances, respectively. ICE-DT heavily relies on predicate templates designed by human experts, which are insufficient for 19 instances that are successfully solved by Code2Inv. Furthermore, to solve the same amount of instances, Code2Inv costs orders of magnitude fewer queries to Z3 compared to the other solvers.

We also run Code2Inv using the time limit of one hour from the 2017 SyGuS competition. Code2Inv solves **92** instances within this time limit with the same hardware configuration. While it cannot outperform existing state-of-the-art solvers based on absolute running times, however, we believe its speed can be greatly improved by (1) pre-training on similar programs, which we show in Section 5.5.3; and (2) an optimized implementation that takes advantage of GPUs or TPUs.

Code2Inv is most related to C2I since both use accumulated counterexamples to adjust the sample distribution of loop invariants. The key difference is that C2I uses MCMC sampling whereas Code2Inv learns using RL. Figure 5.5b shows the sample complexity, i.e., number of candidates generated before successfully finding the desired loop invariant. We observe that Code2Inv needs orders of magnitude less

samples which suggests that it is more efficient in learning from failures.

5.5.2 Ablation Study

We next study the effectiveness of two key components in our framework via ablation experiments: counterexamples and attention mechanism. We use the same dataset as in Section 5.5.1. Table 5.1 shows our ablation study results. We see that besides providing a continuous reward, the use of counterexamples (CE) significantly reduces the verification cost, i.e., number of Z3 queries. On the other hand, the attention mechanism helps to reduce the training cost, i.e., number of parameter updates. Also, it helps to reduce the verification cost modestly. Code2Inv achieves the best performance with both components enabled—the configuration used in other parts of our evaluation.

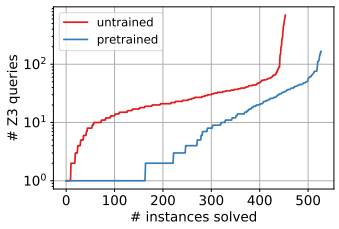
Additionally, to test the effectiveness of neural graph embedding, we study a simpler encoding, that is, viewing a program as a sequence of tokens and encoding the sequence using an LSTM. The performance of this setup is shown in the last row of Table 5.1. With a simple LSTM embedding, Code2Inv solves 13 fewer instances and, moreover, requires significantly more parameter updates.

Table 5.1: Ablation study for different configurations of Code2Inv.

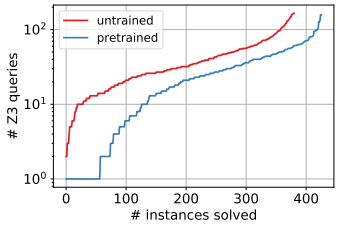
configuration	#solved instances	max #Z3 queries	max #updates
without CE, without attention	91	415K	441K
without CE, with attention	94	147K	162K
with CE, without attention	95	392	337K
with CE, with attention	106	276	290K
LSTM embedding + CE + attention	93	32	661K

5.5.3 Boosting the Search with Pre-training

We next address the question: given an agent that is pre-trained on programs $P_{train} = \{p_i\} \sim \mathcal{P}$, can the agent solve new programs $P_{test} = \{\tilde{p}_i\} \sim \mathcal{P}$ faster than solving from scratch? We prepare the training and testing data as follows. We



(a) with 1 confounding variable



(b) with 5 confounding variables

```
int main()
{
  int a = 0, b = 0, c = 0;

  while(*) {
      a += 1;
      b += 1;
      c -= 1;
  }

  if(a != b) {
    assert (b == -1);
  }
  else{
  }
}
```

(c) attention for invariant $a == b$

```
int main()
{
  int n, k, c = 0;
  assume (n > 0);

  while(*) {
    if(c < n) {
      c = c + 1;
      k = 2;
    }
    if(c == n) {
      c = 1;
      k = 5;
    }
  }

  if(c == n) {
    assert (n > -1);
  }
}
```

(d) attention for the first part of invariant: $c >= -1$ **&&** $n >= 1$

Figure 5.6: (a) and (b) are verification costs of pre-trained model and untrained model; (c) and (d) are attention highlights for two example programs.

take the programs solved by Code2Inv as the initial set and augment it by creating 100 variations for each of them by introducing confounding variables and statements in such a way that any valid loop invariant for the original program is still valid. Finally, 90% of them serves as P_{train}, and the rest are used for P_{test}.

After pre-training the agent on P_{train} for 50 epochs, we save the model and then reuse it for "fine tuning" (or active search [29]), *i.e.*, the agent continues the trial-and-error reinforcement learning, on P_{test}. Figure 5.6a and Figure 5.6b compare the verification costs between the pre-trained model and untrained model on datasets augmented with 1 and 5 confounding variables, respectively. We observe that, on one hand, the pre-trained model has a clear advantage over the untrained model on either dataset; but on the other hand, this gap reduces when more confounding variables are introduced. This result suggests an interesting future research direction: how to design a learning agent to effectively figure out loop invariant related variables from a potentially large number of confounding variables.

5.5.4 Attention Visualization

Figure 5.6c and 5.6d show the attention highlights for two example programs. The original highlights are provided on the program graph representation described in Section 5.3.2. We manually converted the graphs back to source code for clarity. Figure 5.6c shows an interesting example for which Code2Inv learns a strategy of showing the assertion is actually not reachable, and thus holds trivially. Figure 5.6d shows another interesting example for which Code2Inv performs a form of abductive reasoning.

5.5.5 Discussion of limitations

We conclude our study with a discussion of limitations. For most of the instances that Code2Inv fails to solve, we observe that the loop invariant can be expressed in a compact disjunctive normal form (DNF) representation, which is more suited for the decision tree learning approach with hand-crafted features. However, Code2Inv is designed to produce loop invariants in the conjunctive normal form (CNF). The reduction of loop invariants from DNF to CNF could incur an exponential blowup in size. An interesting future research direction concerns designing a learning agent that can flexibly switch between these two forms.

5.6 Related Work

We survey work in program synthesis, program learning, learning loop invariants, and learning combinatorial optimizations.

Program synthesis. Automatically synthesizing a program from its specification has been a key challenge problem since Manna and Waldinger's work [119]. In this context, syntax-guided synthesis (SyGuS) [13] was proposed as a common format to express these problems. Besides several implementations of SyGuS solvers [79, 16, 151, 13], a number of probabilistic techniques have been proposed to model

syntactic aspects of programs and to accelerate synthesis[31, 115, 131]. While logical program synthesis approaches guarantee semantic correctness, they are chiefly limited by their scalability and requirement of rigorous specifications.

Program learning. There have been several attempts to learn general programs using neural networks. One large class of projects includes those attempting to use neural networks to accelerate the discovery of *conventional programs* [26, 129, 59, 134]. Most existing works only consider specifications which are in the form input-output examples, where weak supervision [110, 40, 38] or more fine grained trace information is provided to help training. In our setting, there is no supervision for the ground truth loop invariant, and the agent needs to be able to compose a loop invariant purely from trial-and-error. Drawing inspiration from both programming languages and embedding methods, we build up an efficient learning agent that can perform end-to-end reasoning, in a way that mimics human experts.

Learning program loop invariants. Our work is closely related to recent work on learning loop invariants from either labeled ground truth [36] or active interactions with human experts [34]. Brockschmidt et al. [36] learn shape invariants for data structures (e.g. linked lists or trees). Their approach first extracts features using n-gram and reachability statistics over the program's heap graph and then applies supervised learning to train a neural network to map features to shape invariants. In contrast, we are concerned with general loop invariant generation, and our approach employs graph embedding directly on the program's AST and learns a generation policy without using ground truth as supervision. Bounov et al. [34] propose inferring loop invariants through gamification and crowdsourcing, which relieves the need for expertise in software verification, but still requires significant human effort. In contrast, an automated theorem prover suffices for our approach.

Learning combinatorial optimizations. Our work is also related to recent advances in combinatorial optimization using machine learning [95, 29, 54, 154].

However, as elaborated in Section 5.4, the problem we study is significantly more difficult, in the sense that the objective function is non-smooth (binary objective), and the positive reward is extremely sparse due to the exponentially growing size of the search space with respect to program size.

5.7 Discussion

We studied the problem of learning loop invariants for program verification. Our proposed end-to-end reasoning framework learns to compose the solution automatically without any manual labels. It solves a comparable number of benchmarks as the state-of-the-art solvers while requiring much fewer queries to a theorem prover. Moreover, after being pre-trained, it can generalize the strategy to new instances much faster than starting from scratch. In the future, we plan to extend the framework to discover loop invariants for larger programs which present more confounding variables, as well as to discover other kinds of program correctness properties such as *ranking functions* for proving program termination [46] and *separation predicates* for proving correctness of pointer-manipulating programs [143]. Furthermore, inputs to the framework is not restricted to programs and can be anything reducible to graphs. Similarly, the interaction is not restricted to a theorem prover and can be any checker adaptable to provide a numerical reward.

This chapter is adapted from the following published work:

✎ Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning Loop Invariants for Program Verification. In *Advances in Neural Information Processing Systems, NeurIPS 2018, Montréal, Canada*, pages 7762–7773.

CHAPTER 6

INTRIGUING EXTENSIONS OF CODE2INV

In this section, we discuss two preliminary but intriguing extensions of Code2Inv, the deep reinforcement learning framework presented in the last section. Though Code2Inv is originally designed for generating loop invariants, it can be extended in many flexible ways. This is in part due to that Code2Inv follows the design of the widely used CEGIS framework. On the other hand, deep learning models, especially graph neural networks, provide a whole new dimension of flexibility. Although Code2Inv is an end-to-end learning framework, it can be conceptually decomposed into two parts: the frontend, which embeds *programs* into high-dimensional vectors (also called neural representation), and the backend, which maps the neural representation to *invariants*. The frontend and backend are trained jointly by interacting with a *checker*. The *programs*, *invariants*, and *checker* are all configurable. Programs are essentially graphs, and invariants are some structural output or solution, which can be validated by a checker.

In this chapter, we first present a general formalization of the Code2Inv framework, which is more familiar to the PL/FM community, and then discuss two intriguing extensions on constrained Horn clauses (CHC) solving and syntax-guided program synthesis task.

6.1 Formalization

In the last chapter, Code2Inv is simply presented with some high-level intuition followed by neural network structure description and details of training. Though the semantics of neural networks are not well-understood yet, here we aim to give

a formal treatment of Code2Inv, demystifying the cryptic nature of neural networks and providing meaningful insights for various extensions.

