Neural-guidance for symbolic reasoning

by

Nham Le

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2020

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Some of the text, figures and tables in this thesis are restated from our NeurIPS workshop paper [53] and our current submission.

Occam was written by the researchers at The Computer Science Laboratory at SRI International.

Some of the source code for Spacer was written by Dr. Arie Gurfinkel.

Abstract

Symbolic reasoning begot Artificial Intelligence (AI). With the recent advances in Deep Learning, many traditional AI areas such as Computer Vision and Natural Language Processing have moved to probabilistic-based approaches. However, in applications where there is little to no room for uncertainty, such as Compiler or Software verification, symbolic reasoning is still the go-to option. In this thesis, we bring the advantage of data-driven learnable models into the precise world of symbolic reasoning. In particular, we choose to tackle two specific problems: Model Checking, in the context of Inductive Generalization, and Compiler Optimization, in the context of Software Debloating. We implemented our approach in two tools, named Dopey and DeepOccam, respectively. They both use traces generated from running a task to learn a better heuristic, and use said heuristic to improve subsequent runs of the same or similar tasks. Our results show that both neural-based heuristics outperform handcrafted heuristics.

Acknowledgements

I would like to thank Professor Arie Gurfinkel for sharing his expertise in the subject, his availability, and giving valuable advice for this research. It goes without saying that this document would not be possible without him.

I would like to thank Professor Werner Dietl and Professor Vijay Ganesh for attending the seminar, reviewing the thesis and providing valuable feedback on both occasions.

I would like to thank the people at SRI International, especially Dr. Ashish Gehani and Dr. Jorge Navas, for the wonderful time I had there.

Finally, I would like to thank Dr. Xujie Sie for his deep insight into the world of neural-symbolic. This thesis would be nowhere near its current form without him.

Dedication

This is dedicated to the one I love.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Symbolic reasoning predates Computer Science. The word *algorithm* itself came from the 9th-century mathematician Muhammad ibn Musa al-Khwarizmi, Latinized Algoritmi. Logics itself could be traced back to Aristotle in the 300s BC. It is fair to say that the whole field of Computer Science was born out of symbolic reasoning, with pioneering work such as Hilbert's *Entscheidungsproblem*, Alonzo Church's *Lambda Calculus*, and Alan Turing's *Turing machine*. A century has passed since the birth of modern Computer Science, and nowadays the field is diverse in the use of mathematics: Topology and Geometry in Computer Graphics, Probability and Linear Algebra in Deep learning, among others. Nowadays, symbolic reasoning could still be found in Computer Science in its pure form under the research of Programming Languages, Compilers, Formal Methods, and Automated Reasoning.

Throughout its history, at the heart of all the symbolic reasoning applications are carefully handcrafted heuristics, tried-and-true by years of human experts' research. While those heuristics work wonderfully, they come with the cost of being too specific to the problem at hand. For example, there is no easy way to transfer all the wisdom learned in crafting a SAT-solver heuristic or the heuristic itself to create a better heuristic for a CHC-solver. This raises a natural, practical, and to a certain extend, a philosophical question: can a heuristic for a symbolic reasoning system be learned automatically?

Just ten years ago, only the most ambitious researchers would answer "yes" for the above question. While glimpses of learnable heuristics could already be found in groundbreaking work such as TD-Gammon [82], the whole research direction was practically halted due to hardware limitation. Then, at the turn of the 2010s, researchers realized that the heaviest workload in Deep Learning — matrix multiplication — could be done

very efficiently using GPU — an easy to find and relatively cheap hardware component. AlexNet [49] — one of the first works that demonstrated the scalability of Deep Neural Networks using GPUs - blew every other image recognition methods at the time out of the water, and overnight, old ideas were new again: Convolution Neural Networks, Long-Short Term Memory (LSTM) Networks, Deep Reinforcement Learning, etc. are all in the realm of possibility. Nowadays, it is hard to find a field that is not yet "transformed" or "revolutionized" by Deep learning.

Yet, symbolic reasoning is still one of the less successful applications of Deep learning: while Deep learning has been able to achieve human-level, or even superhuman-level on many tasks, such as Image Recognition, Automated Speech Recognition, Game Playing (Go, Atari), it still fails short of the state-of-the-art heuristics at many symbolic reasoning tasks, such as solving mathematical equations [51] or solving SAT problems [71]. Ironically, symbolic reasoning was also the reason for the first **AI Winter**: Marvin Minsky, founder of the MIT AI Lab, and Seymour Papert, director of the lab at the time, in 1969 published the seminal book Perceptrons [59], that discussed perceptron's inability to learn the simple boolean function XOR because it is not linearly separable, turning research away from the perceptron. The history of symbolic reasoning and Deep Learning, as often said, comes full circle.

We hypothesize that it is possible to learn Deep Learning-based heuristics those are better than handcrafted one. This thesis offers a glimpse into this possibility, by presenting two concrete positive results where effective heuristics are indeed learnable from data, in two particular domains: compiler optimization, and automated reasoning. We choose to tackle those two domains because we believe the handcrafted heuristics there are not optimal and there is still a lot of room for improvement, and as shown in the following chapters, we indeed improved them by a considerable margin.

1.1 Symbolic Model Checking using anti– and co– occurrence probabilities

In the automated reasoning domain, we introduce Dopey, a neural-based Symbolic Model Checker (SMC) / Constraint Horn Clauses (CHC) Solver.

Model checking has been widely used in various important areas such as robustness analysis of deep neural networks [44], verification of hardware designs [25], software verification [9], analysis [32] and testing [74], parameter synthesis in biology [11], and many

others. The central challenge of model checking is to find a concise and sound approximation of all possible states a given system may reach, which does not cover any undesired states (i.e. violating given specifications). Tremendous progresses have been made by innovations in efficient data representations [18], scalable SAT solvers [76, 60, 30], and effective heuristics [24, 23, 56]. Modern model checkers share a common basis, namely, IC3 [15], of which the key insight is *inductive generalization*. This idea has been generalized to support rich theories [39] that are crucial for many verification tasks [47, 34] beyond hardware verification. The generalized IC3 with rich theories, also known as satisfiability checking for Constrained Horn Clauses modulo Theory (CHC) [14], becomes the core part of a broad range of verification tasks.

Existing inductive generalization techniques follow either an enumerative search process [15, 16] or ad-hoc heuristics [33, 48]. Heuristics are effective but may demand non-trivial domain-specific (or even problem-specific) expertise. In this work, we aim to automatically learn such heuristics from the past successful inductive generalizations. We observe that verification problems as well as associated inductive generalizations are not isolated from each other. Taking software verification as an example, verifying different properties of the same program involves similar or same inductive generalizations; different versions of programs have similar code base; and different software may use same coding convention, idioms, library or framework, resulting in similar structures.

A natural solution is to train a deep learning model to directly perform inductive generalization, but this approach also raises many new challenges for deep learning. First of all, the input and the output of inductive generalization are symbolic expressions, which are *highly structured* with *rich semantics*. Slight syntactic variations can lead to dramatic changes in semantics. Second, more importantly, inductive generalization has to satisfy complicated *semantic constraints*. Third, given deep learning models hardly provide any reliable guarantees, how to design a neuro-symbolic system that exhibits *learnability* from past experiences but still preserves *soundness*? All these challenges have to be properly addressed in building a neuro-symbolic reasoning framework.

In Chapter 3, we present a neuro-symbolic engine Dopey, which introduces a neural component into symbolic model checking. Specifically, we make the following contributions:

- we adapt standard deep learning models to effectively represent symbolic expressions by incorporating both syntactic and semantic information;
- we design a simple but effective learning objective so that training data can be collected with nearly no changes of existing model checkers;

- our integration algorithm achieves the soundness by design, and in the worst case, the learning component may only hurt the running time performance;
- we implement Dopey on top of Spacer, a state-of-the-art CHC-solver, using an efficient client-server architecture;
- our empirical evaluations indicate Dopey significantly improves Spacer on challenging benchmarks from CHC-COMP 2018.

1.2 Compiler Optimization using Reinforcement Learning

In the compiler optimization domain, we introduce DeepOccam, a reinforcement-learning–based software debloating tool.

A *Software Debloater* solves a specific problem: given the known running environment, can we remove as much unused code (*debloat*) in the compiled executables and binaries as possible? A motivating example is the frequently used tool `ls`. `ls` has more than 40 arguments, yet for many people, `ls -lath` covers most of the use-cases. By removing code of the rest of `ls`'s arguments, a software debloater hopes to create a smaller, and safer `ls`.

One successful approach for software debloating is based on partial evaluation (PE) [42] in which a *partial evaluator* takes a program and its known input values and produces a *residual* (or *specialized*) program in which those inputs are replaced with constants. While PE has been extensively applied to functional and logic programs, it was less successful on imperative C/C++ programs (with a noteable exception of C-Mix [7] and Tempo [26]). With the advent of LLVM [52], several new partial evaluators of LLVM bitcode have arisen during the last few years (e.g., LLPE [77], Occam [54], and Trimmer [73]). These tools leverage LLVM optimizations such as constant propagation, function inlining, and others to either reduce the size of the residual program (e.g., Occam and Trimmer) or improve performance (e.g., LLPE).

The key step of a PE-based debloater is to estimate whether specializing (or inlining) a function can result in a smaller (or more secure) residual program. Specialization naturally increases the size of the code since it adds extra functions. However, since some inputs in the new functions have been replaced by constant values, optimizations such as constant propagation, might become enabled and result in new optimization opportunities that reduce the code size. Therefore, the decision whether or not a function should be specialized for a particular call-site is non-trivial.

Interestingly, this decision of whether a call-site should be specialized or not resembles a long-horizon Markov Decision Process, which fits nicely into the Reinforcement Learning framework: each decision moves the code (state) from one to another, the decision should depend only on the current state instead of the whole history of transformation, and the quality of the whole process (reward) is only known at the end.

To realize this insight, we need to solve the challenge of deciding on a good state representation. While it is tempting to use the source code (or LLVM intermediate representation (IR)) as a state, this is not computationally tractable. The IR is typically hundreds of MBs in size. Instead, an adequate set of features that captures meaningful information about the code while avoiding state aliasing is required. In particular, these features must capture the *calling context* in order to distinguish between call-sites.

In Chapter 4, we present a software debloater DeepOccam, which learns a heuristic for specialization using Reinforcement Learning. Specifically, we make the following contributions:

- we propose a set of Calling context features that enable RL to find a useful heuristic for PE-based debloating software;
- we implement our method in a software debloater called DeepOccam;
- we evaluate on the reduction of the program size and number of possible code-reuse attacks, by comparing our handcrafted features with features learned automatically via embedding LLVM IR into a vector space using inst2vec [13].

1.3 Organization of the thesis

This thesis proposes two neural-guided heuristics for software debloating — a compiler optimization task, and inductive generalization — a model checking algorithm's components. The key novelty is in automatically learning useful signals from a discrete logic world using a differientable framework. The rest of the thesis is structured as follows:

In Chapter 2, we introduce needed background on our domains, which are Symbolic Model Checking and Software Debloating, our learning paradigms, which are Deep learning and Reinforcement learning.

In Chapter 3, we present a novel neural-symbolic SMC, called Dopey— inspired by the learning of co-occurrence probabilities in NLP. Dopey learns offline dependence between

atoms that do (or do not) appear together in facts (called *lemmas*) that are learned by the SMC. The neural net is then used in successive runs of the SMC on different properties of the same system to guide the inductive generalization heuristic. Thus, in a multi-property setting, learning from one verification task generalizes to others.

In Chapter 4, we present a neural-guided software debloating tool, called DeepOccam. DeepOccam uses Reinforcement learning to learn a heuristic for when to or not to apply *specialization* – a particular compiler optimization in which known fixed inputs/flags (e.g a web server that is known to always run using HTTP 2.0) are replaced with their values. The metrics that DeepOccam tries to optimize are the number of ROP, COP, and JOP gadgets in the compiled binaries.

Finally, we outline a number of future research directions in Chapter 5, and conclude in Chapter 6.

Chapter 2

Background

In this chapter, we present preliminaries needed for the rest of the thesis.

2.1 Symbolic model checking

In this section, we present the SAFETY problem and the state-of-the-art algorithm IC3/SPACER. They are needed for understanding Chapter 3.

The Safety problem. We consider first order logic modulo theories, using the standard notation and terminology. A first-order language modulo theory $\mathcal{T}$ is defined over a signature Σ containing constant, function, and predicate symbols.

A *transition system* is a tuple $\langle \Sigma, \mathit{Init}, \mathit{Tr} \rangle$. Σ, Σ', and Σ^i are used to present the pre-state, the post-state, and the state of the system after executing i steps, respectively ($\Sigma' = \{v' \mid v \in \Sigma\}$, $\Sigma^i = \{v^i \mid v \in \Sigma\}$). Init is a formula over Σ and Tr is a formula over $\Sigma \cup \Sigma'$. For a formula φ over variables in Σ, we denote by φ' the formula obtained by substituting each $v \in \varphi$ by $v' \in \Sigma'$, and φ^i the formula obtained by substituting each $v \in \varphi$ by $v^i \in \Sigma^i$. We also denote Tr^i a formula obtained by substituting each $v \in \mathit{Tr}$ by $v^i \in \Sigma^i$ and each $v' \in \mathit{Tr}'$ by $v^{i+1} \in \Sigma^{i+1}$.

For simplicity, we omit Σ and use the shorthand $\langle \mathit{Init}, \mathit{Tr} \rangle$ to represent the transition system whenever the context is clear.

A SAFETY *problem* is a triple $P = \langle \mathit{Init}, \mathit{Tr}, \mathit{Bad} \rangle$, where $\langle \mathit{Init}, \mathit{Tr} \rangle$ is a transition system and Bad is a formula over Σ representing a set of bad states. P is UNSAFE if and

only if there exists a number N such that the following is satisfiable:

$$Init^0 \wedge (\bigwedge_{i=0}^{N-1} Tr^i) \wedge Bad^N \tag{2.1}$$

In this case, we also say that P has a *counterexample (CEX)* of length N. Vice versa, P is SAFE if and only if Eq. (2.1) is unsatisfiable.

The SAFETY problem defined above is an instance of a more general problem, CHC-SAT, of satisfiability of Constrained Horn Clauses (CHC). With abuse of notation, in this thesis we use *solving CHCs* and *verifying safety properties* interchangeably.

