Machine Learning for Function Synthesis

Julian Parsert

St. Catherine's College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2024

Acknowledgements

First, I would like to thank my supervisors Daniel Kröning and Tom Melham for their guidance and support throughout my studies. Their advice on research, writing, and other matters has been invaluable.

I want to thank Daniel Kröning in particular, for having impeccable foresight with regard to the COVID-19 pandemic and providing precious advice. In hindsight, I can say without a doubt, that his advice has led to me being able to enjoy a quality of life during the pandemic that would have been inaccessible otherwise. I am forever thankful for that.

I am also thankful to my colleagues in Oxford and Edinburgh for the countless insightful discussions and ideas. In particular, I want to thank my collaborators Mirco Giacobbe and Elizabeth Polgreen. I genuinely enjoyed working with you and I believe that I learned a lot from that experience.

Similarly, I want to thank Aart Middeldorp, Vincent van Oostrom, Christian Sternagel, and the Computational Logic group in Innsbruck as a whole for many interesting discussions throughout the years. In particular, I want to point out that the logic course that I attended in 2015 was one of the best courses I have ever attended and is certainly in no small part responsible for my research interests.

When I started my bachelor's degree in 2014 I would have never imagined that I would end up pursuing a doctorate degree. I am certain there is nobody that I have to thank more for introducing me to research in computer science than Cezary Kaliszyk. I am thankful for all professional advice that Cezary provided but also the countless and sometimes long (Google Maps tells me our longest discussion took at least 9 hours) discussions about computer science, mathematics, life, the universe, and everything. When attending Cezary's functional programming course in 2015 I did not expect this to be the beginning of a beautiful friendship. It is safe to say that my life would look very different if I had not been fortunate enough to meet Cezary.

Finally, last but not least, I want to thank my friends and family for providing motivation and sometimes necessary distractions. It is thanks to you that I was able to conduct my research while simultaneously pursuing other endeavours in life.

Abstract

Function synthesis is the process of automatically constructing functions that satisfy a given specification. The space of functions as well as the format of the specifications vary greatly with each area of application. In this thesis, we consider synthesis in the context of satisfiability modulo theories. Within this domain, the goal is to synthesise mathematical expressions that adhere to abstract logical formulas. These types of synthesis problems find many applications in the field of computer-aided verification. One of the main challenges of function synthesis arises from the combinatorial explosion in the number of potential candidates within a certain size. The hypothesis of this thesis is that machine learning methods can be applied to make function synthesis more tractable.

The first contribution of this thesis is a Monte-Carlo based search method for function synthesis. The search algorithm uses machine learned heuristics to guide the search. This is part of a reinforcement learning loop that trains the machine learning models with data generated from previous search attempts. To increase the set of benchmark problems to train and test synthesis methods, we also present a technique for generating synthesis problems from pre-existing satisfiability modulo theories problems. We implement the Monte-Carlo based synthesis algorithm and evaluate it on standard synthesis benchmarks as well as our newly generated benchmarks. An experimental evaluation shows that the learned heuristics greatly improve on the baseline without trained models. Furthermore, the machine learned guidance demonstrates comparable performance to `cvc5` and, in some experiments, even surpasses it.

Next, this thesis explores the application of machine learning to more restricted function synthesis domains. We hypothesise that narrowing the scope enables the use of machine learning techniques that are not possible in the general setting. We test this hypothesis by considering the problem of ranking function synthesis. Ranking functions are used in program analysis to prove termination of programs by mapping consecutive program states to decreasing elements of a well-founded set. The second contribution of this dissertation is a novel technique for synthesising ranking functions, using neural networks. The key insight is that instead of synthesising a mathematical expression that represents a ranking function, we can train a neural network to act as a ranking function. Hence, the synthesis procedure is

replaced by neural network training. We introduce Neural Termination Analysis as a framework that leverages this. We train neural networks from sampled execution traces of the program we want to prove terminating. We enforce the synthesis specifications of ranking functions using the loss function and network design. After training, we use symbolic reasoning to formally verify that the resulting function is indeed a correct ranking function for the target program. We demonstrate that our method succeeds in synthesising ranking functions for programs that are beyond the reach of state-of-the-art tools. This includes programs with disjunctions and non-linear expressions in the loop guards.