Domains of Program Structures:

$\boldsymbol{\mathcal{G}}(T)$	$=$	G_{inst}	(G_{inst} is graph rep. of *verification instance* T)
$\boldsymbol{\mathcal{G}}(A)$	$=$	G_{inv}	(G_{inv} is graph rep. of *invariant grammar* A)
A	$=$	$\langle \Sigma \uplus H, N, P, S \rangle$	(invariant grammar)
x	$\in$	$H \uplus N$	(placeholder symbols and non-terminals)
v	$\in$	Σ	(terminals)
n	$\in$	N	(non-terminals)
p	$\in$	P	(production rule)
		S	(start symbol)
inv	$\in$	$\boldsymbol{\mathcal{L}}(A)$	(invariant candidate)
cex	$\in$	$\mathbb{C}$	(counterexample)
C	$\in$	$\boldsymbol{\mathcal{P}}(\mathbb{C})$	(set of counterexamples)
$check(T, inv)$	$\in$	$\{\bot\} \uplus \mathbb{C}$	(invariant validation)

Domains of Neural Structures:

π	$=$	$\langle \nu_{\text{T}}, \nu_{\text{A}}, \eta_{\text{T}}, \eta_{\text{A}}, \alpha_{\text{ctx}}, \epsilon_{\text{inv}} \rangle$	(neural policy)
d			(positive integer size of embedding)
$\nu_{\text{T}},\ \eta_{\text{T}}(G_{\text{inst}})$	$\in$	$\mathbb{R}^{\lvert G_{\text{inst}} \rvert \times d}$	(graph embedding of verification instance)
$\nu_{\text{A}},\ \eta_{\text{A}}(G_{\text{inv}})$	$\in$	$\mathbb{R}^{\lvert G_{\text{inv}} \rvert \times d}$	(graph embedding of invariant grammar)
ctx	$\in$	$\mathbb{R}^d$	(neural context)
state	$\in$	$\mathbb{R}^d$	(partially generated invariant state)
α_{ctx}	$\in$	$\mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d$	(attention context)
ϵ_{inv}	$\in$	$\boldsymbol{\mathcal{L}}(A) \to \mathbb{R}^d$	(invariant encoder)
$aggregate$	$\in$	$\mathbb{R}^{\text{k} \times d} \to \mathbb{R}^d$	(aggregation of embeddings)
$\nu_{\text{A}}[n]$	$\in$	$\mathbb{R}^{\text{k} \times d}$	(embedding of prod. rules for non-terminal n, where k is #production rules of n in G_{inv})
$\nu_{\text{T}}[h]$	$\in$	$\mathbb{R}^{\text{k} \times d}$	(embedding of nodes annot. by placeholder h, where k is #nodes annotated by h in G_{inst})

Figure 6.1: Semantic domains of Code2Inv.

Fig. 6.1 defines the domains of program structures and neural structures used in Code2Inv. The framework is parameterized by graph constructors $\boldsymbol{\mathcal{G}}$ that produce graph representations of a verification instance T and an invariant grammar A, denoted G_{inst} and G_{inv}, respectively. The invariant grammar uses placeholder symbols H, which represent *abstract* values of entities such as variables, constants, and

operators, and will be replaced by *concrete* values from the verification instance during invariant generation. The framework requires a black-box function *check* that takes a verification instance T and a candidate invariant inv, and returns success (denoted $\perp$) or a counterexample cex.

The key component of the framework is a neural policy π which comprises four neural networks. Two graph neural networks, η_{T} and η_{A}, are used to compute neural embeddings, ν_{T} and ν_{A}, for graph representations G_{inst} and G_{inv}, respectively. The neural network α_{ctx}, implemented as a GRU, maintains the attention context *ctx* which controls the selection of the production rule to apply or the concrete value to replace a placeholder symbol at each step of invariant generation. The neural network ϵ_{inv}, implemented as a Tree-LSTM, encodes the partially generated invariant into a numeric vector denoted *state*, which captures the state of the generation that is used to update the attention context *ctx*.

Algorithm 4 depicts the main algorithm underlying Code2Inv. It takes a verification instance and a proof checker as input and produces an invariant that suffices to verify the given instance[12]. At a high level, Code2Inv learns a neural policy, in lines 1-5. The algorithm first initializes the neural policy and the set of counterexamples (line 1-2). The algorithm then iteratively samples a candidate invariant (line 4) and improves the policy using a reward for the new candidate based on the accumulated counterexamples (line 5). We next elucidate upon the initialization, policy sampling, and policy improvement procedures.

Initialization. The `initPolicy` procedure (line 6-10) initializes the neural policy. All four neural networks are initialized with random weights (line 7), and graph embeddings $\nu_{\text{T}}, \nu_{\text{A}}$ for verification task T and invariant grammar A are computed by applying corresponding graph neural networks $\eta_{\text{T}}, \eta_{\text{A}}$ to their graph representations $\boldsymbol{\mathcal{G}}(T), \boldsymbol{\mathcal{G}}(A)$ respectively. Alternatively, the neural networks can be initialized with pre-trained weights, which can boost overall performance.

[12]Fuzzers may be applied first so that the confidence of existence of a proof is high.

Algorithm 4: Code2Inv Framework

Input: a verification instance T and a proof checker *check*
Output: a invariant inv satisfying *check*$(T, inv) = \bot$
Parameter: graph constructor $\mathcal{G}$ and invariant grammar A

$\pi \quad$ `initPolicy(`T, A`)`
$C \leftarrow \emptyset$
while *true* **do**
  $inv \leftarrow$ `sample(`π, T, A`)`
  $\langle \pi, C \rangle \leftarrow$`improve(`$\pi, inv, C$`)`

Function `initPolicy(`T, A`)`
  Initialize weights of $\eta_{\text{T}}, \eta_{\text{A}}, \alpha_{\text{ctx}}, \epsilon_{\text{inv}}$ with random values
  $\nu_{\text{T}} \quad \eta_{\text{T}}(\mathcal{G}(T))$
  $\nu_{\text{A}} \quad \eta_{\text{A}}(\mathcal{G}(A))$
  return $\langle \nu_{\text{T}}, \nu_{\text{A}}, \eta_{\text{T}}, \eta_{\text{A}}, \alpha_{\text{ctx}}, \epsilon_{\text{inv}} \rangle$

Function `sample(`π, T, A`)`
  $inv \leftarrow A.\text{S}$
  $ctx \leftarrow aggregate(\pi.\nu_{\text{T}})$
  while *inv* is partially derived **do**
    $x \leftarrow$ leftmost non-terminal or placeholder symbol in inv
    state $\leftarrow \pi.\epsilon_{\text{inv}}(inv)$
    $ctx \quad \pi.\alpha_{\text{ctx}}(ctx, state)$
    if *x* is non-terminal **then**
      $p \quad$ `attention(`$ctx, \pi.\nu_{\text{A}}[x], \mathcal{G}(A)$`)`
      expand inv according to p
    else
      $v \quad$ `attention(`$ctx, \pi.\nu_{\text{T}}[x], \mathcal{G}(T)$`)`
      replace x in inv with v
  return inv

Function `improve(`π, inv, C`)`
  $n \leftarrow$ number of counter-examples C that inv can satisfy
  if $n = |C|$ **then**
    $cex \leftarrow$ *check*(T, inv)
    if $cex = \bot$ **then**
      save inv and weights of π
      exit `// a sufficient invariant is found`
    else
      $C \quad C \cup \{cex\}$
  $r \leftarrow n/|C|$
  $\pi \quad$ `updatePolicy(`π, r`)`
  return $\langle \pi, C \rangle$

Function `updatePolicy(`π, r`)`
  Update weights of $\pi.\eta_{\text{T}}, \pi.\eta_{\text{A}}, \pi.\alpha_{\text{ctx}}, \pi.\epsilon_{\text{inv}}, \pi.\nu_{\text{T}}, \pi.\nu_{\text{A}}$ by
  standard policy gradient [175] using reward r

Function `attention(`ctx, ν, G`)`
  Return node t in G such that dot product of ctx and $\nu[t]$
  is maximum over all nodes of G

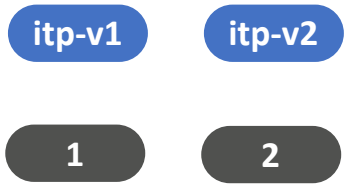
Neural policy sampling. The `sample` procedure (lines 11-24) generates a candidate invariant by executing the current neural policy. The candidate is first initialized to the start symbol of the given grammar (line 12), and then updated iteratively (lines 14-23) until it is complete (i.e. there are no non-terminals). Specifically, the candidate is updated by either expanding its leftmost non-terminal according to one of its production rules (lines 19-20) or by replacing its leftmost placeholder symbol with some concrete value from the verification instance (lines 22-23). The selection of a production rule or concrete value is done through an *attention mechanism*, which picks the most likely one according to the current context and corresponding region of external memory. The neural context is initialized to the aggregation of embeddings of the given verification instance (line 13), and then maintained by α_{ctx} (line 17) which, at each step, incorporates the neural state of the partially generated candidate invariant (line 16), where the neural state is encoded by ϵ_{inv}.

Neural policy improvement. The `improve` procedure (lines 25-36) improves the current policy by means of a *continuous* reward. Simply checking whether the current candidate invariant is sufficient or not yields a discrete reward of 1 (yes) or 0 (no). This reward is too sparse to improve the policy, since most candidate invariants generated are insufficient, thereby almost always yielding a zero reward. Code2Inv addresses this problem by accumulating counterexamples provided by the checker. Whenever a new candidate invariant is generated, Code2Inv tests the number of counterexamples it can satisfy (line 26), and uses the fraction of satisfied counterexamples as the reward (line 34). If all counterexamples are satisfied, Code2Inv queries the checker to validate the candidate (line 28). If the candidate is accepted by the checker, then a sufficient invariant was found, and the learned weights of the neural networks are saved for speeding up similar verification instances in the future (lines 29-31). Otherwise, a new counterexample is accumulated (line 33). Finally, the neural policy (including the neural embeddings) is updated based on the reward (line 35).

6.2 Code2Inv as a CHC solver

```
(set-logic HORN)
(declare-rel itp (Int Int))
...
(rule (=> (and (itp D C)
               (= A (+ 2 C))
               (= B (+ 1 D)))
          (itp B A)))
...
```

(a) CHC instance snippet



(b) node representation

Figure 6.2: A snippet of CHC instance and its corresponding degenerate graph representation, i.e. node representation.