Induction, safe inductive invariant, relative induction, inductive trace. A formula φ is called an *inductive invariant* of a transition system $\langle Init, Tr\rangle$ if and only if:

$$Init \implies \varphi \tag{2.2}$$
$$\varphi \wedge Tr \implies \varphi' \tag{2.3}$$

We also say that a formula φ is *inductive relative* to a formula F if it satisfies initiation and $\varphi \wedge F \wedge Tr \implies \varphi'$.

An inductive invariant φ is SAFE if:

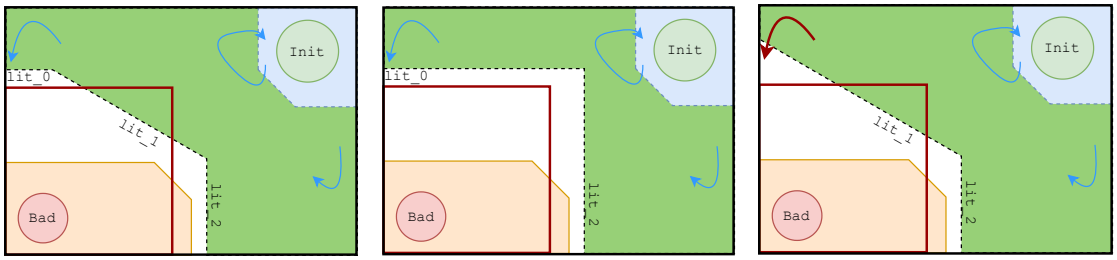
$$\varphi \implies \neg Bad \tag{2.4}$$

An *inductive trace* of a transition system is a list of formulas $F = [F_0, F_1, \ldots, F_N]$ such that

$$Init \implies F_0 \tag{2.5}$$
$$\forall 0 \leq i < N, F_i \wedge Tr \implies F_{i+1} \tag{2.6}$$

We call F_i a *frame*, and we represent each frame F_i as a set of *lemmas*, and each lemma $\ell \in F_i$ is a clause.

Craig Interpolation. Given an unsatisfiable formula $A \wedge B$, a *Craig interpolant*, denoted $\mathrm{ITP}(A, B)$, is a formula I over the shared signature of A and B such that $A \Rightarrow I$ and $I \Rightarrow \neg B$.



(a) An inductive lemma L = {`lit_0`, `lit_1`, `lit_2`}

(b) L is successfully generalized by dropping `lit_1`

(c) L no longer inductive by dropping `lit_0`

Figure 2.1: Visualization of inductive generalization.

IC3/Spacer. The current state-of-the-art algorithm to solve CHC is IC3/Spacer [46]. At the high level, the main IC3/Spacer loop tries to find a CEX of length N, and terminates if the CEX is found (unsafe), or the lemma that proves such CEX doesn't exist is also a safe inductive invariant (safe).

There are many ways to represent the Spacer algorithm, and in this thesis we borrows the presentation used in [48].

Fig. 2.2 presents the key ingredients of Spacer as a set of rules. It maintains the following:

- The current unrolling depth N at which a counterexample is searched (there are no counterexamples with depth less than N).
- An inductive trace $F = [F_0, F_1, \ldots]$. Intuitively, each frame F_i is a candidate inductive invariant s.t. F_i over-approximates states reachable up to i steps from *Init*.
- A queue of *proof obligations* Q, where each proof obligation (pob) in Q is a pair $\langle \varphi, i \rangle$ of a cube φ and a level number i, $0 \leq i \leq N$. Each pob $\langle \varphi, i \rangle$ in Q corresponds to a suffix of a potential counterexample that has to be blocked in F_i, i.e., has to be proven unreachable in i steps.
- An under-approximation $\mathcal{U}$ of reachable states, represented as a disjunction.

The `Candidate` rule adds a pob $\langle Bad, N \rangle$ to the queue. If a pob $\langle \varphi, i \rangle$ cannot be blocked because φ is reachable from frame $(i-1)$, `Predecessor` generates a predecessor ψ of φ using GetPredecessor, and add $\langle \psi, i-1 \rangle$ to Q. `Successor` updates the set of reachable states if the pob is reachable. If the pob is blocked, `Conflict` strengthens

the trace F by using interpolation to learn a new lemma ℓ that blocks the POB, i.e., $\ell \implies \neg\varphi$. `Induction` strengthens a lemma by inductive generalization and `Propagate` pushes a lemma to a higher frame. If the *Bad* state has been blocked at N, `Unfold` increments the depth of unrolling N.

Inductive generalization The `Induction` rule is a crucial optimization to IC3. To block a POB φ, it is enough to learn the clause $\neg\varphi$. However, most of the time, this clause is too weak to block any other POB. As illustrated in Fig. 2.1a, to block the orange POB, we can learn the lemma $\{\texttt{lit\_0}, \texttt{lit\_1}, \texttt{lit\_2}\}$ (slightly shifted because of the interpolation). This lemma is too weak, and cannot be used to block the red-lined POB. Since there could be exponentially many POBs, it is of the utmost importance that a learned lemma could be reused to block multiple POBs. One way to do so is to try to drop literals in the learned lemma, and check that the reduced lemma is still inductive relative to the previous frame. As we can see from Fig. 2.1c, starting from the lemma $\{\texttt{lit\_0}, \texttt{lit\_1}, \texttt{lit\_2}\}$, dropping `lit_0` results in a lemma that is no longer inductive, while dropping `lit_1` doesn't affect its inductiveness, as illustrated in Fig. 2.1b.

2.2 Software debloating

This section presents needed domain background for Chapter 4, which is software debloating using partial evaluation.

The software debloating problem. Given a problem $\mathcal{P}$ that has a set of functionalities $F = \{F_0, F_1, \ldots, F_N\}$ and a user-specified subset of necessary functionalities $F_{spec} = \{F_i, F_j, \ldots, F_k\}$, the goal of a software debloater is to produce a new program $\mathcal{P}'$ that retains F_{spec} and *gracefully* refuses any functionalities in $F \setminus F_{spec}$.

The Python server in Fig. 2.3 has two modes: a fallback mode that doesn't require a trained neural network, and a normal mode that requires one. Given the specification in Fig. 2.4, in which user requires only the first functionality, one possible debloated version of the server is given in Fig. 2.5. Note that even though line 15 to 20 in Fig. 2.3 are still in Fig. 2.5, it cannot be triggered, because the `main` function is not called. Remember that according to our problem definition, a debloated program is only about reducing functionalities, and the code itself could actually be more "bloated".

There are many ways to achieve the debloating goal, such as source-code optimization [37], binary optimization [73], or compile-time optimization [54]. In this thesis, our fo-

cus is compile-time optimization, inspired by OCCAM [54]. Specifically, OCCAM's partial evaluator.

Partial evaluator. A *partial evaluator* takes a program and some of its input values and produces a *specialized* program in which those inputs are replaced with their values. In Fig. 2.5, `main_specialized` is that specialized version of `main`.

While PE has been extensively applied to functional and logic programs, it was less successful on imperative C/C++ programs (with a notable exception of C-MIX [7] and TEMPO [26]). With the advent of LLVM [52], several new partial evaluators of LLVM bitcode have arisen during the last few years (e.g., LLPE [77], OCCAM [54], and TRIMMER [73]). These tools leverage LLVM optimizations such as constant propagation, function inlining, and others to either reduce the size of the residual program (e.g., OCCAM and TRIMMER) or improve performance (e.g., LLPE).

The key step of a PE-based debloater is to estimate whether specializing a function can result in a smaller (or more secure) final binary or library. As we can see in Fig. 2.5, specialization naturally increases the size of the code since it adds extra copies of functions into the code. However, since some inputs in the new functions have been replaced by constant values, optimizations such as constant propagation, might become enabled and result in new optimization opportunities in subsequent optimization passes. Therefore, the decision whether or not a function should be specialized for a particular call-site is non-trivial.

2.3 Deep Learning

This section introduces deep neural networks, recurrent neural networks, and reinforcement learning. They are the machine learning tools that we use in Chapter 3 and Chapter 4.

2.3.1 Perceptron. Multi-layer Perceptron. The fixed-size computing paradigm.

There are already too many technical texts about the mathematical behind Perceptron and Multi-layer Perceptron. In this section, we take a pictorial look at it, starting from matrix multiplication. Multiplying two matrices of size $m \times n$ and $n \times p$ could be visualized as in Fig. 2.6a, which resulted in a new matrix of size $m \times p$.

Perceptron is defined as a function that maps its input $\boldsymbol{x}$ (a real-valued vector) to an output value y (a single binary value). While there are many ways to define the mapping, in practice the perceptron function is

$$y = \begin{cases} 1 & \text{if } \boldsymbol{x} \times \boldsymbol{W} + \text{bias} > 0 \\ 0 & \text{if otherwise} \end{cases}$$

Fig. 2.6b visualizes the perceptron function. In case of having to classify k classes ($k > 2$), a one-vs-all scheme is used: learn k separated perceptron, each perceptron p_i predicts whether (x) is of the class i^{th}.

In the early 60s, it is believed that Single-layer perceptron was all we needed to learn (approximate) any function. In 1969, Minsky and Papert [59] proved that a single perceptron cannot learn the simple XOR function, since XOR is not *linear separable*. There work effectively froze AI research for about ten years.

The solution for the XOR function, as it turned out, was very simple: Multi-layer Perceptrons.

Multi-layer Perceptron. To understand Multi-layer Perceptron, let's take a step back and look at what a Single-layer Perceptron does: it computes a linear transformation of the input ($\boldsymbol{x} \times \boldsymbol{W} + \text{bias}$), then applies a non-linear filter (activation function) over the results (a branching function). The key insight is that the result is also a matrix, hence can be feed into another perceptron, which in total, make a non-linear function! In fact, it is proven [41] that under certain assumptions, 2 layer perceptrons is enough to approximate arbitrary functions!

Fig. 2.6c visualizes a multi-layer perceptron. Looking at the figure, it is also clear why it is also called *Deep Learning*. One way to think about Deep Learning: given a dataset $\mathcal{D}$ of pairs $(\boldsymbol{x}_i, y_i)$, what is a good architecture (the arrangement of the intermediate matrices), and what are the values for entries in the matrices?

The fixed-size computing paradigm. Multi-layer perceptrons are great, and could be used to approximate many functions. But it also forces things to be in the same shape: once the architecture is fixed, all the matrices in between are also fixed in shape, so all inputs have to have the same shape, and all outputs also have to have the same shape as well. For many cases, this is not a big problem: image could be resized into a fixed size, missing values in an input vector could always be set to a dummy values, for examples. But in some cases, the whole paradigm just doesn't work, and requires a different paradigm altogether.

2.3.2 Recurrent-neural network. TreeLSTM.

In many tasks, we need to process inputs of arbitrary length, to predict outputs also of arbitrary length (often called *sequence-to-sequence* problems). There are multiple ways to make sequential inputs to work in the fixed-size paradigm, such as *padding* (add dummy values to the inputs to make sure they are all of the same length), *same length batching* (sort inputs by size and process inputs of the same size together), among others. However, those approaches doesn't help with handling sequential outputs. A better approach is *recurrent neural networks* (RNNs). An RNN defines a function

$$\boldsymbol{h}^t = f(\boldsymbol{h}^{t-1}, \boldsymbol{x}^t) \tag{2.7}$$

in which f is our linear transformation, followed by a non-linear filter. A typical RNN can be

$$f(\boldsymbol{h}^{t-1}, \boldsymbol{x}^t) = tanh(\boldsymbol{x}_t \times \boldsymbol{W} + \boldsymbol{h}_{t-1} \times \boldsymbol{U}) \tag{2.8}$$

This simple RNN is visualized in Fig. 2.6d. Note that $\boldsymbol{W}$ and $\boldsymbol{U}$ are shared. This also means that the longer the chain, the easier it is for the first input to vanish/explode in the computation of the current input, since $\boldsymbol{h}_t$ has the term $\boldsymbol{x}_0 \times \boldsymbol{W}^t$ in it. In practice, vanilla RNNs rarely can learn for chains longer than 10 time steps.

To address this issue, a forget/gating mechanism is added to RNN, most notably in architecture like LSTM [38] and GRU [21]. Equations to compute the output (update) at the time step t for LSTM with a forget gate are:

$$\boldsymbol{f}_t = \sigma(\boldsymbol{x}_t \times \boldsymbol{W}_f + \boldsymbol{h}_{t-1} \times \boldsymbol{U}_f + \text{bias}_f) \tag{2.9}$$
$$\boldsymbol{i}_t = \sigma(\boldsymbol{x}_t \times \boldsymbol{W}_i + \boldsymbol{h}_{t-1} \times \boldsymbol{U}_i + \text{bias}_i) \tag{2.10}$$
$$\boldsymbol{o}_t = \sigma(\boldsymbol{x}_t \times \boldsymbol{W}_o + \boldsymbol{h}_{t-1} \times \boldsymbol{U}_o + \text{bias}_o) \tag{2.11}$$
$$\boldsymbol{a}_t = \tanh(\boldsymbol{x}_t \times \boldsymbol{W}_a + \boldsymbol{h}_{t-1} \times \boldsymbol{U}_a + \text{bias}_a) \tag{2.12}$$
$$\boldsymbol{c}_t = \boldsymbol{f}_t \circ \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \circ \boldsymbol{a}_t \tag{2.13}$$
$$\boldsymbol{h}_t = \boldsymbol{o}_t \circ \tanh(\boldsymbol{c}_t) \tag{2.14}$$

where σ is the sigmoid activation function, and $\circ$ is the element-wise product.