Preface

Contributions to Co-Authored Works

The majority of the content of this thesis originates from two research papers that were written by my collaborators and me during my studies. These papers are:

- Reinforcement Learning and Data-Generation for Syntax-Guided Synthesis [1]
- Neural Termination Analysis [2]

Some parts of this thesis are taken almost verbatim from the papers, while others are altered to fit within the scope of the thesis. My personal contributions to the papers are discussed in the following:

Reinforcement Learning and Data-Generation for Syntax-Guided Synthesis: This paper forms the basis for Chapter 4. I was responsible for the ideas behind the adaption of AlphaZero's algorithm to SyGuS as well as the problem generation procedure using unification and anti-unification. In the former, I was solely responsible for the implementation, while in the latter I contributed the majority of the source code with some contributions by my collaborator. In addition, I conducted the majority of the experimental evaluation. My collaborator, Elizabeth Polgreen and I contributed equally to the writing of the paper.

Neural Termination Analysis: Chapter 6 contains content from this paper. We introduce an approach to machine learning in termination analysis based on three components: *tracing*, *learning*, and *checking*. Mirco Giacobbe and I contributed equally to the main ideas presented in the paper, with high-level guidance from my supervisor Daniel Kröning. I was solely responsible for the implementation of *tracing*, while *checking* was entirely implemented by my collaborator. The main contributions of the paper are contained in *learning*, which was implemented in equal parts by Mirco Giacobbe and me. I collected the benchmark sets and conducted the experimental evaluation while writing was split evenly between Mirco Giacobbe and me with minor edits by Daniel Kröning.

Contents

List of Figures **xi**

List of Tables **xiii**

List of Abbreviations **xv**

1 Introduction **1**

1.1 Motivation 1
1.2 Current Techniques 3
1.3 Goal 4
1.4 Contributions 5

2 Preliminaries **9**

2.1 First-Order Theories 9
2.2 Function Synthesis 10
2.3 Program Analysis 13

3 Literature Survey **21**

3.1 Function Synthesis 21
3.2 Termination Analysis 31

4 Reinforcement Learning in Syntax-Guided Synthesis **35**

4.1 Background 37
4.2 Motivating Example 40
4.3 The Game of Function Synthesis 42
4.4 Monte-Carlo Grammar Tree Search 46
4.5 Search Guidance with Policy and Value 49
4.6 Features 52
4.7 Training Models and Reinforcement Learning 55
4.8 Generating New SyGuS Problems 57
4.9 Experimental Evaluation 62
4.10 Threats to Validity 71
4.11 Related Work 73

5 Application of SyGuS: Termination Analysis **77**
5.1 Background on Program Analysis 79
5.2 SyGuS Problems for Termination Analysis 82
5.3 Termination Analysis Problem Sets 85
5.4 Experimental Evaluation 88
5.5 Threats to Validity 92
5.6 Related Work 93

6 Neural Termination Analysis **95**
6.1 Motivating Example 97
6.2 Neural Termination Analysis Framework 100
6.3 Tracing and Data Generation 103
6.4 Learning Neural Ranking Functions 108
6.5 Verification 116
6.6 Experimental Evaluation 120
6.7 Threats to Validity 130
6.8 Related Work 131

7 Conclusion **137**
7.1 Future Work 139

Appendices

A Example: Generated SyGuS Problem from SMT **147**

B New Termination Problems **151**

C Generating Execution Traces **153**
C.1 Input Sampling 153
C.2 Tracing 154

References **157**

List of Figures

2.1 SMT-LIB syntax to check if there exists a model M in `LIA` such that $M \models x + 2y = 10 \land x - y = 2$. 10
2.2 The procedure for checking if a given candidate f is a solution for a SyGuS problem (T, G, ϕ, F). 13
2.3 Figure 2.3c, shows the Java class and function represented by the snippet shown in Figure 2.3a. Finally, Figure 2.3b shows the Java Bytecode we obtain by compiling the Java class in 2.3c. 14
2.4 Part of a java function with corresponding bytecode. 16

4.1 `SyGuS-IF` problem for a function computing the maximum of two numbers. 40
4.2 Grammar describing the set of terms of addition of 1 and 2 with the corresponding grammar tree. 44
4.3 Bag-Of-Words feature vector of dimension M with 7 words with index 1 and 4 with index 544. 55
4.4 Number of solved problems in the testing set for each iteration in each of the 5 experiment runs. Each line represents one experimental run. 69
4.5 Number of solved problems in the testing set within a certain time budget in each iteration. The dashed blue line is the first iteration with a default value/policy. 69