Constrained Horn Clauses (CHC) are a uniform way to represent recursive, inter-procedural, and multi-threaded programs, and serve as a suitable basis for automatic program verification [32] and refinement type inference [121]. Solving a CHC instance involves determining unknown predicates that satisfy a set of logical constraints. Figure 6.2a shows a simple example of a CHC instance where itp is the unknown predicate. It is easy to see that itp in fact represents an invariant of a loop. Thus, CHC solving can be viewed as a generalization of finding loop invariants [32]. This close connection suggests Code2Inv can be immediately extended as a CHC solver, for which the only required change is to have a new frontend that embeds constraint Horn clauses into high dimensional vectors.

Unlike C programs, which have explicit control-flow and data-flow information, a CHC instance is a set of *un-ordered* Horn rules. The graph construction for Horn rules is not as obvious as for C programs. Therefore, instead of deliberately constructing a graph that incorporates detailed domain-specific information, we use a *node representation*, which is a degenerate case of graph representation and requires only necessary nodes but no edges. Figure 6.2b shows the node representation for the CHC example from Figure 6.2a. The top two nodes are derived from the signature of unknown predicate itp and represent the first and the second arguments of

itp. The bottom two nodes are constants extracted from the Horn rule. We empirically show that node representation works reasonably well. The downside of node representation is that no structural information is captured by the neural embeddings which in turn prevents the learned neural policy from generalizing to other structurally similar instances.

6.2.1 Empirical Evaluations

Benchmarks and evaluation setup. We collect 120 CHC instances using SeaHorn [84] to reduce the C benchmark programs into CHCs[13]. In addition, we also introduce 7 CHC instances that involve *non-linear* arithmetics, which are meant to be challenging. We compare Code2Inv with two state-of-the-art CHC solvers: Spacer [100], which is the default fixedpoint engine of Z3, and DataDrivenCHC [195], which extends Spacer with decision tree learning. We run all solvers on a single 2.4 GHz AMD CPU core up to 12 hours and using up to 4 GB memory. Unless specified otherwise, Code2Inv is always initialized randomly, that is, untrained.

Preliminary results. Our evaluation shows that, without any prior knowledge about Horn rules, Code2Inv can solve 94 (out of 120) CHC instances. Although it is not on a par with state-of-the-art CHC solvers Spacer and LinearArbitrary, which solve 112 and 118 instances, respectively, Code2Inv provides new insights for solving CHCs and could be further improved by better embeddings and reward design.

However, among the 7 challenging instances involving non-linear arithmetic, our case study shows that Code2Inv successfully solves 5 of them, while both Spacer and DataDrivenCHC failed to solve any of them. Tasks involving non-linear arithmetic are particularly challenging because the underlying checker is more likely to get stuck, and no feedback (e.g. counterexample) can be provided, which is critical for

[13] SeaHorn produces empty Horn rules on 13 (out of 133) C programs due to optimizations during VC generation that result in proving the assertions of interest.

existing solvers like Spacer and DataDrivenCHC to make progress. This highlights another strength of Code2Inv—even if the checker gets stuck, the learning process can still continue by simply assigning zero or negative reward.

```
Solution found by Spacer:
(and (or (not (<= B 16)) (not (>= A 8)))
         (not (<= B 0))
     (or (not (<= B 2)) (<= A 0))
     (or (not (<= B 4)) (not (>= A 2)))
     (or (not (<= B 6)) (not (>= A 3)))
     (or (not (<= B 8)) (not (>= A 4)))
     (or (not (<= B 10)) (not (>= A 5)))
     (or (not (<= B 12)) (not (>= A 6)))
     (or (not (<= B 14)) (not (>= A 7)))))))

Code2Inv: (<= v0 (- v1 v0))
```

(a) Spacer on add2.smt

```
Solution found by LinearArbitrary:
(or
  (and true !(V0<=-50)
           V1<=5  ((1*V0)+(-1*V1))<=-45
           V1<=4  !(((1*V0)+(-1*V1))<=-51)
          !(V1<=2)!(((1*V0)+(-1*V1))<=-50)
          !(V1<=3) ((1*V0)+(1*V1))<=-40
  )
  ... // omitting other 4 similar (and ...)
)

Code2Inv: (or (< V0 (+ 0 0)) (> V1 V0))
```

(b) LinearArbitrary on 84.c.smt

Figure 6.3: Comparison of solution naturalness.

Naturalness. We further share a case study on the naturalness of solutions. As illustrated in Fig. 6.3, solutions discovered by Code2Inv tend to be more natural, whereas Spacer and LinearArbitrary tend to find solutions that unnecessarily depend on constants from the given verification instance. Such *overfitted* solutions may become invalid when these constants change. Note that expressions such as `(+ 0 0)` in Code2Inv's solutions can be eliminated by post-processing simplification akin to peephole optimization in compilers. Alternatively, the reward mechanism in Code2Inv could incorporate a regularizer on the naturalness.

6.3 Meta-Learning for Syntax-guided Synthesis

Program synthesis concerns automatically generating a program that satisfies desired functional requirements. The *Syntax-Guided Synthesis* (SyGuS) [13] poses a common formulation — the program synthesizer takes as input a logical formula ϕ and a grammar G, and produces as output a program in G that satisfies ϕ. In this formulation, ϕ constitutes a semantic specification that describes the desired functional requirements, and G is a syntactic specification that constrains the space

of possible programs. Figure 6.4 shows us an instance of the SyGuS problem, which synthesizes a new and equivalent circuit following the given grammar that is properly designed to control timing channels. Our discussion and evaluation will focus on cryptographic circuit synthesis tasks, but the presented idea is applicable for other SyGuS tasks.

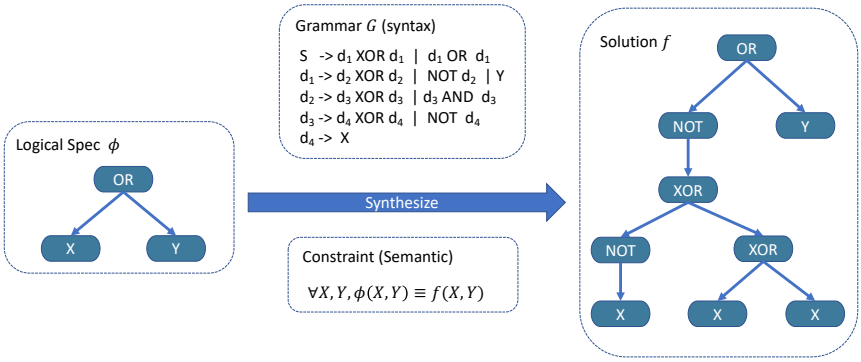


Figure 6.4: An example of a circuit synthesis task from the 2017 SyGuS competition. Given the original program specification which is represented as an abstract syntax tree (left), the solver is tasked to synthesize a new circuit f (right). The synthesis process is specified by the syntactic constraint G (top), and the semantic constraint (bottom) specifies that f must have functionality equivalent to the original program.

A straightforward extension of Code2Inv is to view the logical specification as a program, design actions according to the given grammar, and replace the loop invariant checker with a corresponding checker for the given synthesis task. Such an approach essentially customizes Code2Inv for one particular synthesis task, which makes reinforcement learning process an adaptive search. Although this is a feasible solution, the learned policy cannot generalize to different synthesis tasks because actions are hardcoded according to the given grammar, which varies from task to task. Note that this is not an issue for loop invariants, because the invariant grammar (e.g. CNF) is generally shared across different programs.

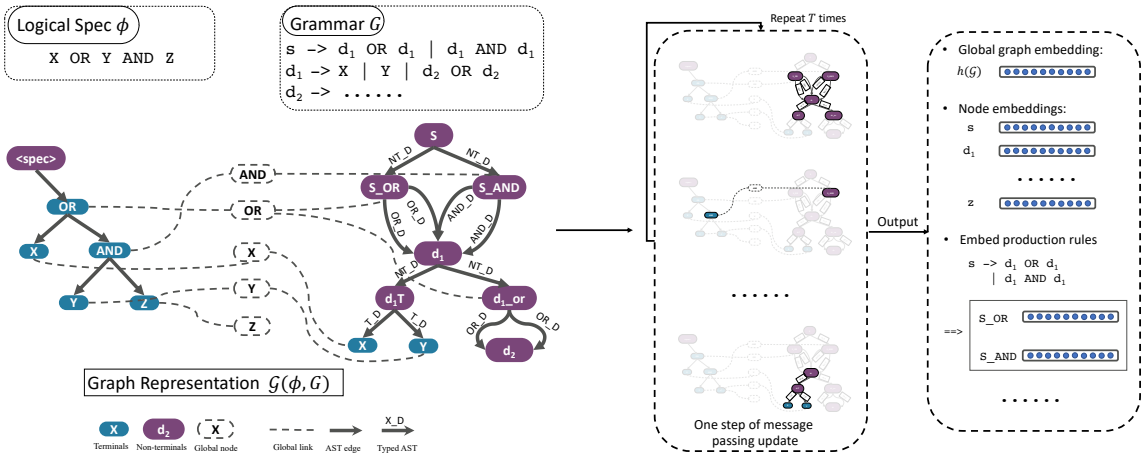


Figure 6.5: Graph representation of a cryptographic circuit synthesis task. Logical specification and grammar are jointly represented as a graph.

6.3.1 A Meta-learning Extension.

We explore an extension that is suitable for meta-learning, i.e. learning across different SyGuS tasks, each of which has a different grammar. The key insight is to make the grammar as part of the *external memory* accessed by the neural agent (i.e. the backend of Code2Inv). That is, both the specification ϕ and the grammar G are viewed as "an input program" and represented as a graph, which is then embedded as high-dimensional vectors (i.e. external memory).

Figure 6.5 illustrates how to jointly represent the specification and grammar as a graph for a cryptographic circuit synthesis task. The graph consists of two parts — the part on the left is an abstract syntax tree of the specification ϕ, and the other part on the right is a graph representation of the grammar G, each node of which corresponds to either a non-terminal or a terminal symbol or a production rule in the grammar. In addition, these two parts are linked together by unique global copy for shared symbols between them.

Then, we only need to slightly adapt the backend of Code2Inv. Instead of taking hardcoded actions, the backend selects the most likely action stored in the external memory, specifically the embeddings of production rules, through the attention

mechanism.