TreeLSTM While LSTM works well on a sequence of inputs, it doesn't explicitly work with a tree of inputs, such as an Abstract Syntax Tree (AST). One can force LSTM to work with tree-like inputs by flattening the trees into sequences of tokens, but this

normally requires adding more separator tokens (like brackets) into the sequence, making it inefficient. A better way is to directly enforce the tree structure into the update equations, which was introduced in TreeLSTM[80]. Given a tree, let C^j denote the set of children of node j. The Child-Sum TreeLSTM update equations are:

$$\boldsymbol{s}_j = \sum_{k \in C^j} \boldsymbol{h}_k \tag{2.15}$$

$$\boldsymbol{f}_{jk} = \sigma(\boldsymbol{x}_j \times \boldsymbol{W}_f + \boldsymbol{h}_k \times \boldsymbol{U}_f + \text{bias}_f) \tag{2.16}$$

$$\boldsymbol{i}_j = \sigma(\boldsymbol{x}_j \times \boldsymbol{W}_i + \boldsymbol{s}_j \times \boldsymbol{U}_i + \text{bias}_i) \tag{2.17}$$

$$\boldsymbol{o}_j = \sigma(\boldsymbol{x}_j \times \boldsymbol{W}_o + \boldsymbol{s}_j \times \boldsymbol{U}_o + \text{bias}_o) \tag{2.18}$$

$$\boldsymbol{a}_j = \tanh(\boldsymbol{x}_j \times \boldsymbol{W}_a + \boldsymbol{s}_j \times \boldsymbol{U}_a + \text{bias}_a) \tag{2.19}$$

$$\boldsymbol{c}_j = \boldsymbol{i}_j \circ \boldsymbol{a}_j + \sum_{k \in C^j} \boldsymbol{f}_{jk} \circ \boldsymbol{c}_k \tag{2.20}$$

$$\boldsymbol{h}_j = \boldsymbol{o}_j \circ \tanh(\boldsymbol{c}_j) \tag{2.21}$$

Intuitively, TreeLSTM incorporates the *summation* of outputs of children (Eq. (2.15)) into the computation of the parent node (in Eq. (2.17) to Eq. (2.19), TreeLSTM uses $\boldsymbol{s}$ instead of $\boldsymbol{h}$ as in vanilla LSTM), which enforces the tree structure. Note that TreeLSTM is a generalized version of vanilla LSTM: if each node has exactly one child, we get the same equations as in vanilla LSTM.

2.3.3 Reinforcement learning

In many tasks, a dataset of input-output pairs $(\boldsymbol{x}, y)$ is too expensive, or even impossible to obtain: imagine having to label all the best moves for half of the number of all possible chess boards! More importantly, in many cases, we do not really need one single correct output, but many possible outputs are equally good, as long as their cumulative effects are the same: as long as we reach the destination in time, it doesn't really matter whether our speed at time t is 40 or 50 km/h. To solve those two problems, we need to have a learning paradigm that optimizes for a global *goal*, while collecting the data all by itself. *Reinforcement learning* is such a paradigm, and has achieved remarkable successes in many difficult tasks, such as Robotics [86] or playing board games [82, 40].

Markov Decision Process. A standard formulation of RL is as a Markov decision process (MDP), that includes:

- a set of states S;
- a set of actions A;
- $P_a(s, s') = Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability a state s transits to another state s', by taking the action a;
- $R_a(s, s')$ is the reward *after* the transition from s to s' by taking the action a.

At each time step t, the system interacts with its environment and receives the current state s_t and the immediate reward r_t. The goal of RL is to learn a policy $\pi : A \times S \mapsto [0, 1], \pi(a, s) = Pr(a_t = a|s_t = s)$ that defines the probability of taking an action a when in state s. Note that in this thesis, we only care about finite action spaces (discrete actions). There are also continuous action spaces, e.g predicting the optimal speed at each time step. We want π to maximize the expected cumulative reward. In our setting, where P is deterministic (taking one action results in only one possible new state), the cumulative reward is defined as:

$$E = \sum_{t=0}^{T-1} R_{a_t}(s_t, s_{t+1}) \tag{2.22}$$

where T could be ∞ if necessary.

The dilemma of RL lies in the fact that we often do not have access to the true reward function R (if we do, we can simply sample many tuple $(s, a, R_a(s, s'))$ and learn it in a supervised manner), only the final reward at the end of the execution of the policy, yet in practice RL cannot be trained without intermediate feed backs. For example, imagine if we can only train a Chess RL system if the only reward is a binary signal at the end! Hence, a big part of RL research is about crafting adequate *proxy rewards*, that tells the agent how well it currently performing. One can directly craft the proxies using domain knowledge, such as assigning scores for board configurations in Chess, or use best-so-far random policy as the target (e.g Policy Gradient [84]), or learn the reward function for a state-action pair instead of the reward for just the state (e.g Deep-Q learning [35]).

1 **function** SPACER:
In: $\langle \mathit{Init}, \mathit{Tr}, \mathit{Bad} \rangle$
Out: $\langle$SAFE, $\mathit{Inv}\rangle$ or UNSAFE
$Q := \emptyset$; // POB queue
$N := 0$; // maximum safe level
$F_0 := \mathit{Init}, F_i := \top$ **for all** $i > 0$; // lemma trace
$\mathcal{U} := \mathit{Init}$; // reachable states
forever do
`Candidate` [isSAT$(F_N \wedge \mathit{Bad})$]
$Q := Q \cup \langle \mathit{Bad}, N \rangle$;
`Predecessor` [$\langle \varphi, i+1 \rangle \in Q$, $M \models F_i \wedge \mathit{Tr} \wedge \varphi'$]
$Q := Q \cup \langle$GETPREDECESSOR$(\varphi, M), i \rangle$;
`Successor` [$\langle \varphi, i+1 \rangle \in Q$, $M \models \mathcal{F}(\mathcal{U}) \wedge \varphi'$]
$\mathcal{U} := \mathcal{U} \vee$ GETSUCCESSOR$(\mathcal{U}, M)[\boldsymbol{x}' \mapsto \boldsymbol{x}]$;
`Conflict` [$\langle \varphi, i+1 \rangle \in Q$, $\mathcal{F}(F_i) \Rightarrow \neg\varphi'$]
$F_j := (F_j \wedge \mathrm{ITP}(\mathcal{F}(F_i), \varphi')[\boldsymbol{x}' \mapsto \boldsymbol{x}])$; **for all** $j \leq i+1$
`Induction` [$\ell \in F_{i+1}, \ell = (\varphi \vee \psi), \mathcal{F}(\varphi \wedge F_i) \Rightarrow \varphi'$]
$F_j \leftarrow F_j \wedge \varphi$ **for all** $j \leq i+1$;
`Propagate` [$\ell \in F_i, F_i \wedge \mathit{Tr} \Rightarrow \ell'$]
$F_{i+1} \leftarrow (F_{i+1} \wedge \ell)$;
`Unfold` [$F_N \Rightarrow \neg \mathit{Bad}$]
$N := N + 1$;
`Safe` [$F_{i+1} \Rightarrow F_i$ **for some** $i < N$]
return $\langle$SAFE, $F_i\rangle$;
`Unsafe` [isSAT$(\mathit{Bad} \wedge \mathcal{U})$]
return UNSAFE;

Figure 2.2: SPACER algorithm as described in [48], modified to make annotation coherent. $\boldsymbol{x}$ and $\boldsymbol{x}'$ are constant symbols, which typically represent program variables. We use the shorthand $\mathcal{F}(\varphi) = \mathcal{U}' \vee (\varphi \wedge \mathit{Tr})$.

```python
parser = argparse.ArgumentParser()
parser.add_argument('-P', '--p-model-path',
                    help='path to the .pt file of the model')
parser.add_argument('-F', '--fallback-mode', action='store_true',
                    help='whether to run in fallback mode')
args = parser.parse_args()
def main():
    fallback_mode = args.fallback_mode

    if fallback_mode:
        server_config={
            "p_model": None,
            "fallback_mode": args.fallback_mode
        }
    else:
        p_model = setup_model(args.p_model_path)
        server_config={
            "p_model": p_model,
            "fallback_mode": args.fallback_mode
        }

    serve(server_config)
main()
```

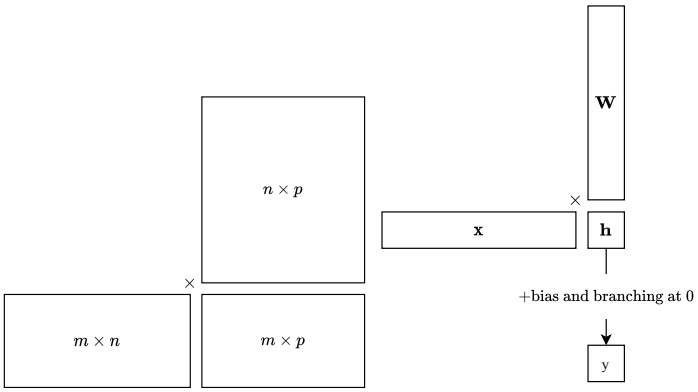
Figure 2.3: A bloated server `server.py`.

```json
{
    "main" : "server.py",
    "args" : ["-F"]
}
```

Figure 2.4: A specification file `spec.json` for `server.py`.

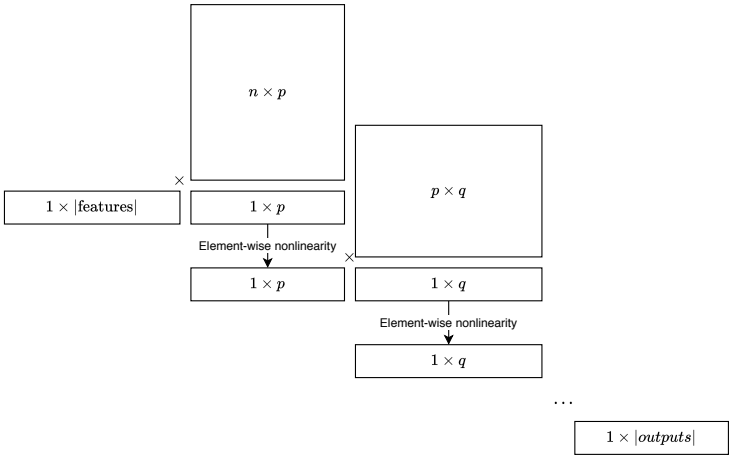
```
parser = argparse.ArgumentParser()
parser.add_argument('-P', '--p-model-path',
                    help='path to the .pt file of the model')
parser.add_argument('-F', '--fallback-mode', action='store_true',
                    help='whether to run in fallback mode')
args = parser.parse_args()
def main():
    '''
    The original code for main
    '''
    ...

def main_specialized():
    fallback_mode = True
    server_config={
        "p_model": None,
        "fallback_mode": True
    }
    serve(server_config)
main_specialized()
```

Figure 2.5: A debloated version of `server.py` with respect to `spec.json`.

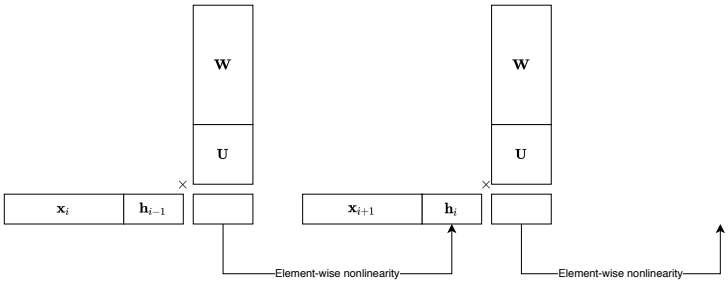


(a) Matrix multiplication.

(b) Perceptron



(c) Multi-layer Perceptron



(d) Recurrent neural network

Figure 2.6: Visualization of matrix multiplication, perceptron, multi-layer perceptron, and recurrent neural network.

Chapter 3

Learning Literal Co- and Anti-occurrences for Inductive Generalization

3.1 Introduction

Model checking has been widely used in various important areas such as robustness analysis of deep neural networks [44], verification of hardware designs [25], software verification [9], analysis [32] and testing [74], parameter synthesis in biology [11], and many others. The central challenge of model checking is to find a concise and sound approximation of all possible states a given system may reach, which does not cover any undesired states (i.e. violating given specifications). Tremendous processes have been made by innovations in efficient data representations [18], scalable SAT solvers [76, 60, 30], and effective heuristics [24, 23, 56]. Modern model checkers share a common basis, namely, IC3 [15], of which the key insight is *inductive generalization (IG)*. This idea has been generalized to support rich theories [39] that are crucial for many verification tasks [47, 34] beyond hardware verification. The generalized IC3 with rich theories, also known as satisfiability checking for Constrained Horn Clauses modulo Theory (CHC) [14], becomes the core part of a broad range of verification tasks.

Existing IG techniques follow either an enumerative search process [15, 16] or ad-hoc heuristics [33, 48]. Heuristics are effective but may demand non-trivial domain-specific (or even problem-specific) expertise. In this work, we aim to automatically learn such heuristics from the past successful IGs. We observe that verification problems as well as

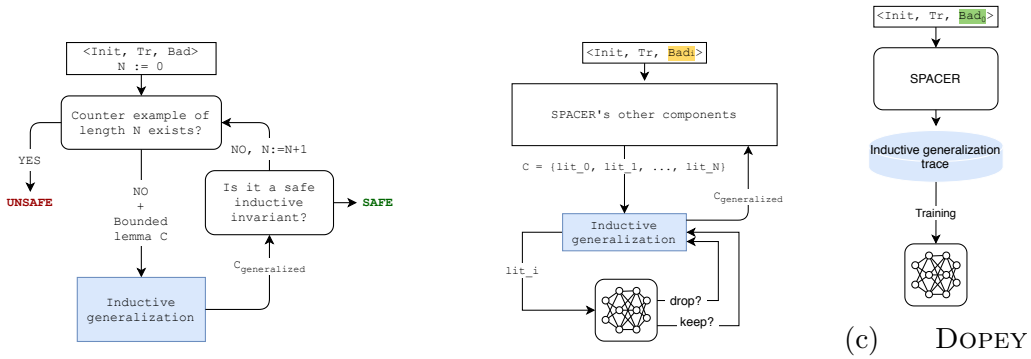
associated IGs are not isolated from each other. Taking software verification as an example, verifying different properties of the same program involves similar or same IGs; different versions of programs have similar code base; and different software may use same coding convention, idioms, library or framework, resulting in similar structures.