5.1 Venn diagram of all states S, all possible states according to the program L, all states defined by the transition constraints T_H, and all states defined by the auxiliary invariants A_H. 80
5.2 Ranking function synthesis problem stated in SyGuS-IF. 83
5.3 Code snippet that has a non-linear and disjunctive loop guard but where the body exhibits “linear” behaviour (i.e. an increase by one for `a` and `b`). 87

6.1 Loop with a simple increment in the loop body and disjunction in the loop guard. 97

6.2 SyGuS-IF solution for the ranking function synthesis problem produced by the snippet in Figure 6.1. 99
6.3 Neural network representing the function $\max\{y-x,0\}+\max\{z-x,0\}$. 99
6.4 Schema of the entire Neural Termination Analysis framework consisting of tracing, training, and verification. 101
6.5 Code snippet with nested loops and three variables. 108
6.6 Architecture for monolithic NRFs with n input neurons where the last layer consists of constant weights 1. 111
6.7 Architecture for lexicographic NRFs. 114
6.8 Loop with non-linear and disjunctive loop guard from the NEW set. . 125
6.9 Percentage of problems solved for neural networks with 1 to 10 hidden neurons. 126
6.10 Code snippet template with variables a and $n_1, \ldots n_k$ a single loop with a single increment in the loop body. The loop guard consists of k disjuncts. 126
6.11 We plot what percentage (as part of the whole problem set) of problems is solved within a certain time budget. 127
6.12 Loops where insufficient data is a problem. 129
6.13 Trace at loop head suggests two ranking functions but one needs an additional auxiliary invariant. 129

A.1 Problem Primes_true-unreach-call.c_276.smt2 from the LIA problem set. 148
A.2 Resulting SyGuS problem generated from SMT problem in Figure A.1.149

B.1 Problem named DynamiteExampleX4. This is one of the simplest examples which is a slight variation from the running example presented in [153]. 151
B.2 Problem named DynExUpTo3VarsDisj with 3 disjuncts and non-linear guard. Non-linearity comes from squaring a single variable, a. 151
B.3 Problem named DynExUpTo4VarsDisj. Same as Figure B.2 but with 4 disjuncts. 152
B.4 Problem named PolyClosingIn. The loop guard has a non-linear term. Furthermore, the variable m decreases while a increases. . . . 152
B.5 Problem named SquareDisj2Vars. We have two disjuncts, one of them is non-linear as it contains a squared term. The other disjunct is linear. 152
B.6 Problem named TwoPol4VarsDisj. The loop guard is a disjunction of two conditions containing polynomial expressions. 152

List of Tables

3.1 Related work that applies some form of machine learning to synthesis categorised in Programming-By-Example (PBE), Natural Language (Natural L.), abstract logical constraints (Logical), and partial programs (Partial). We differentiate the following AI techniques: simple learning models (e.g. Bayesian models like probabilistic grammars), neural networks (NN), pre-trained models (PTM), and reinforcement learning (RL). 31

4.1 Number of functions at depth up to 7 split into complete and partial programs with total being the sum of both. We start with the starting symbol of the grammar at depth 1. 42

4.2 Number of problems by category: basic problems (B), straight-line problems(S), problems with control-flow (C), and unsolved problems (U). 62

4.3 Table showing the experimental results comparing baselines to the best learned iterations. The first and second rows show experimental results on training and testing data respectively. The last row shows the results of the testing data when only considering the subset of problems that originate from the old set. In either case, a problem can only occur in exactly one of test set and train set. 65

4.4 Experimental results with different training and testing sets. Testing and training problems do not overlap. The two old sets indicated with an asterisk are the exact same set. 66

5.1 Termination analysis benchmark sets with the number of problems without and without nesting for `Aprove_09` (TermComp), `term-crafted-lit` (SV-COMP), and `nuTerm-advantage` (nuTerm). 87

5.2 Results of running `RFSynth` with `cvc5` and ϕ`ser` as well as other state-of-the-art tools on all benchmarks without nested loops. . . . 89