6.3.2 Empirical Evaluations

We evaluate this extension, named MetaL[14] on 214 cryptographic circuit synthesis tasks, which are from the general track of the 2017 SyGuS competition[176]. The experiments are conducted in two learning settings. First, we test MetaL as an out-of-box solver, which is directly applied on a synthesis task without training. This setting enables us to compare MetaL to classical solvers developed in the formal methods community. As those solvers do not utilize learning-based strategies, it is sensible to also limit MetaL not to carry over prior knowledge from a separate training set. Second, we evaluate MetaL as a meta-solver which is trained over a training set $\mathcal{D}$, and finetuned on each of the new tasks in a separate set $\mathcal{D}'$. In this setting, we aim to demonstrate that MetaL is capable of learning a transferable representation and policy in order to efficiently adapt to unseen tasks.

6.3.3 Learning an Out-of-Box Solver

Table 6.1: Number of instances solved using: 1) EUSolver, 2) CVC4, 3) ESymbolic, and 4) MetaL (out-of-box). For each solver, the maximum time in solving an instance and the average and median time over all solved instances are also shown below.

	# instances solved	Max time	Avg time	Median time
EUSolver	153 / 214	1h39m	3m	3s
CVC4	129 / 214	5h50m	30m	6s
ESymbolic	31 / 214	40m	8m	5m
MetaL (out-of-box)	141 / 214	4h11m	33m	3m

In the out-of-box solver setting, we compare MetaL against solvers built based on two classical approaches: a SAT/SMT constraint solving based approach and a

[14]The code and data are available on GitHub: https://github.com/PL-ML/metal

search based approach. For the former, we choose CVC4 [142], which is the state-of-the-art SMT constraint solver; for the latter, we choose EUSolver [16], which is the winner of the SyGuS 2017 Competition [15]. Furthermore, we build a search based solver as baseline, ESymbolic, which systematically expands non-terminals in a predefined order (e.g. depth-first-search) and effectively prunes away partially generated candidates by reducing it to 2QBF [24] satisfiability check. ESymbolic can be viewed as a generalization of EUSolver by replacing the carefully designed domain-specific heuristics (e.g. indistinguishability and unification) with 2QBF.

In order to make the comparison fair, we run all solvers on the same platform with a single core CPU[15]. We measure the performance of each solver by counting the number of instances it can solve given a 6 hours limit spent on each task.

Table 6.1 summarizes the total number of instances solved by each solver as well as the maximum, average and median running time spent on solved instances. In terms of the absolute number of solved instances, MetaL is not yet as good as EUSolver, which is equipped with specialized heuristics. However, EUSolver fails to solve 4 instances that are only solved by our framework. All instances solved by CVC4 and ESymbolic are a strict subset of instances solved by EUSolver. Thus, besides being a new promising approach, our framework already plays a supplementary role for improving the current state-of-the-art. Compared with the state-of-the-art CVC4 solver, MetaL has smaller maximum time but higher average and median time usage. This suggests that MetaL excels at solving difficult instances with better efficiency.

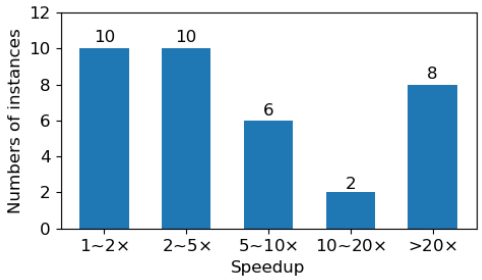
6.3.4 Learning Across Different Tasks

We next evaluate whether MetaL is capable of learning transferable knowledge across different synthesis tasks. We randomly split the 214 circuits synthesis tasks

[15]The SyGuS 2017 competition gives each solver 4-core 2.4GHz Intel processors with 128 GB memory and wallclock time limit of 1 hour; our evaluation uses AMD Opteron 6220 processor, assigns each solver a single core with 32 GB memory and wallclock time limit of 6 hours.

# candidates generated	Percentage solved		
	20%	40%	60%
MetaL (out-of-box)	2564	18K	102K
MetaL (pre-trained)	205	4.5K	59K
Reduction	12.5×	4.0×	1.7×

(a)



(b)

Figure 6.6: Performance improvement with meta-learning. (a) Accumulated number of candidates generated in order to solve 20%, 40%, and 60% of the testing tasks; and (b) speedup distribution over individual instances.

into two sets: 150 tasks for training and the rest 64 tasks for testing. MetaL is then trained on the training set for 35000 epochs. For each epoch, a batch of 10 tasks are sampled. The gradients of each task are averaged and applied to the model parameters using Adam optimizer. In testing phase, MetaL is finetuned on each task in the testing set until either a correct program is synthesized or timeout occurs.

We compare the trained meta-solver with the out-of-box solver in solving tasks in the test set. Out of 64 testing tasks, the out-of-box solver and meta-solver can solve 36 and 37 tasks, respectively. Besides the additional task solved, the performance is also greatly improved by meta-solver, which is shown in Figure 5. Table 5(a) shows the accumulated number of candidates generated to successfully solve various ratios of testing tasks. We see that the number of explored candidates by meta-solver is significantly reduced: for 40% of testing tasks (i.e., 66% of solved tasks), meta-learning enable 4x reduction on average. The accumulated reduction for all solved tasks (60% of testing tasks) is not that significant. This is because meta-learning improve dramatically for most (relatively) easy tasks but helps slightly for a few hard tasks, which actually dominate the number of generated candidates. Figure 5(b) shows the speedup distribution over the 36 commonly solved tasks. Meta-solver achieves at least 2x speedup for most benchmarks, orders of magnitude improvement for 10 out of 36 unseen tasks, and solves one task that is not solvable without

meta-learning.

6.4 Discussion

In this chapter, we formalize the Code2Inv framework and demonstrate two interesting extensions on constraint Horn clause (CHC) solving and syntax-guided program synthesis (SyGuS) tasks, respectively. Our evaluation indicates that Code2Inv outperforms the specialized state-of-the-art solvers on many small yet challenging instances. To make Code2Inv a competitive tool in general, there are a few important challenges to address. First, a modular and scalable representation is necessary. Second, the effectiveness of deep learning as well as meta-learning is still an empirical observation, which deserves a formal analysis. Third, the reward design should leverage more structural information, instead of simple statistics.

This chapter is adapted from the following published work:

- ✎ Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. Code2Inv: A Deep Learning Framework for Program Verification. In *Proceedings of 32nd International Conference on Computer-Aided Verification, CAV 2020*, Los Angeles, California, USA.

- ✎ Xujie Si, Yuan Yang, Hanjun Dai, Mayur Naik, and Le Song. Learning a Meta-Solver for Syntax-Guided Program Synthesis. In *Proceedings of 7th International Conference on Learning Representations, ICLR 2019*, New Orleans, Louisiana, USA.

CHAPTER 7

FUTURE WORK

Recent advances of machine learning, especially deep learning, foster numerous applications in various domains. In this dissertation, we have studied a few exciting applications of machine learning on tasks involving sophisticated logical reasoning, specifically logical rule synthesis, program verification, and constraint solving. We would like to outline a number of future directions on combining machine learning and logical reasoning, which we envision will play an important role in advancing artificial intelligence and broadening its application scope.

Reasoning with perceptual data. Deep learning has made remarkable successes in computer vision and surpassed the human-level performance on many classification tasks [58, 92]. However, it has limited capability of performing reasoning. For instance, deep neural networks can be trained to classify images over one thousand categories (e.g. animals, plants, foods, etc) very accurately, but cannot answer simple queries like "what are things with two ears in the image?", "will the ball drop if the man loosen his hand?", or "is the baby happy?". This is largely due to the lack of interpretability as well as common sense. Of course, understanding the query itself might be a challenging task, but even if the query is stated in an unambiguous and succinct logical form, deep learning models would fail to answer these queries.

One promising way to address these challenges is to equip deep learning models with an explicitly reasoning component. Doing so, on on hand, the learned system has a good interpretability; on the other hand, it is easy to incorporate commonsense knowledge, which also significantly improves data efficiency. In Chapter 4, we show that gradient-based approaches can be used to learn interpretable logical

rules. Given that gradient-based methods have also been successfully applied to perceptual data like images, it is foreseeable to adapt Difflog to perceptual data processing, which adds reasoning capability to the learned model. Furthermore, common sense knowledge can be encoded as relational facts, which are directly taken as input by Difflog.

Learning domain specific heuristics. Designing effective heuristics is arguably the commonest endeavor across all research areas of computer science. The effort and expertise involved are non-trivial — an effective heuristic could be worth an award winning Ph.D. thesis [192]. Heuristics are unavoidable due to fundamental challenges like NP-Hardness or undecidability, for which there is no known algorithms that are efficient (i.e. of polynomial time) or terminating eventually. Can these heuristics be learned rather than designed? Recently, there has been a number of work learning search heuristics [54, 30, 111] for various NP-Complete problems like minimum vertex cover (MVC), traveling salesman problem (TSP) and Boolean satisfiability (SAT). Existing approaches have a specialized design for each problem, limiting space of possible heuristics. A more flexible and general approach is to view heuristics learning as program synthesis, similar to what we have discussed in Chapter 6. Because a heuristic can be viewed as a program in some domain specific language (DSL). We have already shown that Code2Inv can be customized as a CHC solver. Instead of letting the machine learning model directly explore in the solution space, it is more promising to explore in the space of heuristics. We envision this general idea will help to learn effective heuristics for many important problems in software development and maintenance, such as proof assistant automation, software model checking, software testing, fuzzing, symbolic execution, compiler optimization, to name a few, and problems beyond

Robustness and fairness of machine learning. In this thesis, we borrow and adopt techniques developed in the machine learning community to improve pro-

gram verification and synthesis. Exploring the opposite direction, that is, using verification and synthesis techniques to improve machine learning, is equally interesting and perhaps more profound. It is a big surprise that a well-trained machine learning model could give a dramatically different prediction when only a few pixels are perturbed. This is known as adversarial examples [71]. The existence of such adversarial examples in real world [103] makes machine learning applications concerning in many safety critical areas like autonomous driving, disease diagnosis and medical devices. Furthermore, another concern is that machine learning might make biased decisions [48] in many social activities like hiring, commercial loan, criminal justice, etc. Introducing more training data and reducing data bias are helpful ways to improve robustness and fairness, however, which cannot provide any formal guarantees. Program verification and analysis techniques developed in formal methods and programming languages community have a great potential in addressing these critical concerns of machine learning. The key insight is that machine learning models can be viewed as programs, and robustness and fairness can be formalized as certain properties of those programs. Recently, there are a few work [94, 9, 69] in this line of research, which leverages standard SMT solving techniques and abstract interpretation to verify robustness and fairness of deep neural networks. More promising applications of formal reasoning techniques in machine learning research are forthcoming given that machine learning is becoming ubiquitous in various aspects of our daily life.