Our approach is inspired by the recent advances in deep learning, which automatically learn non-trivial patterns from raw pixels [49] as well as semantic correlations between natural language texts [57]. A natural solution is to train a deep learning model to directly perform IG. However, IG raises many new challenges for deep learning. First of all, the input and the output of IG are symbolic expressions, which are *highly structured* with *rich semantics*. Slight syntactic variations can lead to dramatic changes in semantics. Second, more importantly, IG has to satisfy complicated *semantic constraints*. Third, given deep learning models hardly provide any reliable guarantees, how to design a neuro-symbolic system that exhibits *learnability* from past experiences but still preserves *soundness*? All these challenges have to be properly addressed in building a neuro-symbolic reasoning framework. In this work, we share our design choices and empirical findings in building a neuro-symbolic engine Dopey, which introduces a neural component into symbolic model checking. Specifically, we make the following contributions:

- we adapt standard deep learning models to effectively represent symbolic expressions by incorporating both syntactic and semantic information;
- we design a simple but effective learning objective so that training data can be collected with nearly no changes of existing model checkers;
- our integration algorithm achieves the soundness by design, and in the worst case, the learning component may only hurt the running time performance;
- we implement Dopey on top of Spacer, a state-of-the-art CHC-solver, using an efficient client-server architecture;
- our empirical evaluations indicate Dopey significantly improves Spacer on challenging benchmarks from CHC-COMP 2018.

3.2 Overview

In this section, we give an overview of our technique, outline the challenges involved, and our key insights to address them. The context is symbolic SMT-based Model Checking



(a) SMC architecture. (b) DOPEY architecture. (c) DOPEY training.

Figure 3.1: Overview of Symbolic Model Checking and overview of DOPEY.

(SMC) [15, 39, 46], also known as satisfiability checking for Constrained Horn Clauses modulo Theory (CHC) [14]. In Model Checking, the high-level goal is to show that an infinite state transition system (Tr) does not have an execution/path that reaches a set of bad states (Bad) by finding a formula Inv that is an inductive invariant of Tr and does not intersect with Bad. The goal of CHC solving is to show that a set of First Order Logic formulas Φ that satisfy the Horn restriction [14] is satisfiable by exhibiting a symbolic formula M that defines an FOL model that satisfies Φ. The two problems are closely related. Model Checking often reduced to CHC solving. Both problems are in general undecidable.

Fig. 3.1a shows the basic structure of an SMC algorithm based on IC3 architecture. In the paper, we use SMC SPACER [46], but the architecture is common to many engines. SMC iteratively unrolls the Tr, uses an SMT solver to find a bounded counterexamples (which is usually decidable), and, if no counterexample is found, attempts to create an inductive invariant. The invariant is constructed as a set of so called *lemmas*, where each lemma is disjunction of atomic formulas. An example lemma is $x \leq 0 \vee y > 0$. For convenience, we often represent lemmas as a set of formulas, writing instead $\{x \leq 0, y > 0\}$. Many of the details of the algorithm are not important, and we omit them here. The step we focus on in this paper is *inductive generalization* (highlighted in blue in Fig. 3.1a), that is responsible for generalizing learned lemmas. In practice, this step is crucial for the performance of SMC.

Conceptually, inductive generalization is a simple process, usually done with an algorithm similar to the one we call ITERDROP, shown in Fig. 3.2. ITERDROP starts with a valid lemma $\ell = \{l_1, \ldots, l_n\}$, and proceeds to generalize ℓ by removing an arbitrary chosen literal from ℓ, and using an SMT solver to check whether the lemma is still valid (by call-

In: the original F-inductive lemma $L = \{l_1, l_2, ..., l_n\}$
Out: a generalized F-inductive lemma $K \subseteq L$

```
1  K ← ∅  // kept literals
2  C ← L  // literals to check
3  while C ≠ ∅ do
4  |  K, C ← dropOne(K,C)
5  return K
6  function dropOne(K, C)
7  |  lit ← pick(C)
8  |  if isInductive(K ∪ C \ {lit}) then
9  |  |  C ← C \ {lit}
10 |  else
11 |  |  K ← K ∪ {lit} ; C ← C \ {lit}
12 |  return K, C
```

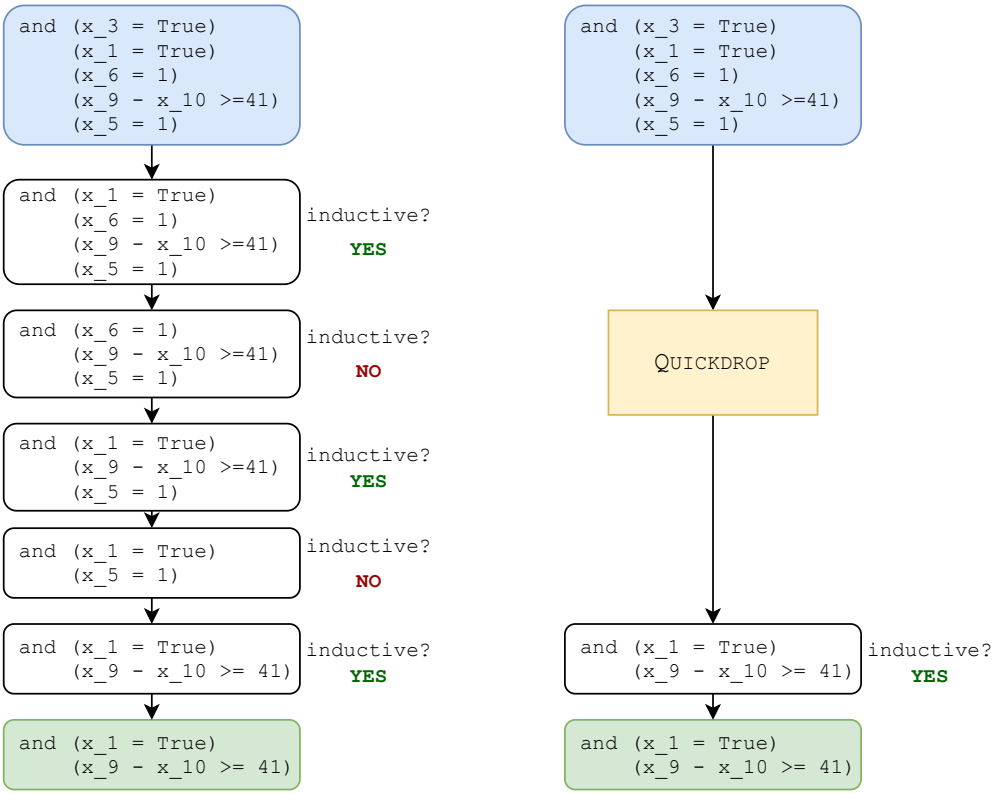
Figure 3.2: IterDrop algorithm.

ing `isInductive`). The details of `isInductive` are not important – but it can be quite expensive. If the call succeeds, the literal is removed, otherwise it is kept. The goal is to generalize to a valid lemma with fewest literals.

From now on, when the context is clear, we use *generalization* instead of inductive generalization.

We illustrate IterDrop with a sample run, shown in Fig. 3.3a. IterDrop proceeds as follows:

- it tries to drop the first literal, $x_3 = \mathsf{true}$, by checking whether $\ell'_1 = \{x_1 = \mathsf{true}, x_6 = 1, x_9 - x_{10} \geq 41, x_5 = 1\}$ is valid;
- assume that ℓ'_1 is valid, then $\ell \leftarrow \ell'_1$, and $x_1 = \mathsf{true}$ is chosen next;
- now, assume that $\ell'_2 = \{x_6 = 1, x_9 - x_{10} \geq 41, x_5 = 1\}$ is not valid. ℓ remains as is and $x_6 = 1$ is chosen next;
- assume that $\ell'_3 = \{x_1 = \mathsf{true}, x_9 - x_{10} \geq 41, x_5 = 1\}$ is valid, then $\ell \leftarrow \ell'_3$, and $x_9 - x_{10} \geq 41$ is chosen next;



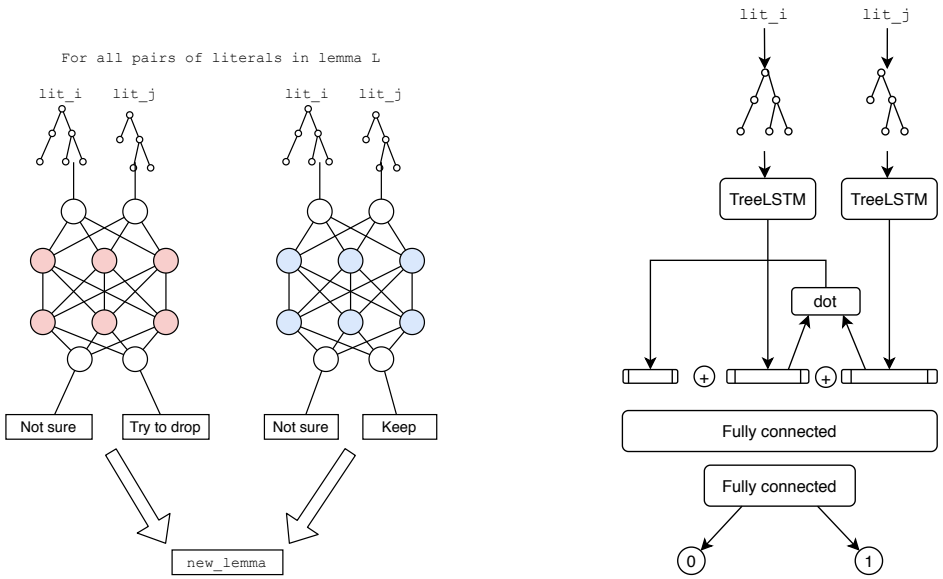
(a) IterDrop example. (b) QuickDrop example.

Figure 3.3: Examples of IterDrop and QuickDrop.

- assume that $\ell'_4 = \{x_1 = \mathsf{true}, x_5 = 1\}$ is not valid, then ℓ is unchanged, and $x_5 = 1$ is chosen next;
- assume that $\ell'_5 = \{x_1 = \mathsf{true}, x_9 - x_{10} \geq 41\}$ is valid, then ℓ'_5 is the final generalized lemma.

The example highlights the difficulty of inductive generalization. First, each call to `checkInductive` is potentially very expensive. Thus, reducing the number of the calls is highly desirable. Second, many of the calls, like steps 2 and 4 are "useless" – no new lemma is learned from them. Thus, reducing such "useless" calls is also highly desirable. Finally, a solver makes many (up to thousands) such inductive generalization calls per run.

Our *key insight* is that since generalization happens frequently, and, while the lemmas are different, the literals are similar, *it is possible to learn the co-occurrence between literals*



(a) QuickDrop example. (b) QuickDrop model.

Figure 3.4: Architectures of QuickDrop.

that do and do not occur in the same lemma together. We conjecture that once such a co-occurrence is known, it can be used to guide IterDrop to make fewer "useless " choices, and, ultimately, increase performance of SMC. Furthermore, to avoid the difficulties of online learning, we rely on the fact that many systems come with multiple properties (i.e., *Bad* states), and learning from solving one property can be transferred to solving another.

Concretely, we propose a new SMC, called Dopey. As shown in Fig. 3.1b, Dopey combines symbolic reasoning with guidance by a neural network. The core of Dopey is a new neural-based generalization algorithm called QuickDrop. Its pseudo-code is shown in Fig. 3.7. QuickDrop uses two neural networks, denoted by M^+ and M^-, to predict whether a currently chosen literal should be kept in the lemma or dropped. The prediction is based on past co- and anti-occurrences of pairs of literals in lemmas.

We illustrate a run of QuickDrop on the same example as IterDrop. Recall that the initial lemma is $\ell = \{x_3 = \mathsf{true}, x_1 = \mathsf{true}, x_6 = 1, x_9 - x_{10} \geq 41, x_5 = 1\}$. Assume that $x_1 = \mathsf{true}$ is checked and kept, QuickDrop proceeds as follows:

- it runs M^+ and predicts that $\{x_9 - x_{10} \geq 41\}$ should be kept;
- it runs M^- and predicts that $\{x_3 = \mathsf{true}, x_6 = 1\}$ should be dropped;

- it combines the results from steps 1 and 2 and suggests a candidate $\ell_{cand} = \{x_1 = \mathsf{true}, x_9 - x_{10} \geq 41\}$;
- it checks the inductiveness of ℓ_{cand}.

Note that QUICKDROP runs only *one* inductiveness check, compared to 5 used by ITERDROP.

The key idea of QUICKDROP is to use the *co-occurrence* and *anti-occurrence* between literals in lemmas to predict which literals are likely to be together in future lemmas. This intuition comes from an observation that in many generalizations either: (a) a literal is always got removed whenever another literal is kept (anti-occurrence) or (b) a literal is kept whenever another literal is also kept (co-occurrence). We believe that a) happens because the state space of a typical system under analysis can be partitioned into disjoint components, and many lemmas reference only a few of the components at a time; and b) happens because those literals are part of the same piece-wise linear function.

To learn the desired anti- and co-occurrence, we use two neural networks, called M^- and M^+, respectively. The networks are trained based on a run of an SMC on a problem instance. We gather the set of all lemmas and their generalizations, and use the data to train the network. Each network predicts, given a literal l_i the likelihood that a literal l_j appears in a lemma together with l_i. The details of the training are given in Section 3.4.

Challenges. To make DOPEY a practical verification engine, we have to address challenges in three aspects: (a) machine learning, (b) logical soundness, and (c) engineering. For machine learning, the challenge is in representing symbolic expressions as vectors, while maintaining their rich semantic structure to enable to learn and generalize co-occurrences between them. For logical soundness, the challenge is to use the neural nets in a way that guarantees the soundness of a verification engine. For engineering, the challenge is to integrate ML and symbolic components in an effective way.

Representation learning of symbolic formula. Unlike raw images or natural language text, for which there are standard deep learning models including convolutional and recurrent neural networks, a literal in a lemma is a symbolic formula, which is structured and meaning of which is sensitive to small changes. Simply viewing a literal as a sequence of tokens, as in NLP, fails to capture the subtle semantic differences between structurally similar literals.

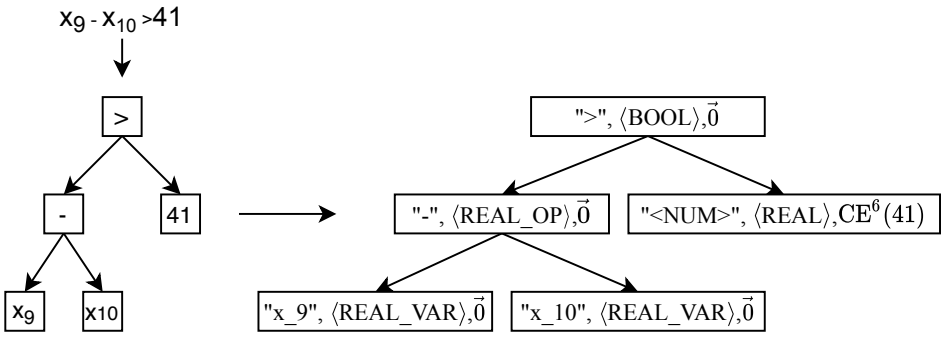


Figure 3.5: An example of an AST and its semantic features.