6.1 Results of running ν`Term` (with 7 neurons), `AProVE`, `Ultimate`, and `DynamiTe` on `SV-COMP`, `TermComp`, and `NEW`. In the case of ν`Term` we report the average results rounded to the first decimal. The last two columns show the union of the first two problem sets and all three problem sets, respectively. 122
6.2 Comparing results when split up into problems with nested loops and without nested loops. We also add the results from Chapter 5 of running `RFSynth` with `cvc5` to compare. 123

List of Abbreviations

AI . . . . Artificial Intelligence
CEGIS . . . . Counterexample Guided Inductive Synthesis
CHC . . . . . . Constrained Horn Clauses
DSL . . . . . . Domain-Specific Language
JVM . . . . . . Java Virtual Machine
JVMTI . . . . Java Virtual Machine Tool Interface
LGG . . . . . . Least General Generaliser
LIA . . . . . . Linear Integer Arithmetic
LLM . . . . . . Large Language Model
MCTS . . . . . Monte-Carlo tree search
ML . . . . Machine Learning
NN . . . . Neural Network
NRF . . . . . . Neural Ranking Functions
NTA . . . . . . Neural Termination Analysis
PBE . . . . . . Programminy-By-Example
PTM . . . . . . Pre-Trained Models
RL . . . . Reinforcement Learning
SAT . . . . . . Boolean satisfiability problem
SMT . . . . . . Satisfiability modulo theories
SOTA . . . . . State of the art
SSA . . . . . . Single Static Assignment
SV-COMP . . Software Verification Competition
SyGuS . . . . . Syntax-Guided Synthesis
SyGuS-IF . . . Syntax-Guided Synthesis Input Format
TermComp . . Termination Competition
UCT . . . . . . Upper Confidence Bound for Trees.

1
Introduction

1.1 Motivation

Function synthesis describes the process of generating functions that satisfy a given specification. From a theoretical point of view, function synthesis can be characterised as any method to produce a witness for f that satisfies an existential formula of the form:

$$\exists f.\ \phi \tag{1.1}$$

In this setting, the specification ϕ is a logical formula describing what properties f should satisfy. Historically, Alonzo Church defined and used the notion of function synthesis to derive complexity results [3, 4] as early as 1957. In this seminal work, the synthesis targets were circuits and the specifications were given using an extension of recursive arithmetic. Hence, applying the aforementioned formalism f would take the form of terms describing circuits and ϕ would be a specification in the language of recursive arithmetic. Unfortunately, but also unsurprisingly, Church also identified the undecidability of all but the most restricted versions of synthesis. Nevertheless, in subsequent work, function synthesis has found numerous practical and theoretical applications with many different function and specification formats.

These days, one popular domain of function synthesis is that of Programming-By-Example, where the specification takes the form of a set of input/output pairs. In other words, the synthesis procedure is tasked with constructing a function that, given the inputs produces the corresponding outputs, for all specification pairs. This domain has found huge success in automatic data wrangling and spreadsheet formula generation [5]. Most recently, the rapid development of deep neural networks and large language models has led to a surge in popularity of many synthesis-like tasks related to automatic program generation. For example, generating programs in general purpose programming languages from natural language specifications has become a popular problem [6]. Similarly, GitHub's Copilot [7] promises to support programmers by providing autocompletion, automatic program generation or documentation, and many other assistive features.

Finally, function synthesis has many applications in *computer-aided verification and program analysis* [8–10]. Over the years, computer hardware and software have become larger and more complex. Simultaneously, computer systems have made their way into every corner of modern-day life. As a result, providing guarantees about these systems has become *more important* and *more difficult* at the same time. Formal methods, and computer-aided verification in particular, promise to provide a solution for this discrepancy by utilising computers to alleviate the difficulties of having to reason about the ever-growing number of complex systems [11]. Most program analysis techniques of today employ function synthesis techniques in one way or another. The most prominent example of this is Hoare logic [12, 13], which describes an axiomatic basis for proving correctness of computer programs. One of the main ingredients to proving partial correctness are *loop invariants*. Generating invariants is an instance of function synthesis. Furthermore, Hoare logic also identifies a connection to a different important problem in program analysis – termination. *Termination analysis* has been a problem of interest in computer science since its inception. Historically speaking, the unsolvability of the "halting problem", and therefore termination, was proven by Turing in what is considered one of the founding works of computer science [14]. Following this, termination analysis

for computer programs, transition systems, and other models of computation became an important area of research in theoretical computer science. From a more practical point of view, non-termination bugs are omnipresent [15] and have been known to cause problems in real world software [16], making automated termination analysis invaluable. The combination of theoretical interest and practical applications has led to termination analysis being one of the most fundamental problems in computer-aided verification and formal methods. As a result, many techniques for termination analysis have been developed and implemented in different tools. One practical method for proving termination is by means of synthesising a ranking function [17]. As the name suggests, this, too, is an instance of function synthesis.