CHAPTER 8

CONCLUSION

This thesis presents a new paradigm for program reasoning asks, specifically, program synthesis and program verification, which improves the state-of-the-art techniques in a general and end-to-end learning fashion, completely different from the traditional case-by-case manual design. We leverage recent advances in machine learning community and draw inspirations from the formal methods and programming language community, especially the popular counterexample-guided inductive synthesis (CEGIS) framework.

The machine learning view of sophisticated logical reasoning tasks enables a whole new dimension of innovations — correct proofs, desired programs, and efficient heuristics can be automatically learned through self-supervision, e.g. interacting with underlying checkers. These checkers could be a simple numerical function computing losses like mean squared error (MSE) and cross-entropy, which are suitable for traditional machine learning applications, or a non-trivial symbolic reasoning process like SAT solving, SMT solving, and least fixed-point computation, which are unavoidable for program reasonings. This thesis developed techniques passing learning signals through such complicated symbolic reasoning process, which is essential for an end-to-end learning framework for program reasoning. First, this thesis proposes a novel numerical relaxation of logical rules, making it feasible to synthesize a rich set of logical rules using efficient gradient-based approaches. In addition, a sound early termination condition is proposed, and parallelization could further speed up the synthesis due to the stochastic nature of the optimization process. Second, this thesis proposes a deep reinforcement learning framework for program verifications, where graph neural networks are used to learn effective

representations of programs and reinforcement learning is used to infer loop invariants, which is achieved in a sample efficient way because of a counterexample-based continuous reward mechanism. Furthermore, the same deep reinforcement learning framework is formalized and extended to handle quite different tasks such as solving constrained Horn clauses and syntax-guided program synthesis.

This thesis explored a few point successes of leaning-aided design in the domain of program verification and synthesis. Looking forward, this insight applies to a broad range of problems from low-level constraint solving such as SAT solving, SMT solving, and combinatorial optimization to high-level program reasoning and testing such as software model checking, automated theorem proving, symbolic execution, fuzzing, and concolic testing. On the other hand, these program reasoning techniques have a great potential in improving and verifying properties like robustness and fairness of machine learning models, since machine learning models are essentially programs as well.

APPENDIX A

PROOFS AND ARTIFACTS

A.1 Proofs of Properties in Chapter 4

Theorem 4.4.2. *Let $S = (\mathcal{I}, \mathcal{O}, I, O_+, O_-, R)$ be a synthesis problem such that there exists a solution to S. Let $\mathbf{P}$ be the output of ALPS. Then:*

1. *(Soundness) Every $P \in \mathbf{P}$ is a solution to S.*
2. *(Completeness) For every solution $P \in \mathcal{H}$ to S, there exist programs $P_l, P_u \in \mathbf{P}$ such that $P_l \sqsubseteq P \sqsubseteq P_u$. An immediate corollary is that if there exists a program P that is a solution to S, then $\mathbf{P}$ is nonempty.*
3. *(Termination) ALPS terminates.*

To prove Theorem 4.4.2, we use two key properties about the interplay between QBC and bidirectional search. The following lemma captures the fact that the algorithm does not miss any controversial examples, and thus always makes progress in terms of pruning the search space.

Lemma A.1.1. *Let $\mathbf{P} \subseteq \mathcal{H}$ and $e \in \mathcal{B}$. Then $D(e, \mathbf{P}) \neq 0$ iff there exist programs $P_1, P_2 \in \mathbf{P}$ such that $P_1 \cup I \models e$ and $P_2 \cup I \not\models e$.*

Proof. This is directly implied by the definition of vote entropy (see Definition 4.4.1). □

The next lemma states key invariants that hold at every iteration of the algorithm: (*i*) It does not miss any programs that are solutions to the synthesis problems, by ensuring that the contours of the version space form an upper/lower bound of every solution. (*ii*) It ensures that if the current version space contains non-solutions,

then there are non-zero entropy examples we can ask the oracle that can eliminate them.

Lemma A.1.2 (Invariant)**.** *Let $S = (\mathcal{H}, \mathcal{O}, I)$ be a synthesis problem such that a solution to S exists in $\mathcal{H}$. Let $E = (E^+, E^-)$ be the set of known examples at any point during execution, and $\mathbf{P} = \overline{\mathbf{P}} \cup \underline{\mathbf{P}}$. Then:*

1. *For every solution $P' \in \mathcal{H}$ to S, there exist programs $P_l, P_u \in \mathbf{P}$ such that $P_l \sqsubseteq P' \sqsubseteq P_u$.*

2. *If there exists a program $P \in \mathbf{P}$ that is not a solution to S, then $\exists e \in \mathcal{B}.\, D(e, \mathbf{P}) \neq 0$.*

Proof. We first show item (1). First, notice that every solution $P' \in \mathcal{H}$ to S belongs in the version space $\mathcal{V}_E$, since it satisfies all current examples. Since $\mathbf{P} \supseteq \max(\mathcal{V}_E)$ (by the definition of $F^{\uparrow}$ in Algo. 1), there exists $P_u \in \mathbf{P}$ such that $P' \sqsubseteq P_u$. Similarly, since $\mathbf{P} \supseteq \min(\mathcal{V}_E)$, there exists $P_l \in \mathbf{P}$ such that $P_l \sqsubseteq P'$.

We next show item (2). Suppose that $P \in \mathbf{P}$ is not a solution to S, and let P' be a solution to S. Then, there exists an example $e \in \mathcal{B}$ such that either (a) $P \cup I \models e$ and $P' \cup I \not\models e$, or (b) $P \cup I \not\models e$ and $P' \cup I \models e$. We now distinguish two different cases:

- Case (a) holds: since P' is a solution, item (1) tells us that there exists $P_u \in \mathbf{P}$ such that $P' \sqsubseteq P_u$. This implies that $P_u \cup I \not\models e$. Because now P, P_u disagree on example e, Lemma A.1.1 implies that $D(e, \mathbf{P}) \neq 0$.

- Case (b) holds: since P' is a solution, item (1) tells us that there exists $P_l \in \mathbf{P}$ such that $P_l \sqsubseteq P'$. This implies that $P_l \cup I \models e$. Because now P, P_l disagree on example e, Lemma A.1.1 implies that $D(e, \mathbf{P}) \neq 0$.

Thus, in both cases we find an example e such that $D(e, \mathbf{P}) \neq 0$. □

We are ready to prove Theorem 4.4.2 using the Lemmas A.1.1 and A.1.2.

Proof. The algorithm terminates when for every example $e \in \mathcal{B}$, we have $D(e, \mathbf{P}) = 0$. Recall that $\mathbf{P} = \overline{\mathbf{P}} \cup \underline{\mathbf{P}}$, where $E = (E^+, E^-)$ are the known examples.

- (Soundness) As $\forall e \in \mathcal{B}.\, D(e, \mathbf{P}) = 0$, the contrapositive of item (2) of Lemma A.1.2 indicates that every $P \in \mathbf{P}$ is a solution.
- (Completeness) This holds directly from item (1) of Lemma A.1.2.
- (Termination) At every iteration, the algorithm adds one example to either E^+ or E^-. Notice that, after an example e is added to $E = (E^+, E^-)$, it cannot be added again, since it will never be controversial ($D(e, \mathbf{P}) \neq 0$) from that point on. Since we have finitely many examples in $\mathcal{B}$, the algorithm terminates after finitely many steps.

□

Theorem 4.2.2. *Determining whether an instance of the rule selection problem, ($\mathcal{I}$, $\mathcal{O}$, I, O_+, O_-, R), admits a solution is NP-hard.*

Proof. Consider a 3-CNF formula φ over a set V of variables:

$$\varphi = (l_{11} \vee l_{12} \vee l_{13}) \wedge (l_{21} \vee l_{22} \vee l_{23}) \wedge \cdots \wedge (l_{k1} \vee l_{k2} \vee l_{k3}),$$

be the given 3-CNF formula, where each literal l_{ij} appearing in clause c_i is either a variable, $v_{ij} \in V$, or its negation, $\neg v_{ij}$. Assume that there are no trivial clauses in φ, which simultaneously contain both a variable and its negation. We will now encode its satisfiability as an instance of the rule selection problem.

1. For each variable $v \in V$, define the input relations:

$$\texttt{pos}_v = \{(c) \mid v \in c\}, \text{ and} \tag{A.1}$$

$$\texttt{neg}_v = \{(c) \mid \neg v \in c\}, \tag{A.2}$$

consisting of all one-place tuples $\texttt{pos}_v(c)$ and $\texttt{neg}_v(c)$ indicating whether the variable v occurs positively or negatively in the clause c.

2. Also, for each variable v, define the input relation $\texttt{var}_v$ which is inhabited by a single tuple $\texttt{var}_v(v)$:

$$\texttt{var}_v = \{(v)\}. \tag{A.3}$$

3. The idea is to set up the candidate rules so that subsets of chosen rules correspond to assignments of true / false values to the variables of φ. Let $C_2(c, v)$ be an output relation: we are setting up the problem so that if the tuple $C_2(c, v)$ is derivable in the synthesized solution, then there is a satisfying assignment of φ where clause c is satisfied due to the assignment to variable v.

4. For each variable v, create a pair of candidate rules r_v and $r_{\neg v}$ as follows:

$$r_v = \text{“}C_2(c, v') \coloneq \texttt{pos}_v(c), \texttt{var}_v(v')\text{”, and}$$
$$r_{\neg v} = \text{“}C_2(c, v') \coloneq \texttt{neg}_v(c), \texttt{var}_v(v')\text{”.}$$

   Selecting the rule r_v corresponds to assigning the value true to the corresponding variable v, and selecting the rule $r_{\neg v}$ corresponds to assigning it the value false.

5. To prevent the simultaneous choice of rules r_v and $r_{\neg v}$, we set up the three-place input relation $\texttt{conflict}(c, c', v)$, which indicates that the reason for the simultaneous satisfaction of clauses c and c' cannot be a contradictory variable v:

$$\texttt{conflict} = \{(c, c', v) \mid v \in c \text{ and } \neg v \in c'\} \cup \{(a, a, a)\}, \tag{A.4}$$

   where a is some new constant not seen before. We will motivate its necessity

while defining the canary output relation `error` next.