We incorporate both syntactic and semantic information of a literal into its representation. Our approach views a literal as a directed acyclic graph (DAG), which is post-processed from its abstract syntax tree (AST), and then adapts TreeLSTM [80] to embed such a DAG structure. Our approach also takes semantic level information into consideration so that specific properties of values are respected. For instance, embedding of numerical values should preserve their relative order and equality.

Learning for inductive generalization. Directly using machine learning to address the generalization problem is a non-trivial structure prediction problem. It takes in a set of symbolic formulas and outputs another set of symbolic formulas that are more general and more concise. Though in principle, sequence-to-sequence models [51] are applicable, they are unlikely to accomplish such a complicated reasoning task with existing ML models. In fact, even predicting equivalence between two simple symbolic expressions remains challenging [4]. Rather than having an end-to-end ML solution, we embed a learning component in a classic symbolic approach of generalization. Specifically, the learning component captures the co-occurrence between literals appearing in past runs and predicts the likelihood of keeping or dropping a literal in the current run. Furthermore, uncertainties introduced by the learning component have to be carefully controlled, which otherwise could lead to unsound conclusion. DOPEY is designed to make sound progress no matter what predictions the learning component provides. Bad predictions may be harmful to the performance, but not to soundness!

Integrating machine learning with logical reasoning. ML models and logical reasoning framework are implemented in very different programming environments and rely on different hardware. Their integration unavoidably involves communication overhead

$$
\begin{aligned}
\textbf{Token} &::= \text{Var} \mid \langle \text{NUM} \rangle \mid \text{Op} \\
\text{Var} &::= \text{variable name} \\
\text{Op} &::= + \mid - \mid < \mid \cdots \\
\textbf{Kind} &::= \langle \text{BOOL\_OP} \rangle \mid \langle \text{REAL\_OP} \rangle \mid \langle \text{BOOL\_VAR} \rangle \\
\mathbf{CE}^n(p) &::= s\ d_1 \cdots d_{2n+1} \qquad\qquad s \in \mathbb{R}, d_i \in \{0, 1\}
\end{aligned}
$$

Figure 3.6: Grammar for AST node features.

among different runtimes and hardware. This may offset performance gains of fast prediction, posing a significant engineering challenge.

3.3 Representation learning

Representing symbolic formulas. We represent logical formulas as Abstract Syntax Trees (ASTs): operators label nodes of the tree, operands are children, constants (boolean and numeric) and variables are leaves. An example of an AST is shown in Fig. 3.5.

ASTs are natural representations of formulas that are traditionally used in parsing and compilers. They preserve the key structure of the formula, while hiding (or abstracting) unnecessary details such as white space, commas and parentheses. Alternative representations, for example, as a sequence of tokens, abstract too much of the structure of the formula, while highlighting unnecessary differences.

Nonetheless, multiple semantically equivalent formulas can be represented by different trees. For example, the formulas $x + y > 0$ and $y + x > 0$ are semantically equivalent, yet differ in the concrete syntax, *and* have different ASTs. To mitigate this, we put each formula in a "normal" form by simplifying all expressions that we can (for example, rewriting $x+0$ into x), and ordering all associative operators. While there are many ways to normalize formulas (while not achieving a true normal form), we adopt a simple heuristic by using a simplification engine of the SMT solver Z3 [28]. The normalizer cannot handle deep semantic equivalences, such as normalizing $2/7 \cdot x_9 - 4/7 \cdot x_{10} \geq 0$ into $1/7 \cdot x_9 - 2/7 \cdot x_{10} \geq 0$. However, we believe it is good enough for our setting.

Note that semantically equivalent rewriting and normalization make our representations of symbolic formulas essentially *directed acyclic graphs (DAGs) modulo semantic equivalence*, because semantically equivalent subtrees share the exact same embedding. Indeed,

representations of symbolic formulas in our implementation are DAGs, although they are viewed as if they were trees by the embedding model. Without further notice, when we refer to a node in a tree, we actually mean its corresponding node in the DAG.

We use TreeLSTM [80] to aggregate information at each node in the tree into a fixed-length vector before feeding it to our subsequent networks. The process is illustrated in Fig. 3.4b. In the rest of this section, we describe the features that are used to encode each AST node into a vector to be combined by a TreeLSTM.

AST node features. We assume that the reader is familiar with the basics of the TreeLSTM neural networks. One of the main requirements is that each node N of the input tree must be represented by a fixed-length vector in $\mathbb{R}^n$.

A common technique to map a node N to a vector is to first map the infinite (or simply large) set Σ of all possible nodes into a finite set T of tokens (a.k.a. *tokenization*), and then have a dictionary E that maps each token in T into a fixed-length vector (a.k.a. *embedding*). During tokenization, many nodes have to be mapped into the same token. For example, in NLP applications, all out-of-vocabulary words are mapped into a token `<UNK>`. Similarly in Programming Language applications, all variable and function names and all numeric constants are mapped into three tokens: `<VAR>`, `<FUNC>`, and `<NUM>`, respectively.

Unfortunately, this tokenization scheme is not applicable to our setting. We believe that both the variable names and the values of the numeric constants are highly relevant! For example, consider two pairs of formulas:

$$x_1 - 2x_3 + 7x_5 \geq 10 \qquad x_1 - 2x_3 + 7x_5 \geq 14 \tag{3.1}$$

$$x_1 - 2x_3 + 7x_5 \geq 10 \qquad x_1 + x_3 - x_5 \geq 0 \tag{3.2}$$

Pair (3.1) represents two parallel hyperplanes, with the first subsuming the second. Pair (3.2) represents two intersecting hyperplanes and cannot be simplified any further. The difference between the two pairs disappears when all numeric constants are mapped to a small finite set of tokens. Yet, this difference is crucial for successful learning in our context!

Instead, we represent each AST node by a vector of features that captures the necessary semantic information. Specifically, a node N is first represented by a tuple $\langle$**Token**, **Kind**, **CE**$\rangle$. The grammar for each feature is shown in Fig. 3.6, where angled brackets are used to represent terminals (e.g., $\langle$NUM$\rangle$ is a symbol `<NUM>`). At the embedding step, **Token** and **Kind** are mapped into fixed-length vectors $\boldsymbol{T}$ and $\boldsymbol{K}$, while **CE** is *not* embedded further. The final feature vector is the concatenation of $\boldsymbol{T}$, $\boldsymbol{K}$, and **CE**.

Token captures the symbol of the node. For variables and operations this is their syntactic representation; for real valued constants, it is a single token ⟨NUM⟩. Note that each variable is kept as a *separated* token. This is possible because in our setting, each system has finitely many variables.

Kind captures the type (or sort) of the expression rooted at the AST node. The value is one of a fixed pre-defined values such as ⟨BOOL_OP⟩, for a Boolean operator, etc. In principle, the sort can be learned automatically, similar to how heads in BERT [22] seems to be able to learn part-of-speech for words in an unsupervised manner. However, we see no benefit of learning the sort since this information is already easily available.

CE (constant encoding) captures the numeric value of the node for real valued constants, and is a vector of 0 for other nodes. The encoding of each number $p \in \mathbb{R}$ is parameterized by an encoding constant n (in our experiments, $n = 6$) that determines the bounds of the magnitude of p.

Concretely, for each numerical constant p that is represented as $s \times 10^e$ in the scientific notation, we encode the significant s by its float representation and encode the exponent e using a one-hot encoding vector. To one-hot encode e, we put e in the range $[-n, n]$. If e is out of range, we set it to either the upper or lower bound. For example, $\text{CE}^2(41) = [4.1\,0\,0\,0\,1\,0]$, and $\text{CE}^0(41) = [4.1\,1]$. Our motivation for CE is to help Dopey to quickly extract magnitudes of real constants along with their value.

We conclude this section with an example. Fig. 3.5 shows an AST for $x_9 - x_{10} > 41$ and its transformation into a tree of feature vectors. The tree is further embedded and fed into a standard TreeLSTM model.

3.4 Learning co-occurrence probabilities

Recall from Section 3.2 that in this work, instead of having an end-to-end ML solution, we learn to predict whether a literal could be dropped from a lemma, based on its co-occurrences with other literals in the training data. Concretely, we want to learn how likely a literal is kept, given the existence of another literal in the solution. While calculating the exact probability of this event in the training data and learn a neural network to approximate it in a *regression* manner seems like a natural approach, it is actually not desirable: the exact probabilities, even with binning, *are not consistent between different properties of the same system*. Fortunately, we observe that pairs with co-occurrence probabilities above a certain threshold are *consistent* across different properties!

Thus, we frame our problem into a *classification problem*:

Problem 1 (Literal Co- and Anti-occurrence Classification) *Given a set of literals $\mathcal{L}$ and a scoring function $f : \mathcal{L} \times \mathcal{L} \mapsto [0, 1]$, train a classifier M s.t. $M(lit_i, lit_j) \approx f(lit_i, lit_j)$.*

We use two scoring functions:

$$f^+(lit_i, lit_j) = \begin{cases} 0 & \text{if } P^+_{ij} < threshold \\ 1 & \text{if } P^+_{ij} \geq threshold \end{cases}$$

$$f^-(lit_i, lit_j) = \begin{cases} 0 & \text{if } P^-_{ij} < threshold \\ 1 & \text{if } P^-_{ij} \geq threshold \end{cases}$$

where P^+ and P^- are co- and anti-occurrence matrices that capture the likelihood that two literals appear or do not appear in the same generalized lemma, respectively. To define P^+ and P^- formally, assume that there is a a finite set of *original* lemmas *Lem*, and a corresponding set of generalized lemmas *GLem*. Inductive generalization is a function $indgen : Lem \mapsto GLem$. The sets of all literals in *GLem* and *Lem* are *GLit* and *Lit*, respectively. A generalized lemma is indicated by a prime, so that lemma $\ell' = indgen(\ell)$ is the generalization of lemma ℓ. Then, the co- and anti-occurrence matrices are defined as follows:

$$P^+_{ij} = Pr(lit_j \in \ell' \mid lit_i \in \ell')$$
$$P^-_{ij} = Pr(lit_j \notin \ell' \mid lit_i \in \ell', lit_i \in \ell, lit_j \in \ell)$$

That is, P^+_{ij} is the probability that a literal lit_j appears in a lemma $\ell' \in GLem$ that contains lit_i, and P^-_{ij} is the probability that a literal lit_j does not appear in a lemma ℓ' given that literal $lit_i \in \ell'$ and both lit_i and lit_j are in ℓ. P^+ is of the size $|GLit| \times |GLit|$, and P^- is of the size $|Lit| \times |Lit|$. They are *not* complimentary: $P^+_{ij} + P^-_{ij} \neq 1$. Neither is symmetric: $P^+_{ij} \neq P^+_{ji}$ and $P^-_{ij} \neq P^-_{ji}$.

- Let F be a literal-frequency matrix of size $|1 \times GLit|$. F_i is the number of times a literal lit_i appears in a *generalized* lemma. Formally, $F_i = |\{\ell \in GLem \mid lit_i \in \ell\}|$.
- Let X be a literal co-occurrence matrix of size $|GLit| \times |GLit|$. X_{ij} is the number of times both lit_i and lit_j are in some *generalized* lemma in *GLem*. Formally, $X_{ij} = |\{\ell \in GLem \mid lit_i \in \ell, lit_j \in \ell\}|$. Note that $X_{ij} = X_{ji}$. For consistency, we let $X_{ii} = F_i$.
- P^+ is a *literal co-occurrence probability matrix* of size $|GLit| \times |GLit|$. P^+_{ij} is the probability that lit_j appears in a *generalized* lemma ℓ' given that lit_i is in ℓ'. Formally, $P^+_{ij} = Pr(lit_j \in \ell' \mid lit_i \in \ell') = X_{ij}/F_i$ if $F_i \neq 0$, and 0 otherwise.

The definition of the anti-occurrence matrix P^- is similar.

- Let X^{cor} be a literal co-occurrence matrix of size $|Lit| \times |Lit|$, such that X^{cor}_{ij} is the number of times lit_i and lit_j are in a lemma $\ell \in Lem$. Formally, $X^{cor}_{ij} = |\{\ell \in Lem \mid lit_i \in \ell, lit_j \in \ell\}|$. Clearly, $X^{cor}_{ij} = X^{cor}_{ji}$.
- Let X^{anti} be a literal anti-occurrence matrix of size $|Lit| \times |Lit|$, such that X^{anti}_{ij} is the number of times a pair (lit_i, lit_j) satisfies the following conditions: (a) lit_i and lit_j are in in some lemma $\ell \in Lem$, (b) lit_i is in the corresponding *generalized* lemma $\ell' \in GLem$, and (c) lit_j is *not* in ℓ'. For consistency, we set $X^{anti}_{ii} = 0$. Formally, $X^{anti}_{ij} = |\{\ell \in Lem \mid \ell' = indgen(\ell), lit_i, lit_j \in \ell, lit_i \in \ell', lit_j \notin \ell'|\}$.
- A literal anti-occurrence probability matrix P^- of size $|Lit| \times |Lit|$ is defined such t hat P^-_{ij} is the probability that lit_j is *not* the generalized lemma given that lit_i is in the generalized lemma. Formally, $P^-_{ij} = X^{anti}_{ij}/X^{cor}_{ij}$ if $X^{cor}_{ij} \neq 0$, and 0 otherwise. Note that, like P^+, P^- is not symmetric.

Dataset creation and Model training. Our goal is to learn the two classifiers M^+ and M^- that approximate f^+ and f^-, respectively. To create the sets Lem and $GLem$, we run SPACER on an instance and collect all generalization steps. This data is used to compute P^- and P^+ based on their definitions. We then fix a threshold and convert the matrices into a binary classification dataset.

We train both classifiers using the architecture shown in Fig. 3.4b. It consists of a TreeLSTM network followed by two fully connected layers. To enforce the asymmetry in the learned classifier, we take the dot product of the TreeLSTM output vectors and concatenate them together before feeding them to the first fully connected layer. Since both P^+ and P^- are sparse, we employ a negative sampling rate, directly inspired by work on learning word co-occurrence probability [63].