1.2 Current Techniques

Satisfiability Modulo Theories (SMT) describes an extension of Boolean satisfiability to incorporate first-order [18] and higher-order [19] theories. By utilising theory specific decision procedures, SMT solvers perform very well despite being significantly more expressive than traditional Boolean satisfiability solvers. As a result, SMT and the corresponding standardised SMT-LIB input format have been widely adopted by the formal methods community. Capitalising on the success of SMT, the synthesis community developed an input format for Syntax-Guided Synthesis (SyGuS). As the name suggests, SyGuS is a synthesis problem that allows users to provide a semantic and syntactic specification using the language of SMT. Hence, synthesis in the context of SMT describes solving problems as shown in (1.1) where ϕ is an SMT formula. This is extended in syntax-guided synthesis, where f additionally has to satisfy

$$f \in L, \tag{1.2}$$

where L is a set of expressions defined by a context-free grammar. The majority of SyGuS solvers are "enumerative solvers". These solvers systematically enumerate expressions $e \in L$ and use SMT solvers to check if e satisfies ϕ. Enumerative solvers

use techniques such as Counterexample Guided Inductive Synthesis (CEGIS) [20], blocking constraints [21] etc. to speed up verification and prune the search space.

Interestingly, synthesis techniques that apply machine learning techniques are few and far between. In addition, the vast majority of synthesis tools that do apply some form of statistical learning only consider the domain of Programming-By-Example (PBE). We hypothesise that this is due to the relative ease with which artificial training data can be generated in this domain compared to other domains. In contrast, function synthesis problems resulting from *program analysis* usually cannot be represented using PBE. In fact, most synthesis problems originating from program analysis applications utilise abstract SMT expressions and formulas as specifications. As a result, machine learning techniques are rarely used in function synthesis applied to verification.

1.3 Goal

The goal of this thesis is to develop different techniques for using *machine learning* in *function synthesis*. In particular, we want to tackle synthesis problems in the context of program analysis and SMT. This entails investigating where the usage of machine learning (ML) techniques is most effective, what type of ML models can be applied in each setting, and how the correctness of the algorithms and results can be ensured despite the usage of statistical models. We proceed as follows:

First, we consider the domain of Syntax-Guided synthesis. This is the most general setting, as all SMT based synthesis problems can be transformed into a SyGuS problem. We present an algorithm based on AlphaZero's Monte-Carlo tree search [22] to prune the enormous search space using machine learning based heuristics. These heuristics are supposed to prioritise functions that are likely to be solutions to the synthesis problems. The machine learning models are trained on data gathered from a reinforcement learning loop. In each iteration, the aforementioned algorithm is applied to the training set problems, and from this, we collect training data that is used to train the models for successive iterations. To adequately train

and test our methods, we also present a method for automatically generating new problem sets from pre-existing SMT problems.

Subsequently, we move away from the general setting of SyGuS and address a more restricted case of function synthesis problems – *ranking function synthesis*. Ranking functions are used in proving termination of computer programs. Hence, we investigate the use of a SyGuS solver as a termination prover for Java programs. However, using the general setting of SyGuS for ranking functions prohibits the use of domain-specific knowledge that we have access to in the setting of termination analysis. In particular, by focusing on termination analysis we can use the input programs that we want to analyse to generate execution traces. These are used as training data for machine learning models. Consequently, this approach is out of reach in the more general data-scarce setting. We follow the principle wherein *finding a proof is much harder than checking that a given candidate proof is valid*. The hypothesis is that one can leverage neural networks and gradient descent to synthesise ranking functions while using traditional formal methods to check the validity of the proposed candidates. To support this theory, we develop and evaluate a framework – Neural Termination Analysis – that encapsulates the data generation, training, and verification of ranking functions all in one go.