6. We detect the simultaneous selection of a pair of rules r_v and $r_{\neg v}$ using the rule r_e:

$$r_e = \text{“}\texttt{error}(c, c', v) \coloneq C_2(c, v), C_2(c', v), \texttt{conflict}(c, c', v)\text{”}$$

Here `error` is a three-place output relation indicating the selection of an inconsistent assignment. We would like to force the synthesizer to choose the error-detecting rule r_e. The selection of the rule r_e, the presence of the input tuple $\texttt{conflict}(a, a, a)$, and the selection of the rule r_a:

$$r_a = \text{“}C_2(x, x) \coloneq \texttt{conflict}(x, x, x)\text{”}$$

is the only way to produce the output tuple $\texttt{error}(a, a, a)$, which we will mark as desired.

7. The output tuple $C_2(c, v)$ indicates the satisfaction of the clause c because of the assignment to variable v. We use the presence of such tuples to mark the clause c itself as being satisfied: let $C_1(c)$ be a one-place output relation, and include the rule:

$$r_c = \text{“}C_1(c) \coloneq C_2(c, v)\text{”}.$$

8. In summary, let the rule selection problem $P_\varphi = (\mathcal{I}, \mathcal{O}, I, O_+, O_-, R)$ be defined as follows:

   (a) $\mathcal{I} = \{\texttt{var}_v, \texttt{pos}_v, \texttt{neg}_v \mid v \in V\} \cup \{\texttt{conflict}\}$.

   (b) $\mathcal{O} = \{C_2, C_1, \texttt{error}\}$.

   (c) Define the set of input tuples, I, using equations A.1, A.2, A.3, and A.4.

   (d) $O_+ = \{C_1(c) \mid \text{clause } c \in \varphi\} \cup \{\texttt{error}(a, a, a)\}$.

(e) $O_{-} = \{\texttt{error}(c, c', v) \mid \text{clauses } c, c' \text{ and variable } v \text{ occurring in } \varphi\}$.

(f) $R = \{r_v, r_{\neg v} \mid v \in V\} \cup \{r_e, r_a, r_c\}$.

Given a 3-CNF formula φ, the corresponding instance P_φ of the rule selection problem can be constructed in polynomial time. Furthermore, it can be seen that, by construction, P_φ admits a solution iff φ is satisfiable. It follows that the rule selection problem is NP-hard. □

Next, we turn our attention to Theorem 4.5.2. The first part of the claim follows immediately from the definition in Equation 4.2. We therefore focus on the second part: Note that the proof of continuity does not immediately follow from Equation 4.2 because the supremum of an infinite set of continuous functions need not itself be continuous. It instead depends on the observation that there is a finite subset of dominating derivation trees whose values suffice to compute $v_t(\boldsymbol{w})$.

Theorem 4.5.2. *The value of the output tuples, $v_t(\boldsymbol{w})$, varies monotonically with the rule weights $\boldsymbol{w}$, and is continuous in the region $0 < w_r < 1$.*

Proof. Fix an assignment of rule weights $\boldsymbol{w}$. Next, focus on a specific output tuple t, and consider the set of all its derivation trees τ. Let σ_τ be a pre-order traversal over its nodes. For example, for the tree τ_1 in Figure 4.3a, we obtain $\sigma_{\tau_1} = \texttt{samegen}(\texttt{Will}, \texttt{Ann}), r_1(\texttt{Will}, \texttt{Ann}, \texttt{Ben}), \texttt{parent}(\texttt{Will}, \texttt{Ben}), \texttt{parent}(\texttt{Ann}, \texttt{Ben})$. It can be shown that the set of all pre-order traversals, σ_τ, over all derivation trees τ forms a context-free grammar L_t.

We are interested in trees τ with high values $v_\tau(\boldsymbol{w})$, where the value of a tree depends only on the number of occurrences of each rule r. It therefore follows that the weight $v_\tau(\boldsymbol{w})$ is completely specified by the Parikh image, $\{r \mapsto \#r \text{ in } \tau\}$, which counts the number of occurrences of each symbol in each string of the language L_t. From Parikh's lemma, we conclude that this forms a semilinear set. Let

$$p(L_t) = \bigcup_{i=1}^{m} (\boldsymbol{c}_{i0} + \sum_{j=1}^{n} \boldsymbol{c}_{ij})$$

be the Parikh image of L_t, and for each $i \in \{1, 2, \ldots, m\}$, let τ_i be the derivation tree corresponding to the rule count $\boldsymbol{c}_{i0}$. It follows that:

$$v_t(\boldsymbol{w}) = \sup_{\tau \text{ with conclusion } t} v_\tau(\boldsymbol{w}) = \max_{i=1}^{m} v_{\tau_i}(\boldsymbol{w}).$$

We have reduced the supremum over an infinite set of continuous functions to the maximum of a finite set of continuous functions. It follows that $v_t(\boldsymbol{w})$ varies continuously with $\boldsymbol{w}$. □

Finally, we turn to the proof of Theorem 4.5.3.

Theorem 4.5.3. *Fix a set of input relations $\mathcal{I}$, output relations $\mathcal{O}$, and candidate rules R. Let* EVALUATE$(R, \boldsymbol{w}, I) = (F, \boldsymbol{u}, \boldsymbol{l})$. *Then: (a) $F = R(I)$, and (b) $\boldsymbol{u}(t) = v_t(\boldsymbol{w})$. Furthermore,* EVALUATE$(R, \boldsymbol{w}, I)$ *returns in time $poly(|I|)$.*

Proof. The first part of the following result follows from similar arguments as the correctness of the classical algorithm. We briefly describe the proof of the second claim. For each output tuple t, consider all of its derivation trees τ_{hi} with maximal value, and identify the tree τ_t with shortest height among these. All first-level subtrees of τ_t must themselves possess the shortest-height-maximal-value property, so that their height is bounded by the number of output tuples. Since the $(F, \boldsymbol{u}, \boldsymbol{l})$-loop in step 3 of Algorithm 3 has to hit a fixpoint within as many iterations, and since each iteration runs in polynomial time, the claim about running time follows. □

A.2 Artifacts

We make the research artifact (including source code, benchmarks and pre-trained models) of each project presented in this thesis publicly available. The artifact links are shown in Table A.1.

Prototype	Artifact Link	Keywords
ALPS	https://tinyurl.com/y7rfj99a	- logical rule synthesis - bi-directional search - active learning
Difflog	https://tinyurl.com/y7oem4st	- logical rule synthesis - numerical relaxation - Las Vegas algorithm
Code2Inv	https://tinyurl.com/ybvecubj	- loop invariant inference - deep learning - reinforcement learning
MetaL	https://tinyurl.com/yd2sllfs	- crypto-circuits synthesis - meta-learning
APISAN	https://tinyurl.com/y6vuybhx	API sanitizer
Datalog-bench	https://tinyurl.com/y8m6gdyu	benchmark

Table A.1: Research artifact links.

BIBLIOGRAPHY

[1] Bdd-based deductive database. http://bddbddb.sourceforge.net/. [Cited on page 16]

[2] Datomic. https://www.datomic.com/. [Cited on page 16]

[3] Iris (integrated rule inference system) reasoner. http://repo.roscidus.com/java/iris. [Cited on page 16]

[4] Logicblox. http://www.logicblox.com/. [Cited on page 16]

[5] Righting code. http://rightingcode.org/. [Cited on page 56]

[6] Semmle. https://semmle.com/. [Cited on page 16]

[7] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Addison-Wesley, 1995. [Cited on pages 51, 55, and 56]

[8] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2013. [Cited on pages 70 and 80]

[9] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. Fairsquare: probabilistic verification of program fairness. *Proc. ACM Program. Lang.*, 1(OOPSLA):80:1–80:30, 2017. [Cited on page 103]

[10] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. Constraint-based synthesis of Datalog programs. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, CP, 2017. [Cited on pages 32, 54, 57, and 59]

[11] Miltiadis Allamanis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles Sutton. Learning continuous semantic representations of symbolic expressions. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2017. [Cited on page 74]

[12] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018. [Cited on page 13]

[13] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, 2013. [Cited on pages 3, 69, 84, and 94]

[14] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015. [Cited on pages 2, 33, and 36]

[15] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438*, 2017. [Cited on page 98]

[16] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2017. [Cited on pages 5, 84, and 98]

[17] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. Search-based program synthesis. *Commun. ACM*, 61(12):84–93, November

2018. ISSN 0001-0782. doi: 10.1145/3208071. URL https://doi.org/10.1145/3208071. [Cited on page 2]

[18] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, 2011. [Cited on page 16]

[19] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, DIKU, University of Copenhagen, 1994. Ph.D. thesis. [Cited on pages 21, 55, and 56]

[20] Andrew W. Appel. Verified Software Toolchain. In *Proceedings of the 20th European Symposium on Programming (ESOP)*, 2011. [Cited on page 67]

[21] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. Design and implementation of the Logicblox system. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1371–1382. ACM, 2015. [Cited on page 20]

[22] Michael Arntzenius and Neelakantan R. Krishnaswami. Datafun: A functional datalog. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2016. [Cited on page 16]

[23] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015. [Cited on page 13]

[24] Valeriy Balabanov, Jie-Hong Roland Jiang, Christoph Scholl, Alan Mishchenko, and Robert K. Brayton. 2QBF: Challenges and solutions. In Nadia Creignou and Daniel Le Berre, editors, *Proceedings of the International*

Conference on Theory and Applications of Satisfiability Testing (SAT), pages 453–469, 2016. [Cited on page 98]

[25] Thomas Ball and Sriram Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2002. [Cited on page 1]

[26] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017. [Cited on page 85]

[27] V. Bapst, T. Keck, A. Grabska-Barwińska, C. Donner, E. D. Cubuk, S. S. Schoenholz, A. Obika, A. W. R. Nelson, T. Back, D. Hassabis, and P. Kohli. Unveiling the predictive power of static structure in glassy systems. *Nature Physics*, 16(4):448–454, 2020. doi: 10.1038/s41567-020-0842-8. URL https://doi.org/10.1038/s41567-020-0842-8. [Cited on page 2]

[28] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin G. Zorn. Flashrelate: extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2015. [Cited on page 2]

[29] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016. [Cited on pages 68, 78, 83, and 85]

[30] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016. URL http://arxiv.org/abs/1611.09940. [Cited on page 102]

[31] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: Probabilistic model

for code. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2016. [Cited on page 85]

[32] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, pages 24–51, 2015. [Cited on page 92]

[33] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2003. [Cited on page 1]

[34] Dimitar Bounov, Anthony DeRossi, Massimiliano Menarini, William G. Griswold, and Sorin Lerner. Inferring loop invariants through gamification. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, 2018. [Cited on page 85]