Discussion. There are many alternative ways to guide generalization using a neural component than the one we chose. Perhaps most desirable is to have an end-to-end solution in which the neural component takes an original lemma as input and produces a generalized lemma as output. However, the symbolic reasoning required for this is so complex that we believe that such a solution is much harder to train and scale up. Another alternative is to learn an approximation of the inductive check, i.e., the function $isInductive(Context, \ell) \mapsto \{true, false\}$ that determines whether a candidate lemma ℓ is inductive in the current context. We have tried such an approach, but could not make it effective. The difficulty

In: the original F-inductive lemma $L = \{l_1, l_2, ..., l_n\}$
Out: a generalized F-inductive lemma $K \subseteq L$

```
1  K ← initKeep(L)  // kept literals
2  C ← L  // literals to check
3  while C ≠ ∅ do
4      ΔK ← {l | l ∈ C ∧ M⁺(K, l)}
5      ΔC ← {l | l ∈ C \ ΔK ∧ M⁻(K ∪ ΔK, l)}
6      if ΔK = ΔC = ∅ then
7          K, C ← dropOne(K,C)
8      else
9          if isInductive(K ∪ C ∪ ΔK \ ΔC) then
10             K ← K ∪ ΔK
11             C ← C \ ΔC \ ΔK
12         else
13             K, C ← dropOne(K,C)
14 return K
```

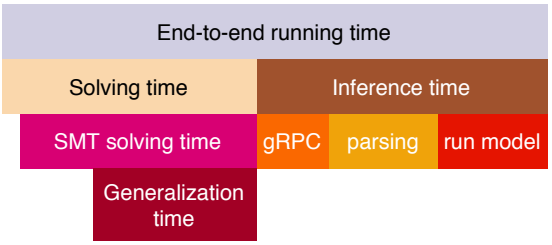
Figure 3.7: QuickDrop algorithm.

is that the *Context* that is used by the inductive checker is a large symbolic formula. This makes training the network difficult. We suspect it is as hard as *learning a neural SMT-solver* [72, 70]. We have also considered using just the co- or anti-occurrence matrices, but the results are not as good, as shown by our empirical evaluation (Section 3.6).

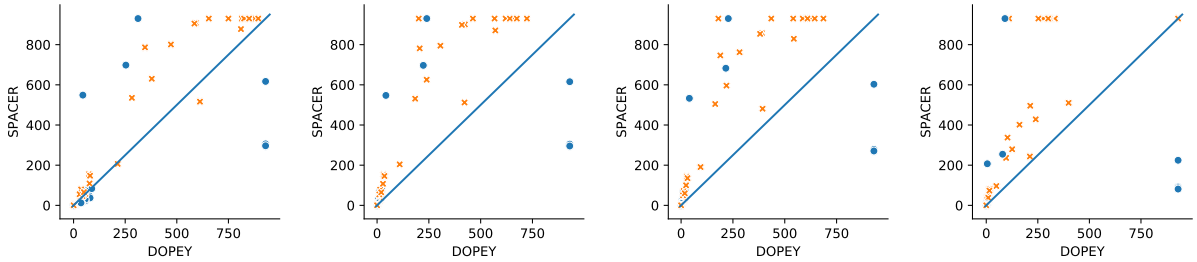
3.5 Dopey: Spacer with QuickDrop

With a positive model M^+ and a negative model M^-, we next illustrate the design and implementation of Dopey.

Inductive generalization with M^+ and M^-. Dopey uses M^+ and M^- to guide IterDrop, resulting in a new algorithm named QuickDrop (shown in Fig. 3.7). The key differences of QuickDrop from IterDrop are highlighted in blue. Instead of iteratively checking if each literal should be kept or dropped, QuickDrop makes a more aggressive decision – keep and drop many at once (line 4-5).



(a) Runtime dist. of DOPEY.

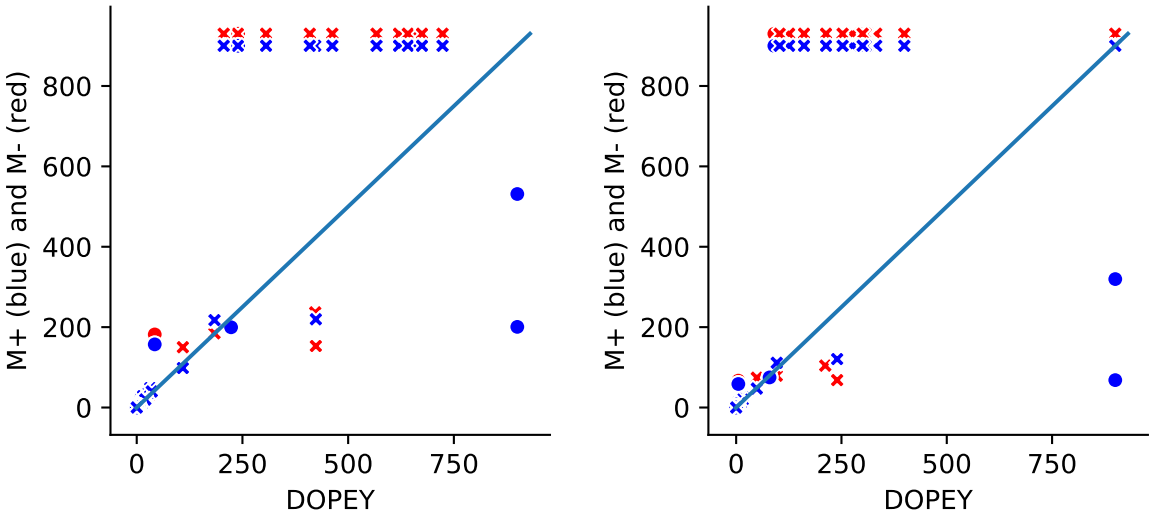


(b) Solving + infer. time. (c) Solving time. (d) SMT-solving time. (e) Generalization time.

Figure 3.8: Comparing DOPEY's and SPACER's running time, where blue dot (•) indicates an instance with unsafe property, orange cross (x) indicates an instance with safe property, top (right) most place are instances SPACER (DOPEY) timed out.

Given that deep learning models could make arbitrary predictions, special care need to be taken in order to preserve soundness. Line 1 makes sure that the initial set of kept literals is not empty. The `initKeep` can be a process similar to ITERDROP except for terminating immediately when the first literal to keep is found. Lines 6 and 9 assure that the aggressive decisions by the algorithm always result in a valid generalization; otherwise, a fallback mechanism is triggered. In the worst case, QUICKDROP should be effectively the same as ITERDROP. More formally, we have the following important yet straightforward theorem.

Theorem 1 QUICKDROP *is sound and terminating.*



(a) Solving time.

(b) Gen. time.

Figure 3.9: DOPEY vs. using only either M^+ or M^-.

Implementation and optimizations. QUICKDROP is implemented in Python using PyTorch [62], while SPACER is implemented in C++. The communication overhead between these two could diminish the performance gain by fast inductive generation, creating an engineering challenge. Simply exchanging information via file IO is too expensive. Instead, we implement a client-server architecture in which QUICKDROP is wrapped in a gRPC server connects to a gRPC client inside SPACER. As communicating and parsing over gRPC dominates the overhead, we explore further optimizations.

First, we use *caching*. Generalizing a lemma triggers multiple requests to QUICKDROP. For each request, DOPEY sends over a lemma, the server parses it and runs M^- and M^+. Since multiple requests over a period share the same lemma, we can parse it once and cache the result until a flag indicating QUICKDROP needs to handle a new lemma.

Second, we use *parallel precomputation*. Instead of invoking M^- and M^+ once for each literal pair, we *precompute* the full co- and anti-occurrence matrices at once, and use them for *all* subsequent requests. As long as the matrices fit into the GPU memory, computing the full matrices take the same time as computing one pair of literals.

Together, the two optimizations reduce overhead by up to 50%. Although gRPC communication and parsing still dominates the inference time, empirically it is good enough to achieve absolute speedups.

	All	Unsafe	Safe
Solving + infer. time	1.51	1.50	1.58
Solving time	2.16	2.46	2.02
SMT solving time	2.25	2.71	2.08
Generalization time	3.26	3.63	1.67

Table 3.1: DOPEY's speed up compared to SPACER on instances solved by both.

3.6 Empirical Evaluation

3.6.1 Setup.

We collect benchmarks from CHC-COMP 2018 [2] with a particular focus on benchmarks that corresponds to verification of hybrid systems, which are known to be challenging for inductive generalization, and for which SPACER behaves poorly. We train DOPEY on execution traces of 17 benchmarks (or hybrid systems) and test DOPEY on other 170 verification tasks (i.e., 10 different properties for each benchmark).

Training details.

We train our model using Adam optimizer [45], with embedding dimension of 16 and TreeLSTM' hidden size of 64, dropout rate 0.5, negative sampling rate 5, *threshold* set to 0.8 for both P^+ and P^-. Each model is trained up to 3,000 epochs. All experiments are done using an Ubuntu 20.04 PC with an Nvidia 1050 Ti.

3.6.2 Comparing Dopey with Spacer.

DOPEY successfully solved 41 instances on which SPACER failed.[1] On instances solved by both, the speedups achieved by DOPEY are summarized in Table 3.1. The first column lists the running time of different phases of DOPEY, of which the distribution is illustrated in Fig. 3.9a. The "All", "Unsafe", "Safe" columns show the average speedups for all instances, instances with unsafe properties (i.e. a counterexample is found), instances with safe properties (i.e. a proof is found), respectively. The second row shows that DOPEY

[1] According to CHC-COMP 2018, failure means no result is produced in 15 minutes.

is 1.51× faster than SPACER on average over all instances solved by both. The last three rows show the average speedups in specific phases.

Beyond quantitative improvements, DOPEY also achieved higher quality of inductive generalization, indicated by the safe instances (Safe column in Table 3.1). Suppose all the performance gain is from the inductive generalization alone, then the speedups should *decrease* when the overall solving time or SMT solving time is considered (as in the Unsafe column of Table 3.1). Surprisingly, this is not the case. Our analysis shows that this is because DOPEY learns *better* lemmas and improves the reasoning steps *outside* of the inductive generalization phase (SMT-solving time)!

Fig. 3.8 plots the running time of each individual instance by DOPEY and SPACER. Except for a few instances whose solving time is short, DOPEY significantly outperforms SPACER.

3.6.3 Ablation study.

To highlight the combined benefits of M^+ and M^-, we also evaluate DOPEY with a *single* model. As shown in Fig. 3.9, except for a few outliers, using both models is faster and solves more instances – DOPEY times out on 54 (58) instances when using only M^+ (M^-).

3.7 Conclusion

We proposed a neuro-symbolic system, DOPEY, which uses deep learning models to improve symbolic model checking. DOPEY uses a positive and a negative model to approximate co- and anti-occurrences of literals appeared in past inductive generalizations. The logical and neural component of DOPEY are implemented as an efficient gRPC client-server architecture. Our empirical evaluation on CHC-COMP 2018 indicates significant improvements of DOPEY over the state-of-the-art SMC engine, SPACER: inductive generalization is 3.26× faster, and total solving time is 1.51× faster. To the best of our knowledge, DOPEY is the first neural-symbolic SMC guidance that works well in practice. We open source the code and welcome contributors to contribute, improve and extend DOPEY.

This chapter is adapted from our collaboration with Dr.Xujie Si, and is currently under submission.

Chapter 4

Reinforcement Learning Guided Software Debloating

4.1 Introduction

The rapid increase of software productivity in the last decades was fueled by the extensive use of abstraction and reuse of software components. While this enabled building larger and more complex software systems, these gains came at the expense of less efficient and less secure software systems, and contribute to a troublesome trade-off between productivity and performance/security. When an application built using general-purpose components is deployed, the majority of the general-purpose functionality is never used. This creates two problems: first, as the use of abstraction layers makes it more difficult to optimize the software, this has a detrimental impact on performance; second, it increases the attack surface for security vulnerabilities.

An emerging solution to this problem is a set of tools, called *Software Debloaters*, that automatically customize a program to a user-specified environment. One successful approach for software debloating is based on partial evaluation (PE) [42] in which a *partial evaluator* takes a program and some of its input values and produces a *residual* (or *specialized*) program in which those inputs are replaced with their values. While PE has been extensively applied to functional and logic programs, it was less successful on imperative C/C++ programs (with a noteable exception of C-Mix [7] and Tempo [26]). With the advent of LLVM [52], several new partial evaluators of LLVM bitcode have arisen during the last few years (e.g., LLPE [77], Occam [54], and Trimmer [73]). These tools leverage LLVM optimizations such as constant propagation, function inlining, and others

to either reduce the size of the residual program (e.g., OCCAM and TRIMMER) or improve performance (e.g., LLPE).

The key step of a PE-based debloater is to estimate whether specializing (or inlining) a function results in a smaller (or more secure) residual program. Specialization naturally increases the size of the code since it adds extra functions. However, since some inputs in the new functions have been replaced by constant values, optimizations such as constant propagation, might become enabled and result in new optimization opportunities that reduce the code size. Therefore, the decision whether or not a function should be specialized for a particular call-site is non-trivial. Current PE-based software debloaters use naive heuristics. For example, OCCAM implements two heuristics of "never specialize" or "always specialize", respectively; TRIMMER specializes a function only if it is called once in the whole program.

In this paper, we present a new approach based on reinforcement learning (RL) to automatically infer effective heuristics for specializing functions for PE-based software debloating. RL is a good fit for the problem for two reasons. First, code specialization resembles a Markov Decision Process: each decision moves the code from one state to another, and specialization depends only on the current state rather than the history of the transformations. Second, the quality of debloating can only be measured after all specialization actions and corresponding compiler optimizations have been performed.

The main challenge in applying RL is in deciding on a good state representation. While it is tempting to use the source code (or LLVM intermediate representation (IR)) as a state, this is not computationally tractable. The IR is typically hundreds of MBs in size. Instead, an adequate set of features that captures meaningful information about the code while avoiding state aliasing is required. In particular, these features must capture the *calling context* in order to distinguish between call-sites.