1.4 Contributions

The main contribution of this work is the establishing of a portfolio of techniques to apply machine learning to different function synthesis settings. The contributions are as follows:

Syntax-Guided Synthesis This dissertation contributes to this research area in two ways. First and foremost, we present SyGuS with abstract logical specifications as a tree search problem to which we apply an AlphaZero style tree search. This search is guided using machine learned heuristics that are trained iteratively in a reinforcement learning loop. Secondly, we present a technique for automatically

generating more SyGuS problems with abstract specifications from pre-existing SMT problems. Listed in detail, the contributions are as follows

- We state enumerative function search in SyGuS as a tree search problem and adapt AlphaZero's Monte-Carlo tree search (MCTS) based algorithm so that we can apply it to enumerative function synthesis.
- We present a method for expressing synthesis states and actions in the context of SyGuS that is conducive to machine learned policy and value predictors to guide the search. These predictors are then used with the Upper-Confidence Bound for Trees (UCT) to balance exploration and exploitation during the synthesis procedure.
- We use a reinforcement learning loop that allows for the iterative generation of training data from successful and unsuccessful synthesis procedures.
- We design and implement a method for generating SyGuS problems from SMT problems using anti-unification and unification in order to overcome a lack of training data. This approach applies to generating training data for any synthesis problems with logical specifications.
- We evaluate the algorithm and the reinforcement learning setup on a combination of pre-existing benchmark sets from the SyGuS competition [23] as well as our newly generated problem set.

The significance of these contributions taken together is that we present a method for using machine learning heuristics for synthesis that also work in domains with abstract logical specifications. In addition, we also provide a method for generating new synthesis problems to challenge and test synthesis tools on unseen problems.

Ranking Function Synthesis The main contribution of this thesis to this field of study is *Neural Termination Analysis*. Neural Termination Analysis is a framework that combines data generation, neural network training, and verification to synthesise correct ranking functions. In particular, the contributions are:

- We evaluate the use of SyGuS for ranking function synthesis.
- We identify a class of termination problems that are difficult for state-of-the-art tools. These involve disjunctive loop guards, non-linear loop guards, or a combination of both.
- We design Neural Termination Analysis as a method for applying machine learning to termination analysis in a “guess-and-check” paradigm.
- We propose machine learning primitives such as loss functions and neural network architectures that are suitable for Neural Termination Analysis.
- We implement Neural Termination Analysis, consisting of a tracing tool for the Java Virtual Machine, the aforementioned primitives for neural ranking functions, and a verification procedure for these ranking functions.
- We test the efficacy of Neural Termination Analysis on a standard set of benchmarks. Furthermore, by developing termination problems that state-of-the-art tools struggle to solve, we present the strengths of Neural Termination Analysis.

These contributions provide a novel method for applying machine learning to function synthesis in restricted settings: By translating abstract specifications to the world of differentiable functions, we can utilise the power of gradient descent to synthesise functions. Conversely, by translating the differentiable functions to symbolic expressions, we can benefit from the correctness guarantees provided by formal methods. Neural Termination Analysis provides a basis for combining these two seemingly unrelated areas.

2

Preliminaries

2.1 First-Order Theories

We will assume that the reader is familiar with the syntax and semantics of first-order logic. We refer to [24] for a general introduction and [18] for details on Satisfiability Modulo Theories (SMT). In notation, we will use $M \models \phi$ to denote that a model M satisfies or entails a first-order formula ϕ. For brevity, we will not use syntax to distinguish a separate interpretation function and instead assume it is part of M. We define a theory T as a collection of models. It helps to think of theories as "a set of models that satisfy a certain set of axioms". However, since there are theories that are not finitely axiomatisable (e.g. Peano Arithmetic) we will talk about collections of models instead. Satisfiability Modulo Theories (SMT) considers the satisfiability problem for theories. We say that a formula ϕ is T-satisfiable (satisfiable in short) if there exists a model M in T, such that $M \models \phi$. We say T-unsatisfiable if it is not satisfiable. Finally, we say that ϕ is T-valid (valid in short) if for all models M in T, we have $M \models \phi$. We will write $T \models \phi$ in short to denote that ϕ is T-valid. SMT solvers are programs designed to decide the satisfiability and validity questions for a given formula and theory. As it turns out, many theories with real-world applications are decidable [25]. Therefore, the strengths of SMT solvers in contrast to fully-fledged first-order logic theorem

```
(set-logic LIA)
(declare-const x Int)
(declare-const y Int)
(assert (= (+ x (* 2 y)) 20))
(assert (= (- x y) 2))
(check-sat)
```