[35] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2009. [Cited on pages 16 and 32]

[36] Marc Brockschmidt, Yuxin Chen, Pushmeet Kohli, Siddharth Krishna, and Daniel Tarlow. Learning shape analysis. In *Proceedings of the Static Analysis Symposium (SAS)*, 2017. [Cited on page 85]

[37] Lukas Bulwahn, Maryam Kamali, and Sven Linker, editors. *Proceedings First Workshop on Formal Verification of Autonomous Vehicles, FVAV@iFM 2017, Turin, Italy, 19th September 2017*, volume 257 of *EPTCS*, 2017. URL http://arxiv.org/abs/1709.02126. [Cited on page 1]

[38] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018. [Cited on page 85]

[39] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Method Symposium*. Springer, 2015. [Cited on page 1]

[40] Xinyun Chen, Chang Liu, and Dawn Song. Towards synthesizing complex programs from input-output examples. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018. [Cited on page 85]

[41] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014. [Cited on page 12]

[42] William W. Cohen. Pac-learning non-recursive prolog clauses. *Artificial Intelligence*, 79(1), 1995. [Cited on page 17]

[43] William W. Cohen and C. David Page. Polynomial learnability and inductive logic programming: Methods and results. *New Generation Comput.*, 13 (3&4):369–409, 1995. [Cited on page 65]

[44] Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. Linear invariant generation using non-linear constraint solving. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2003. [Cited on page 70]

[45] Byron Cook. Formal reasoning about the security of amazon web services. In *Proceedings of the International Conference on Computer Aided Verifica-*

tion (CAV), pages 38–47, 2018. doi: 10.1007/978-3-319-96145-3_3. URL https://doi.org/10.1007/978-3-319-96145-3_3. [Cited on page 1]

[46] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5), 2011. [Cited on page 86]

[47] Byron Cook, Kareem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Model checking boot code from AWS data centers. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 467–486, 2018. doi: 10.1007/978-3-319-96142-2_28. URL https://doi.org/10.1007/978-3-319-96142-2_28. [Cited on page 1]

[48] Sam Corbett-Davies and Sharad Goel. The measure and mismeasure of fairness: A critical review of fair machine learning. *CoRR*, abs/1808.00023, 2018. URL http://arxiv.org/abs/1808.00023. [Cited on page 103]

[49] Andrew Cropper, Alireza Tamaddoni-Nezhad, and Stephen H Muggleton. Meta-interpretive learning of data transformation programs. In *Proceedings of the 24th International Conference on Inductive Logic Programming*, 2015. [Cited on pages 57 and 64]

[50] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4), 1991. [Cited on page 74]

[51] Jacek Czerniak and Hubert Zarzycki. Artificial intelligence and security in computing systems. chapter Application of Rough Sets in the Presumptive Diagnosis of Urinary System Diseases. 2003. [Cited on page 56]

[52] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: contract-based modular verification of concurrent C. In

Proceedings of the International Conference on Software Engineering (ICSE), pages 429–430, 2009. [Cited on page 1]

[53] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2016. [Cited on pages 13 and 74]

[54] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. [Cited on pages 68, 78, 85, and 102]

[55] Hanjun Dai, Yingtao Tian, Bo Dai, Steven Skiena, and Le Song. Syntax-directed variational autoencoder for structured data. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018. [Cited on page 76]

[56] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2008. [Cited on pages 2, 68, 69, and 81]

[57] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, 2007. [Cited on pages 50 and 64]

[58] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*, pages 248–255. IEEE Computer Society, 2009. [Cited on page 101]

[59] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2017. [Cited on page 85]

[60] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2013. [Cited on pages 68, 70, and 80]

[61] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2015. [Cited on page 74]

[62] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. doi: 10.1007/978-3-540-24605-3_37. URL https://doi.org/10.1007/978-3-540-24605-3_37. [Cited on page 2]

[63] Varun Embar, Dhanya Sridhar, Golnoosh Farnadi, and Lise Getoor. Scalable structure learning for Probabilistic Soft Logic. *CoRR*, abs/1807.00973, 2018. [Cited on page 65]

[64] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data (Extended abstract). In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2018. [Cited on page 65]

[65] Manuel Fahndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-Oriented Software*, 2010. [Cited on page 67]

[66] Yoav Freund, H. Sebastian Seung, Eli Shamir, and Naftali Tishby. Selective sampling using the query by committee algorithm. *Machine Learning*, 28 (2-3), 1997. [Cited on page 44]

[67] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. Ice: A robust framework for learning invariants. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2014. [Cited on page 70]

[68] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2016. [Cited on pages 67, 69, 70, and 80]

[69] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. AI2: safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 3–18. IEEE Computer Society, 2018. doi: 10.1109/SP.2018.00058. URL https://doi.org/10.1109/SP.2018.00058. [Cited on page 103]

[70] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the International Conference on Machine Learning (ICML)*, page 1263–1272, 2017. [Cited on page 13]

[71] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In Yoshua Bengio and Yann LeCun, ed-

itors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6572. [Cited on page 103]

[72] Georg Gottlob, Christoph Koch, Robert Baumgartner, Marcus Herzog, and Sergio Flesca. The lixto data extraction project: Back and forth between theory and practice. In *PODS*, 2004. [Cited on page 16]

[73] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014. URL http://arxiv.org/abs/1410.5401. [Cited on page 13]

[74] S. Grebenshchikov, A. Gupta, N. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A software verifier based on Horn clauses. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2012. [Cited on page 16]

[75] Todd Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the 26th Symposium on Principles of Database Systems*, PODS, 2007. [Cited on pages 33, 49, and 51]

[76] Todd J. Green. Logiql: A declarative language for enterprise applications. In *PODS*, 2015. [Cited on page 20]

[77] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1828–1836, 2015. [Cited on page 13]

[78] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Symposium on Principles of Programming Languages*, POPL, 2011. [Cited on pages 2 and 32]

[79] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2011. [Cited on page 84]

[80] Sumit Gulwani and Nebojsa Jojic. Program verification as probabilistic inference. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2007. [Cited on page 70]

[81] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2011. [Cited on page 5]

[82] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin Zorn. Inductive programming meets the real world. *Communications of the ACM*, 58(11), October 2015. [Cited on page 2]

[83] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017. [Cited on page 2]

[84] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 343–361, 2015. [Cited on page 93]

[85] Isabelle Guyon, Masoud Nikravesh, Steve R. Gunn, and Lotfi A. Zadeh, editors. *Feature Extraction - Foundations and Applications*, volume 207 of *Studies in Fuzziness and Soft Computing*. Springer, 2006. ISBN 978-3-540-35487-1. doi: 10.1007/978-3-540-35488-8. URL https://doi.org/10.1007/978-3-540-35488-8. [Cited on page 2]

[86] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the Myria big data management service. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *SIGMOD*, 2014. doi: 10.1145/2588555.2594530. [Cited on page 16]

[87] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90. URL https://doi.org/10.1109/CVPR.2016.90. [Cited on page 2]

[88] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), October 1969. [Cited on page 69]

[89] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. [Cited on page 12]

[90] Kryštof Hoder, Nikolaj Bjørner, and Leonardo De Moura. μz–an efficient engine for fixed points with constraints. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2011. [Cited on page 54]

[91] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991. doi: 10.1016/0893-6080(91)90009-T. URL https://doi.org/10.1016/0893-6080(91)90009-T. [Cited on page 11]

[92] Ahmed Hosny, Chintan Parmar, John Quackenbush, Lawrence H Schwartz, and Hugo J W L Aerts. Artificial intelligence in radiology. *Nature reviews. Cancer*, 18(8):500–510, 08 2018. doi: 10.1038/s41568-018-0016-5. URL https://pubmed.ncbi.nlm.nih.gov/29777175. [Cited on pages 2 and 101]

[93] Feng-hsiung Hsu, Murray S. Campbell, and A. Joseph Hoane. Deep blue system overview. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, page 240–244, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917286. doi: 10.1145/224538.224567. URL https://doi.org/10.1145/224538.224567. [Cited on page 14]

[94] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kuncak, editors, *Proceedings of the International Conference on Computer Aided Verification (CAV)*, volume 10426, pages 97–117. Springer, 2017. [Cited on page 103]

[95] Elias Boutros Khalil, Pierre Le Bodic, Le Song, George L Nemhauser, and Bistra N Dilkina. Learning to branch in mixed integer programming. 2016. [Cited on page 85]

[96] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. An algebraic Prolog for reasoning about possible worlds. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011. [Cited on page 64]

[97] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. *J. of Applied Logic*, 22(C), July 2017. [Cited on page 64]

[98] Ross D. King. Applying inductive logic programming to predicting gene function. *AI Magazine*, 25(1), March 2004. [Cited on page 16]

[99] Stanley Kok and Pedro M. Domingos. Learning the structure of Markov Logic Networks. In *Machine Learning, Proceedings of the Twenty-Second International Conference*, 2005. [Cited on page 64]

[100] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. *Formal Methods in System Design*, 48(3): 175–205, 2016. [Cited on page 93]

[101] Vijay R. Konda and John N. Tsitsiklis. Actor-critic algorithms. In *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*, pages 1008–1014, 1999. [Cited on page 79]

[102] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1106–1114, 2012. [Cited on page 2]

[103] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *CoRR*, abs/1607.02533, 2016. URL http://arxiv.org/abs/1607.02533. [Cited on page 103]

[104] Matt J Kusner, Brooks Paige, and José Miguel Hernández-Lobato. Grammar variational autoencoder. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2017. [Cited on page 76]

[105] Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 527–534, 2000. ISBN 1-55860-707-2. [Cited on page 2]

[106] Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014. [Cited on page 2]

[107] Rustan Leino. Dafny: An automatic program verifier for functional correctness. In *16th International Conference, LPAR-16, Dakar, Senegal*, pages 348–370, April 2010. URL https://www.microsoft.com/en-us/research/publication/

dafny-automatic-program-verifier-functional-correctness-2/. [Cited on page 1]

[108] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52 (7):107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814. [Cited on page 1]

[109] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015. [Cited on page 13]

[110] Chen Liang, Jonathan Berant, Quoc Le, Kenneth D Forbus, and Ni Lao. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. *arXiv preprint arXiv:1611.00020*, 2016. [Cited on page 85]

[111] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Exponential recency weighted average branching heuristic for SAT solvers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3434–3440, 2016. [Cited on page 102]

[112] Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical Bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning*, 2010. [Cited on page 66]