The main contributions of this paper are threefold: (1) calling context features that enable RL to find a useful heuristic for PE-based debloating software; (2) implementation of our method in a tool called DEEPOCCAM; and (3) evaluation on the reduction of the program size and number of possible code-reuse attacks, by comparing our handcrafted features with features learned automatically via embedding LLVM IR into a vector space using INST2VEC [13]. Our initial evaluation suggests that RL is a viable method for developing an effective specialization policy for debloating, and learning with handcrafted features is easier than with INST2VEC.

Callee	Caller	Module	Call-Site
Basic blocks	Basic blocks	Functions	Arguments
Instructions	Instructions	Instructions	Known arguments
Store instructions	Store instructions	Basic blocks	
Call instructions	Call instructions	Direct calls	
Branch instructions	Branch instructions		
Loops	Loops		
Uses	Uses		
Influenced branches	Untouched call-sites		
Influenced instructions			

Table 4.1: RL features. *Influenced instructions (branches)* are the those whose operands have statically known arguments. *Uses* indicates how many times the *caller* and the *callee* are invoked.

4.2 Methods

In this section, we formulate the debloating problem as a reinforcement learning problem, describe the learning procedure, and its hyperparameters.

Action, state, and reward. To determine whether a call-site should be specialized or not, we train a policy using reinforcement learning. This requires defining *action*, *state*, and *reward*.

An action is whether to run a code transformation, called *specialization*, or not. Consider a call to function *calleeF* with arguments A at a call-site c_i. Let $\mathsf{formals}$ be a map from call-site arguments to their corresponding formal parameters. Let B, $B \subseteq A$, be the arguments whose values are are statically known. Then, specialization of the call-site does:

1. Create a new function *spec_calleeF*($\mathsf{formals}(A \setminus B)$), such that *spec_calleeF*'s body is identical to the body of the original *calleeF* except that $\mathsf{formals}(B)$ are replaced with their values.

2. Perform constant propagation to push forward the information to the rest of callee's body, potentially specializing more call-sites in the callee.

3. Replace $calleeF(A)$ with $spec\_calleeF(A \setminus B)$ at the call-site c_i.

Note that after specialization, a software debloater can trigger other optimization passes such as inlining and dead-code elimination.

A state is a vector capturing all the relevant information about the code. We experiment with two state representations: (a) a vector of hand crafted features (**HF**), and (b) an embedding of LLVM IR via INST2VEC (**IV**).

In **HF**, the state is a concatenation of four feature vectors, summarized in Table 4.1. They capture relevant information about the *callee*, *caller*, *compilation unit (module)*, and *call-site*. Most features are self-explanatory, and hence, we focus only on those which are novel or more relevant for avoiding state aliasing. The *state aliasing* problem occurs when two different states have the same representation in the RL model. In our context, our features must distinguish between two call-sites in the same basic block when invoking the same callee with the same arguments if one is specialized and the other not. In [50], the features *InLoop* (whether a call-site is in a loop), *InlineDepth* (the current inlining depth), and *currentGraphSize* (how many instructions are there in the caller, including one added by inlining), are used to decide inlining. However, in our case, these features are insufficient to prevent state aliasing. Thus, we also add *Untouched call-sites* that counts the number of call-sites yet to be processed. Since the call-sites are processed in a fixed order, *Untouched call-sites* is sufficient to distinguish any two call-sites in a function.

In **IV**, we use INST2VEC embedding from [13]. We extract the LLVM IR of the *caller*, the *callee*, and the *calling context* (a window of n instructions around the call-site) as a lists of instructions. Each instruction is embedded into a vector space using INST2VEC. Additionally, we encode the arguments at the call-site as a bitvector, in which each statically known argument is encoded by 1, and an unknown argument by 0. This bitvector is then embedded using a different embedding matrix. Finally, the **IV** state is a tuple of the above four 2-D matrices. Note that in this case, the calling context of each call is explicitly represented by the IR.

For rewards, we focus on two different metrics. First, we measure the number of instructions in the final binary produced by the software debloater after specialization took place. Second, we measure the reduction of the *attack surface*, focusing on code reuse attacks.

Code reuse attacks are exploits in which an attacker makes use of the available instructions in the binary to chain together short sequences called *gadgets*, and use those gadgets to compromise the control flow of the program, causing a malicious effect. Those sequences

are often categorized based on their last instruction, into ROP (return-oriented programming), JOP (jump-oriented programming), and COP (call-oriented programming) gadgets, respectively. Since the relationship between the number of gadgets and exploitability is an open question [17], we focus on reducing the number of gadgets without making any further claim about the security of the debloated code.

The rewards are the negations of the number of instructions, ROP, JOP, and COP gadgets. The negation is necessary because we are interested in minimizing our metrics. We only have one measurement at the end of an episode (when the final binary is produced). Immediate rewards after each action are set to zero.

Learning procedure, policy network, and hyper-parameters. Each state representation requires a different neural net for the policy network. For **HF**, we use a 3-layer fully connected network. Before training, we run the pipeline with a random policy 100 times to calculate the mean and standard deviation of each feature, and use this metadata to normalize all features into mean of 0 deviation of 1.

For **IV**, we run the *caller*, the *callee*, the *calling context* and the *arguments bitvector* through four separate 2-layer GRU [21] blocks, concatenate the last hidden outputs of these 4 blocks, and then feed it to a 3-layer fully connected network. The architecture is depicted in Fig. 4.1.

Both networks use ReLU [61] as the activation function. We use REINFORCE [84] with normalized rewards to update the policy for both models. At each REINFORCE iteration, we roll out k runs of the current policy, batch them together, and use the Adam optimizer [45] to update the network. For all metrics, we use the same hyper-parameters: INST2VEC calling context $n = 10$, number of runs in each policy rollout $k = 75$, Adam's learning rate $= 0.001$, and train up to 340 iterations.

4.3 Implementation and Evaluation

We have implemented our prototype, called DEEPOCCAM, using OCCAM [54]. The architecture of DEEPOCCAM is depicted in Fig. 4.2. DEEPOCCAM takes as inputs a set of LLVM modules (main application and libraries) and a manifest (i.e., a user-defined execution environment) and produces a specialized binary. We used Gadget Set Analyzer (GSA) [17] to evaluate the specialized binary and PYTORCH [62] for training the deep learning models.

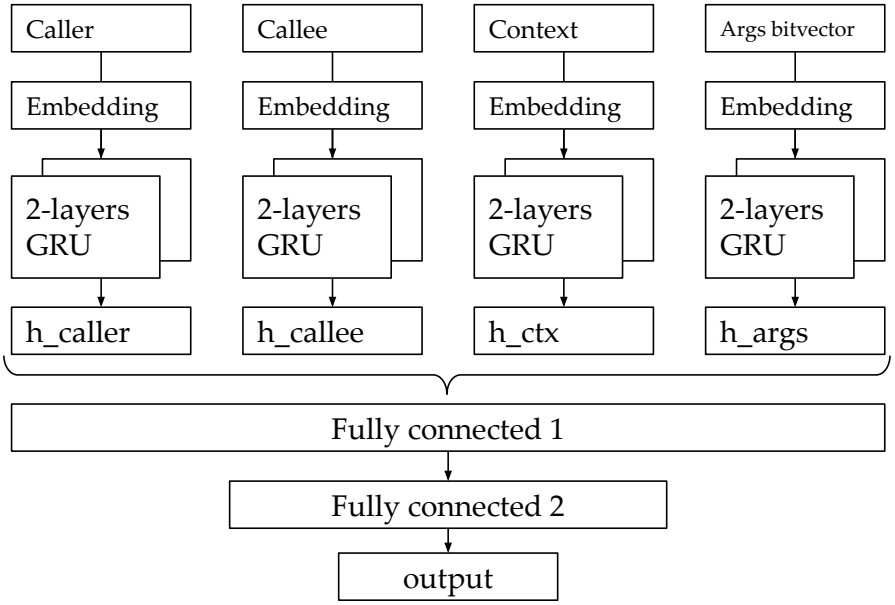


Figure 4.1: Policy network architecture using INST2VEC.

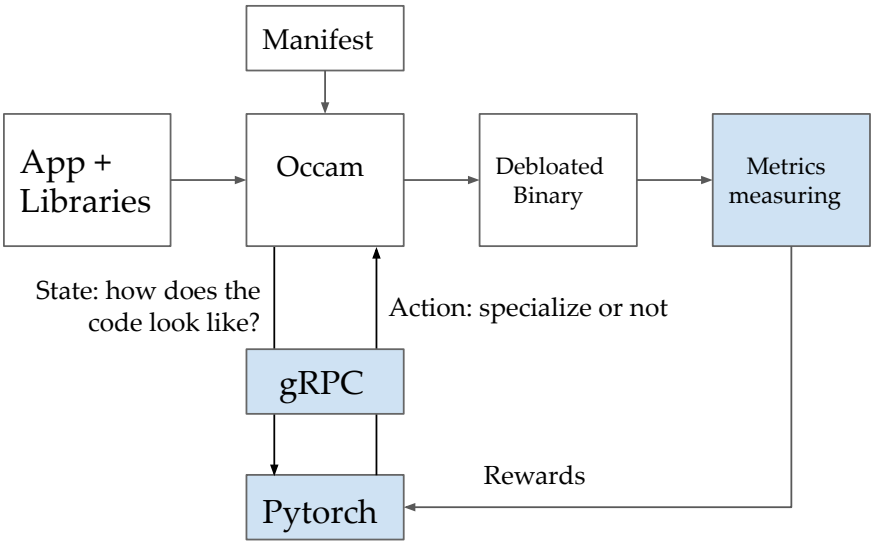


Figure 4.2: DEEPOCCAM architecture. Boxes in blue correspond to this work.

At each call-site, OCCAM calculates the features described in Table 4.1 from the LLVM module, sending them to the PYTORCH server, and transforming the code based on the returned decision. During inference, the PYTORCH server receives the message and runs the learned policy to decide whether to specialize that call-site. During training, the PYTORCH server also keeps track of all the decisions it has made and uses that information to update its policy. We run multiple copies of OCCAM on multiple copies of the PYTORCH server to scale up the learning process, which is justified by the Markov property of the state.

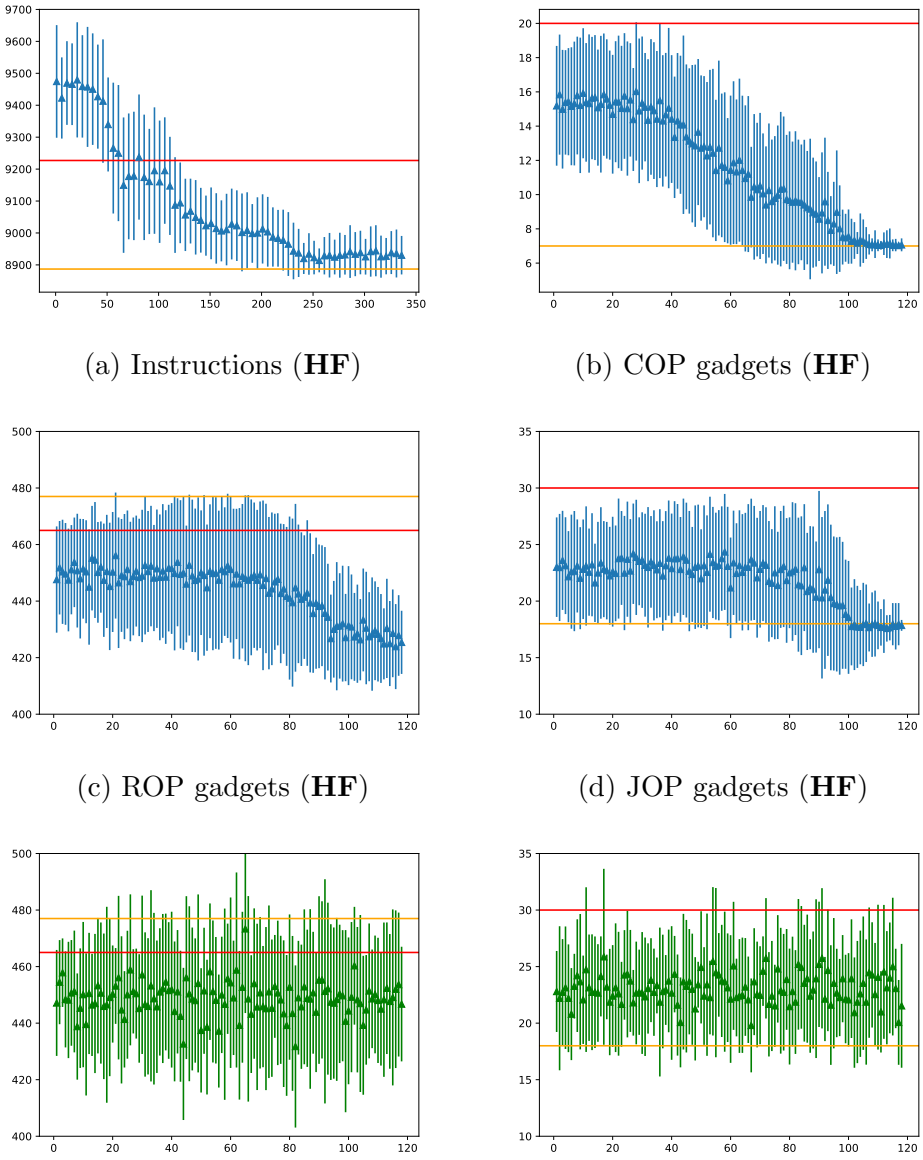
We compare DEEPOCCAM with OCCAM running in two modes. First, OCCAMAGG that runs OCCAM with a *nonrecursive-aggressive* policy. This always specializes a call-site if the callee is not a recursive function. Second, OCCAMNONE that runs OCCAM without specializing any call-site. Fig. 4.3 shows the comparision between DEEPOCCAM using **HF** , DEEPOCCAM using **IV**, OCCAMAGG, and OCCAMNONE for debloating GNU TREE. On average, DEEPOCCAM outperforms both OCCAMNONE and OCCAMAGG in reducing the number of ROP and JOP gadgets, matches OCCAMNONE on COP gadgets, and underperforms in reducing the number of instructions. Interestingly, while DEEPOCCAM matches OCCAMNONE on COP, it finds a different policy that specialize some call-sites. For optimizing the number of instructions, we run 200 more training iterations but are not able to outperform OCCAMNONE. We include the results for the number of instructions only to demonstrate the flexibility of our approach.