Figure 2.1: SMT-LIB syntax to check if there exists a model M in LIA such that $M \models x + 2y = 10 \land x - y = 2$.

provers derive from the fact that they implement theory specific decision procedures. An example of such a theory is the theory of *linear integer arithmetic* (LIA). The SMT library (SMT-LIB) defines a unified input format for SMT problems that most SMT solvers use [26]. Figure 2.1 shows an example of a simple problem for the theory of LIA stated in the SMT-LIB format.

2.2 Function Synthesis

Inherently, function synthesis is a second-order problem. In particular, we are looking at problems that can be expressed as a second-order existential formula

$$\exists f.\ \phi, \tag{2.1}$$

where ϕ is a first-order formula most likely containing f. In this formula, f is a second-order variable, and hence we have an existential second-order quantification. In some cases, it is enough to prove (2.1) in a non-constructive way. However, usually, we are also interested in the witness for f. This is even more evident in function or program synthesis in domains such as spreadsheet formulas, string manipulation, or general purpose programming languages. In those settings, knowing that a function exists without having a witness is usually of no use. For example, knowing that a function computing the sum of all entries of a table exists does not help a user of Excel as much as actually providing the function. However, in settings such as ranking function synthesis the mere existence is already sufficient, although a witness can be useful in interpreting or verifying the result.

2.2.1 Syntax-Guided Synthesis

Context-Free Grammar: A context-free grammar G is a tuple $G = (N, T, R, S)$, where N is a finite set of non-terminal symbols, T is a finite set of terminal symbols, and $N \cap T = \emptyset$. $R \subseteq N \times (N \cup T)^*$ is a set of production rules and $S \in N$ is the start symbol, where * is the Kleene star. For $x, y \in (N \cup T)^*$ and $(\alpha, \beta) \in R$ we say that $x\alpha y$ yields $x\beta y$, written $x\alpha y \rightarrow x\beta y$. For $x \in T^*, y \in (N \cup T)^*$ and $(\alpha, \beta) \in R$ we write $x\alpha y \rightarrow^L x\beta y$ to denote the *leftmost derivation* and $\rightarrow^{L^*}$ for its reflexive, transitive closure. Hence, $u \rightarrow^L v$ denotes an application of a rule in R to the leftmost non-terminal in u. A word w is in *sentential form* if $w \in (N \cup T)^*$ and is called a *sentence* if $w \in T^*$. Since sentences do not contain non-terminals, no reductions can be applied anymore. We define the language of G, written L^G as all sentences that can be obtained from reductions starting at S. Hence, we have $L^G = \{w \in T^* \mid S \rightarrow^{L^*} w\}$. We refer to [27] for more details on context-free grammars.

Syntax-Guided Synthesis: As noted above, in program synthesis we are interested in solving a second-order existential problem while also being interested in the witness. SyGuS makes this even more explicit by allowing syntactic restrictions on the witness [28]. SyGuS works within the domain of first-order theories as introduced in Section 2.1. Formally, a SyGuS problem is a 4-tuple $\langle T, G, \phi, F \rangle$ such that T is a first-order theory, G is a context-free grammar, ϕ is a first-order formula, and F is a function symbol that may occur in ϕ. SyGuS closely follows the syntax and semantics of SMT, and hence T usually refers to theories that are also common in SMT. A solution to a SyGuS problem $\langle T, G, \phi, F \rangle$ is either a function f such that $T \models \phi[F \mapsto f]$ and $f \in L^G$ or a proof that no such function can exist. Note that the notation $\phi[F \mapsto f]$ denotes the replacing of all occurrences of F in ϕ with f while substituting all arguments to f by the arguments of F in the same order. We will denote by `ALL` the free theory that is the union of all theories supported by the Solver. Finally, we use the free grammar $*_G$ to indicate

that we do not impose any syntactic restrictions on the solution space. Hence, any term legal in the given theory is allowed.