[113] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David Gay, Joseph Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: Language, execution and optimization. In *Proceedings of the 2006 International Conference on Management of Data*, SIGMOD, 2006. [Cited on page 32]

[114] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion

Stoica. Declarative networking. *Communications of the ACM*, 52(11), November 2009. [Cited on page 16]

[115] C.J. Maddison and D. Tarlow. Structured generative models of natural source code. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2014. [Cited on page 85]

[116] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. From datalog to flix: A declarative language for fixed points on lattices. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, PLDI '16, pages 194–208, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908096. URL http://doi.acm.org/10.1145/2908080.2908096. [Cited on page 16]

[117] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. A user-guided approach to program analysis. In *Proceedings of the ACM Symposium on Foundations of Software Engineering (FSE)*, 2015. [Cited on page 16]

[118] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018. [Cited on page 64]

[119] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. In *Communications of the ACM*, 1971. [Cited on page 84]

[120] Scott Mayer McKinney, Marcin Sieniek, Varun Godbole, Jonathan Godwin, Natasha Antropova, Hutan Ashrafian, Trevor Back, Mary Chesus, Greg C. Corrado, Ara Darzi, Mozziyar Etemadi, Florencia Garcia-Vicente, Fiona J. Gilbert, Mark Halling-Brown, Demis Hassabis, Sunny Jansen, Alan Karthikesalingam, Christopher J. Kelly, Dominic King, Joseph R. Ledsam, David Melnick, Hormuz Mostofi, Lily Peng, Joshua Jay Reicher, Bernardino Romera-

Paredes, Richard Sidebottom, Mustafa Suleyman, Daniel Tse, Kenneth C. Young, Jeffrey De Fauw, and Shravya Shetty. International evaluation of an ai system for breast cancer screening. *Nature*, 577(7788):89–94, 2020. doi: 10.1038/s41586-019-1799-6. URL https://doi.org/10.1038/s41586-019-1799-6. [Cited on page 2]

[121] Kenneth L McMillan and Andrey Rybalchenko. Solving constrained horn clauses using interpolation. *Tech. Rep. MSR-TR-2013-6*, 2013. [Cited on page 92]

[122] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2002. [Cited on pages 55 and 56]

[123] Alexander Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. Key-value memory networks for directly reading documents. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2016. [Cited on page 74]

[124] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. [Cited on page 14]

[125] Raymond J. Mooney. Inductive logic programming for natural language processing. In *Inductive Logic Programming: Selected papers from the 6th International Workshop*. Springer Verlag, 1996. [Cited on page 16]

[126] Stephen Muggleton. Inverse entailment and Progol. *New generation computing*, 13(3-4), 1995. [Cited on pages 17, 32, 55, and 56]

[127] Stephen Muggleton. Stochastic logic programs. In *New Generation Computing*. Academic Press, 1996. [Cited on page 65]

[128] Stephen H Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1), 2015. [Cited on pages 18, 19, 33, 36, 38, 54, 55, 56, 58, and 65]

[129] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018. [Cited on page 85]

[130] Mayur Naik. Chord: A program analysis platform for Java. http://jchord.googlecode.com/, 2011. [Cited on page 1]

[131] Anh Tuan Nguyen and Tien N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015. [Cited on page 85]

[132] OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. 2019. URL https://arxiv.org/abs/1912.06680. [Cited on page 14]

[133] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2016. [Cited on pages 69, 70, and 80]

[134] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016. [Cited on pages 76 and 85]

[135] Gordon D Plotkin. A note on inductive generalization. *Machine intelligence*, 5(1), 1970. [Cited on page 41]

[136] Oleksander Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2015. [Cited on page 2]

[137] David Poole. Logic programming for robot control. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'95)*, 1995. [Cited on pages 16 and 32]

[138] Rachel Potvin and Josh Levenberg. Why google stores billions of lines of code in a single repository. *Communications of the ACM*, 59:78–87, 2016. URL http://dl.acm.org/citation.cfm?id=2854146. [Cited on page 1]

[139] J. Ross Quinlan and R. Mike Cameron-Jones. Foil: A midterm report. In *Proceedings of the European Conference on Machine Learning (ECML'93)*, 1993. [Cited on page 17]

[140] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. User-guided program reasoning using Bayesian inference. In *Proceedings*

of the ACM Conference on Programming Language Design and Implementation (PLDI), 2018. [Cited on page 16]

[141] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: correctness checking for real code. In *Proceedings of the USENIX Security Symposium*, 2015. [Cited on pages 26 and 27]

[142] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2015. [Cited on pages 2 and 98]

[143] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*, 2002. [Cited on page 86]

[144] Matthew Richardson and Pedro Domingos. Markov Logic Networks. *Machine Learning*, 62(1-2), 2006. [Cited on page 64]

[145] Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. [Cited on page 65]

[146] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003. ISBN 0137903952. [Cited on page 18]

[147] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at Google. *Communications of the ACM*, 61(4), March 2018. [Cited on page 1]

[148] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using Gröbner bases. In *Proceedings of the ACM*

Symposium on Principles of Programming Languages (POPL), 2004. [Cited on page 70]

[149] JCA Santos, H Nassif, D Page, SH Muggleton, and MJE Sternberg. Automated identification of protein-ligand interaction features using inductive logic programming: a hexose binding case study. *BMC BIOINFORMATICS*, 13, 2012. [Cited on page 16]

[150] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. [Cited on page 74]

[151] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013. ISBN 978-1-4503-1870-9. [Cited on pages 5 and 84]

[152] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic program optimization. *Communications of the ACM*, 59(2), 2016. [Cited on page 66]

[153] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the International Conference on Compiler Construction (CC)*, CC 2016, 2016. [Cited on pages 16 and 20]

[154] Daniel Selsam, Matthew Lamm, Benedikt Bunz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018. [Cited on page 85]

[155] Andrew W. Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander W. R. Nelson, Alex Bridgland, Hugo Penedones, Stig Petersen, Karen Simonyan, Steve

Crossan, Pushmeet Kohli, David T. Jones, David Silver, Koray Kavukcuoglu, and Demis Hassabis. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020. doi: 10.1038/s41586-019-1923-7. URL https://doi.org/10.1038/s41586-019-1923-7. [Cited on page 2]

[156] Jiwon Seo. Datalog extensions for bioinformatic data analysis. In *40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, EMBC, 2018. [Cited on page 32]

[157] Jiwon Seo, Stephen Guo, and Monica S Lam. SociaLite: Datalog extensions for efficient social network analysis. In *ICDE*, 2013. [Cited on page 16]

[158] Burr Settles. Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1):1–114, 2012. [Cited on page 44]

[159] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *Proceedings of the 5th Annual Workshop on Computational Learning Theory (COLT'92)*, 1992. [Cited on page 44]

[160] Ehud Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, 1983. [Cited on page 64]

[161] Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2014. [Cited on pages 69, 70, and 80]

[162] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2011. [Cited on page 70]

[163] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A data driven approach for algebraic loop invariants. In *Proceedings of the European Symposium on Programming (ESOP)*, 2013. [Cited on page 70]

[164] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2016. [Cited on pages 16 and 32]

[165] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999. doi: 10.1109/12.769433. URL https://doi.org/10.1109/12.769433. [Cited on page 2]

[166] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. doi: 10.1038/nature16961. URL https://doi.org/10.1038/nature16961. [Cited on pages 2 and 14]

[167] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017. [Cited on pages 2 and 78]

[168] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran,

Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018. ISSN 0036-8075. doi: 10.1126/science.aar6404. URL https://science.sciencemag.org/content/362/6419/1140. [Cited on page 2]

[169] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2013. [Cited on page 64]

[170] Y. Smaragdakis and M. Bravenboer. Using Datalog for fast and easy program analysis. In *Datalog 2.0 Workshop*, 2010. [Cited on page 1]

[171] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2011. [Cited on pages 55 and 56]

[172] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Combinatorial sketching for finite programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006. [Cited on pages 2, 3, 5, and 64]

[173] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer (STTT)*, 2013. [Cited on page 64]

[174] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. End-to-end memory networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2015. [Cited on pages 13 and 74]

[175] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998. ISBN 978-0-262-19398-6. [Cited on pages 15 and 90]

[176] SyGuS Competition, 2017. http://sygus.seas.upenn.edu/SyGuS-COMP2017.html. [Cited on pages 80 and 97]

[177] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the Association for Computational Linguistics (ACL)*, 2015. [Cited on pages 12 and 77]

[178] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. TRANSIT: Specifying protocols with concolic snippets. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2013. [Cited on page 5]

[179] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019. doi: 10.1038/s41586-019-1724-z. URL https://doi.org/10.1038/s41586-019-1724-z. [Cited on page 14]

[180] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly

expressive SQL queries from input-output examples. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2017. [Cited on pages 21, 55, and 57]

[181] William Yang Wang, Kathryn Mazaitis, and William Cohen. Structure learning via parameter learning. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, 2014. [Cited on page 65]

[182] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. In Yoshua Bengio and Yann LeCun, editors, *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015. [Cited on page 13]

[183] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2004. [Cited on pages 55 and 56]

[184] J. Whaley, D. Avots, M. Carbin, and M. Lam. Using Datalog with binary decision diagrams for program analysis. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS'05)*, 2005. [Cited on page 1]

[185] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated bug removal for software-defined networks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017. [Cited on page 16]

[186] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens,

George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016. URL http://arxiv.org/abs/1609.08144. [Cited on page 2]

[187] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019. [Cited on page 13]

[188] Fan Yang, Zhilin Yang, and William W. Cohen. Differentiable learning of logical rules for knowledge base reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. [Cited on page 65]

[189] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018. [Cited on page 13]

[190] Alan L. Yuille, Peter W. Hallinan, and David S. Cohen. Feature extraction from faces using deformable templates. *Int. J. Comput. Vis.*, 8(2):99–111, 1992. doi: 10.1007/BF00127169. URL https://doi.org/10.1007/BF00127169. [Cited on page 2]

[191] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. Apisan: Sanitizing api usages through semantic cross-checking. In *Proceedings of the USENIX Security Symposium*, 2016. [Cited on pages 55 and 56]

[192] Lintao Zhang. *Searching for Truth: Techniques for Satisfiability of Boolean Formulas*. PhD thesis, Princeton University, 2003. [Cited on page 102]

[193] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in datalog. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014. [Cited on page 16]

[194] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. Effective interactive resolution of static analysis alarms. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2017. [Cited on page 16]

[195] He Zhu, Stephen Magill, and Suresh Jagannathan. A data-driven CHC solver. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2018. [Cited on page 93]