In all of our experiments, the learning procedure does not converge when we use INST2VEC pre-trained embedding. Upon closer inspection, we observe that the software in our test suite, once compiled to LLVM IR, contained many instructions regarded as low-frequency by INST2VEC. Consequently, they are all mapped into the same `UNK!` token in the cutoff dictionary. Hence, the calling context window that we use suffers from the state aliasing problem: different calling contexts are mapped to the same vector of tokens.

This is surprising since INST2VEC claims to train the embedding on a wide range of software written in C, C++, FORTRAN, and OpenCL, specifically to avoid overfitting to a small family of code bases.

4.4 Related work and Conclusions

Recent advances in deep learning and RL have opened new frontiers to the design of compiler heuristics [8]. Most current approaches define actions at the compilation unit level: whether to run a particular optimization pass, or how to schedule optimization



(a) Instructions (**HF**)

(b) COP gadgets (**HF**)

(c) ROP gadgets (**HF**)

(d) JOP gadgets (**HF**)

(e) ROP gadgets (**IV**)

(f) JOP gadgets (**IV**)

Figure 4.3: Results for optimizing different metrics using **HF** and **IV**. x-axis is the number of RL iterations. y-axis is the results. Dots and bars are mean and std. dev. of k runs in each iteration. **HF** results are in blue while **IV** results are in green. Orange and red lines are OCCAMNONE and OCCAMAGG results, respectively.

passes. For example, Cummins et al. [27] model code as a natural language problem to predict whether to run the code on CPU or GPU, as well as the optimal GPU thread coarsening factors. Kulkarni et al. [50] use NEAT — a genetic algorithm — to learn an inlining heuristic for MaxineVM as a neural network. For interpretability, they then approximate the neural network by a decision tree. We, on the other hand, use actions at much lower granularity, focusing on specialization at each call-site.

Program code is a rich structure that can be presented in a variety of ways, e.g., as raw text, control-flow and call- graphs, etc. Recent application of deep learning for code experiment with models based on techniques from Natural Language Processing and Graph Neural Network (e.g., [6, 13, 5, 20]). Among them, INST2VEC [13] is the only one that works on LLVM IR.

The closest related work is Chisel [37] – a software debloater based on RL. Inspired by C-Reduce [66], Chisel takes a program P and a property of interest φ (i.e., P must compile and pass tests defining φ) and produces the smallest program P' that satisfies φ. RL accelerates the search for the reduced program providing a scalability boost. Compared to DEEPOCCAM, Chisel has very different state and action space, and, is generally only as sound as the test cases defining φ.

Conclusions In this thesis, we present DEEPOCCAM— an end-to-end tool to learn specialization heuristics for software debloating. Our preliminary results suggest that it is feasible to use RL to learn an effective specialization heuristic to optimize a variety of related metrics. They also suggest that out of the box, pretrained embedding such as INST2VEC might not be applicable for the task.

We hope that DEEPOCCAM contributions in feature engineering and architecture might be applicable to other compiler optimization tasks such as inlining.

This chapter is adapted from the following published work:

- Nham Le, Ashish Gehani, Arie Gurfinkel, Susmit Jha, Jorge A.Navas, *Reinforcement Learning Guided Software Debloating*, Workshop on ML for Systems at NeurIPS 2019

Chapter 5

Future Work

Recent advances in deep learning, especially language modeling, has played a tremendous role in solving many natural language reasoning tasks that used to be considered infeasible. Yet, neural-guidance for symbolic reasoning is still a barely charted territory, with successes few and far between. In previous chapters, we have studied two exciting applications of learning a better heuristic for symbolic reasoning tasks, specifically Inductive Generalization, and Software Debloating. In this chapter, we outline some future directions on guiding symbolic reasoning using machine learning, which we think are feasible, and can play a major role in making neural-guidance not just another tool, but a powerful hammer in tackling down symbolic reasoning tasks.

Representation learning for formal languages. Deep learning has made tremendous progress in understanding natural language, and in many cases surpassing human-level [3]. However, there is a big gap between natural language understanding and formal language understanding. Natural language, as a means of transferring information through noisy channels (i.e sound wave through the air, or word-of-mouth gossiping), has a lot of redundancy in both grammar and context [85]. This redundancy helps with, among others, enhancing the comprehensibility of natural language. In contrast, formal languages, such as formulas or programming languages, are designed to be as concise and compact as possible, with little to no redundancy. This makes representation learning for formal languages very challenging. For example, many breakthroughs in NLP are based on the word-guessing tasks — given a sentence, we randomly remove some words in it, and try to guess them using the remaining words — such as word2vec [58] or BERT [22]. Due to redundancy, it is reasonable to try guessing a word from its context, but it is counter-intuitive to try to do the same for formal languages.

One promising way to learn a good representation for formal languages is to use the structure of the entities. In Chapter 3, we show that augmenting an AST with domain knowledge about the tokens can achieve good results on representing linear arithmetic formulas. Given that graph-based and tree-based methods have been successful in structure learning tasks, such as drug discovery [19], it is reasonable to believe that representation learning for small to medium mathematical formulas is feasible. While in Chapter 4, we cannot transfer results of inst2vec [13] into our domain, this graph-based work shows promising results in many different tasks, and it is foreseeable that better feature engineering at the token level may help us achieve even better code representation.

Transfer learning for formal languages. One of the main reasons for the wide adoption of deep learning is the success of *transfer learning*. Starting from a model pretrained on millions of labeled images using hundreds of GPU hours, an individual researcher can easily fine tune it for a task that may have only a handful of training examples, using a small amount of computing power. ELMo [64] and BERT [22] were big breakthroughs in NLP for the same reason: previously, each NLP task required a full end-to-end training, making it very difficult to use deep learning to solve NLP problems with small datasets.

The main challenge in successful transfer learning for symbolic reasoning lies in the fact that downstream tasks are often *representation specific*: while one can imagine a model trained on a dataset about the Python programming language can be helpful to do Python reasoning tasks (e.g neural-based auto-completion), there is little reason to believe it would help detect memory leaks in C++, a bug that is completely missing in Python. This is very different in natural language, where tasks for different languages are often the same: answering questions, analysing sentiments, extracting entities, among others. Because of this reason, while the amount of available code and formulas are tremendous, it is still not clear how one can train a useful general model that can be finetuned to solve multiple different tasks.

A useful idea that borrowed from the compilers community: at the end of the day, all programming languages have to be compiled to machine instructions. A model trained on this lowest level of abstraction, or some other forms of Intermediate Representation (IR), could be the key to the puzzle. While there is still no pre-trained model that is helpful for multiple tasks in practice, we envision that this is a solvable missing piece to make neural-guidance learning truly useful.

Chapter 6

Conclusion

This thesis presents a glimpse into the feasibility of learning state-of-the-art heuristics for symbolic reasoning tasks, specifically, inductive generalization and software debloating, using neural networks.

For inductive generalization, we proposed a neuro-symbolic system, DOPEY, which uses a positive and a negative model to approximate co- and anti-occurrences of literals that appeared in past inductive generalizations to improve the overall symbolic model checking process. For software debloating, we present DEEPOCCAM— an end-to-end tool to learn specialization heuristics using Reinforcement Learning.

In both cases, our results show that the learned heuristics are better than state-of-the-art handcrafted ones, even when we take into account the communication costs between our neural components in Python and our symbolic engine in C++, in the case of DOPEY. To the best of our knowledge, they are both the first neural-symbolic guidance that works well in practice for their tasks.

Looking forward, we want to extend the work presented in this thesis to make it more useful, by trying to learn a better representation for the formulas and LLVM IRs, as well as trying to make the learned heuristics transferable to different problems in the same task.

References

[1] Microgadgets: Size does matter in turing-complete return-oriented programming. In *Presented as part of the 6th USENIX Workshop on Offensive Technologies*, Bellevue, WA, 2012. USENIX.

[2] Constrained horn clauses (chc) competition, 2018.

[3] Glue benchmark leaderboard, 2020.

[4] Miltiadis Allamanis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles A. Sutton. Learning continuous semantic representations of symbolic expressions. In *ICML 2017*, volume 70, 2017.

[5] Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *CoRR*, abs/1808.01400, 2018.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. pages 40:1–40:29, 2019.

[7] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.

[8] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5):96:1–96:42, 2019.

[9] Thomas Ball. Secrets of software model checking. In *AGP 2002*, 2002.

[10] Mislav Balunovic, Pavol Bielik, and Martin T. Vechev. Learning to Solve SMT Formulas. In *NeurIPS*, 2018.

[11] J. Barnat, L. Brim, A. Krejci, A. Streck, D. Safranek, M. Vejnar, and T. Vejpustek. On parameter synthesis by parallel model checking. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(3), 2012.

[12] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[13] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. *CoRR*, abs/1806.07336, 2018.

[14] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Gurevich 75*, 2015.

[15] Aaron R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, 2011.

[16] Aaron R. Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In *FMCAD*, 2011.

[17] Michael D. Brown and Santosh Pande. Is less really more? towards better metrics for measuring security improvements realized through software debloating. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, Santa Clara, CA, August 2019. USENIX Association.

[18] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8), 1986.

[19] Hongming Chen, Ola Engkvist, Yinhai Wang, Marcus Olivecrona, and Thomas Blaschke. The rise of deep learning in drug discovery. *Drug Discovery Today*, 23(6):1241 – 1250, 2018.

[20] Zimin Chen and Martin Monperrus. A literature study of embeddings on source code. *CoRR*, abs/1904.03061, 2019.

[21] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[22] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. What does BERT look at? an analysis of bert's attention. *CoRR*, abs/1906.04341, 2019.

[23] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods Syst. Des.*, 19(1), 2001.

[24] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.

[25] Edmund M. Clarke, Kenneth L. McMillan, Sérgio Vale Aguiar Campos, and Vasiliki Hartonas-Garmhausen. Symbolic model checking. In *CAV*, 1996.

[26] Charles Consel, Luke Hornof, Renaud Marlet, Gilles Muller, Scott Thibault, E-N Volanschi, Julia Lawall, and Jacques Noyé. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys*, 30(3es), 1998.

[27] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232, Sep. 2017.

[28] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[29] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, page 125–134, Austin, Texas, 2011. FMCAD Inc.

[30] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, 2003.

[31] Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. Can neural networks understand logical entailment? In *ICLR*, 2018.

[32] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI*, 2002.

[33] Alberto Griggio and Marco Roveri. Comparing different variants of the ic3 algorithm for hardware model checking. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 35(6), 2016.

[34] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In *CAV*, 2015.

[35] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 2094–2100. AAAI Press, 2016.

[36] Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Learning fast and precise numerical analysis. In *PLDI*, 2020.

[37] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *CCS*, pages 380–394, 2018.

[38] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[39] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, volume 7317, 2012.

[40] Sean D. Holcomb, William K. Porter, Shaun V. Ault, Guifen Mao, and Jin Wang. Overview on deepmind and its alphago zero ai. In *Proceedings of the 2018 International Conference on Big Data and Education*, ICBDE '18, page 67–71, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989.

[42] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993.

[43] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[44] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. The marabou framework for verification and analysis of deep neural networks. In *CAV*, 2019.

[45] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[46] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In *CAV*, 2014.

[47] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. Automatic abstraction in smt-based unbounded software model checking. In *CAV*, 2013.

[48] Hari Govind Vediramana Krishnan, YuTing Chen, Sharon Shoham, and Arie Gurfinkel. Global guidance for local generalization in model checking. In *CAV*, 2020.

[49] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, 2012.

[50] Sameer Kulkarni, John Cavazos, Christian Wimmer, and Doug Simon. Automatic construction of inlining heuristics using machine learning. In *CGO*, pages 9:1–9:12, 2013.

[51] Guillaume Lample and François Charton. Deep learning for symbolic mathematics, 2019.

[52] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–, 2004.

[53] Nham Le, Ashish Gehani, Arie Gurfinkel, Susmit Jha, and Jorge A Navas. Reinforcement learning guided software debloating. 2019.

[54] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. Automated software winnowing. In *SAC*, pages 1504–1511, 2015.

[55] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. *CoRR*, abs/1907.08194, 2019.

[56] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.

[57] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *NeurIPS*, 2013.

[58] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, 2013.

[59] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 2017.

[60] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, 2001.

[61] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress.

[62] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[63] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *EMNLP*, 2014.

[64] Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[65] Anh Quach, Aravind Prakash, and Lok-Kwong Yan. Debloating software through piece-wise compilation and loading. In *USENIX Security Symposium*, pages 869–886, 2018.

[66] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *PLDI*, pages 335–346, 2012.

[67] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. CLN2INV: learning loop invariants with continuous logic networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.

[68] Venkata Keerthy S, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. Ir2vec: A flow analysis based scalable infrastructure for program encodings. *CoRR*, abs/1909.06228, 2019.

[69] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[70] Daniel Selsam and Nikolaj Bjørner. Guiding High-Performance SAT Solvers with Unsat-Core Predictions. In *SAT*, 2019.

[71] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. *CoRR*, abs/1802.03685, 2018.

[72] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT Solver from Single-Bit Supervision. In *ICLR*, 2019.

[73] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. Trimmer: Application specialization for code debloating. In *ASE*, pages 329–339, 2018.

[74] Oleg Sheyner, Joshua W. Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *SSP*, 2002.

[75] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *NeurIPS*, 2018.

[76] João P. Marques Silva and Karem A. Sakallah. GRASP – a new search algorithm for satisfiability. In *ICCAD*, 1996.

[77] Christopher S.F. Smowton. I/O Optimisation and elimination via partial evaluation. Technical Report UCAM-CL-TR-865, University of Cambridge, Computer Laboratory, December 2014.

[78] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[79] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[80] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *ACL*, 2015.

[81] O. Tange. GNU Parallel – The Command-Line Power Tool. *;login: The USENIX Magazine*, 36(1), February 2011.

[82] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995.

[83] A. Turing. *Checking a Large Routine*, page 70–72. MIT Press, Cambridge, MA, USA, 1989.

[84] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256, May 1992.

[85] Ernst-Jan C Wit and Marie Gillette. What is linguistic redundancy. 1999.

[86] Yilin Wu, Wilson Yan, Thanard Kurutach, Lerrel Pinto, and Pieter Abbeel. Learning to manipulate deformable objects without demonstrations, 2020.