Most SyGuS solvers can be classified as enumerative solvers. These solvers systematically enumerate the space of all potential solutions (i.e. elements in L^G) and for each element $f \in L^G$ check if $\phi[F \mapsto f]$ is T-valid. Due to SyGuS supporting the same theories as SMT solvers, one can use off-the-shelf SMT solvers to check this validity. Note that since SMT solvers usually work with SAT and $UNSAT$ we check the unsatisfiability of $\neg\phi[F \mapsto f]$ which is equivalent to checking the validity of the formula. In summary, enumerative solvers operate in two stages, a search stage where a candidate is constructed and a verification stage where an SMT solver is queried to check if the candidate is a solution or not.

Counter-Example Guided Inductive Synthesis (CEGIS): CEGIS [20] is a common technique used by synthesis tools to speed up verification by abusing the following facts:

- SMT solvers produce a (counter) model in the case of SAT.
- If ϕ does not contain variables, checking for validity is extremely efficient.

Assume we want to solve a problem of the form $\exists F.\ \forall \vec{x}.\ \phi$. Figure 2.2 gives an overview of how CEGIS works where if we remove the universal quantifier $\vec{x}$ is the vector of all free first-order variables in ϕ. The procedure keeps a set I, of assignments to the free first-order variables $\vec{x}$ in ϕ. Initially, this set is empty. Instead of immediately checking for validity for a given f, we first check if it is consistent with all assignments $\vec{c} \in I$. This has the advantage that we only have to check the validity of the ground term

$$\bigwedge_{c \in I} \phi[F \mapsto f, \vec{x} \mapsto \vec{c}].$$

Checking the validity of this conjunction is very efficient, as I is finite and every single conjunct is ground (i.e. does not contain variables). If the conjunction is inconsistent, f is not a solution to the SyGuS problem. On the other hand, if the

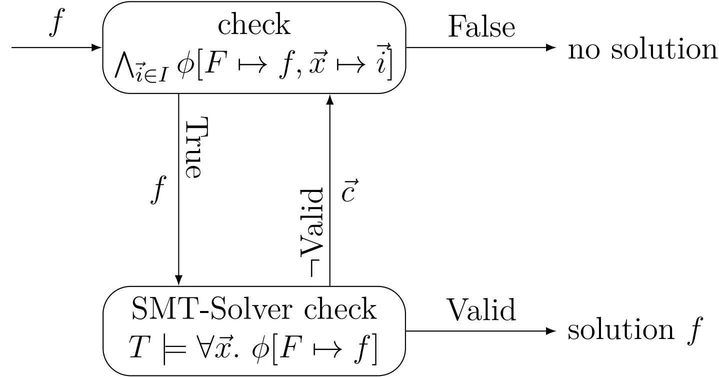


Figure 2.2: The procedure for checking if a given candidate f is a solution for a SyGuS problem (T, G, ϕ, F).

conjunction is consistent, we use the SMT solver to check for validity. In case of failure, this call produces an assignment (i.e. a counter model) $\vec{c}$ which we add to the set I. The verification procedure for a candidate solution f is depicted in Figure 2.2. Note that this technique is standard practice for state-of-the-art SyGuS solvers and is not a contribution of this thesis. In subsequent sections, we will simply refer to this setup as the *verification* procedure.

2.3 Program Analysis

A major part of this thesis considers the application of function synthesis to program analysis and in particular termination analysis. We make use of many standard techniques from static analysis that we briefly introduce here. These techniques have been around for a long time and are considered common practice.

We will be using code snippets that roughly follow the Java syntax. The reason for that is that the experiments are conducted using Java and its Bytecode. In the snippets presented in this thesis, to increase readability we will not use class or method declaration syntax. Unlike the Java standard, we will assume that the type `int` is the type of *unbounded* integers. We also use the keywords `or` as well as `and` instead of `||` and `&&` to denote logical disjunctions and conjunctions in code. Furthermore, we introduce a non-deterministic value $\star$. Hence, the expression

```
int x = ⋆
```

is an assignment of an unspecified integer to x. We will use this to denote variables whose values are given as arguments to the function and where no assumptions can be made. As an example, compare the snippet in Figure 2.3a compared to the corresponding Java class and method in Figure 2.3c. Finally, Figure 2.3b shows Java Bytecode corresponding to the Java program.

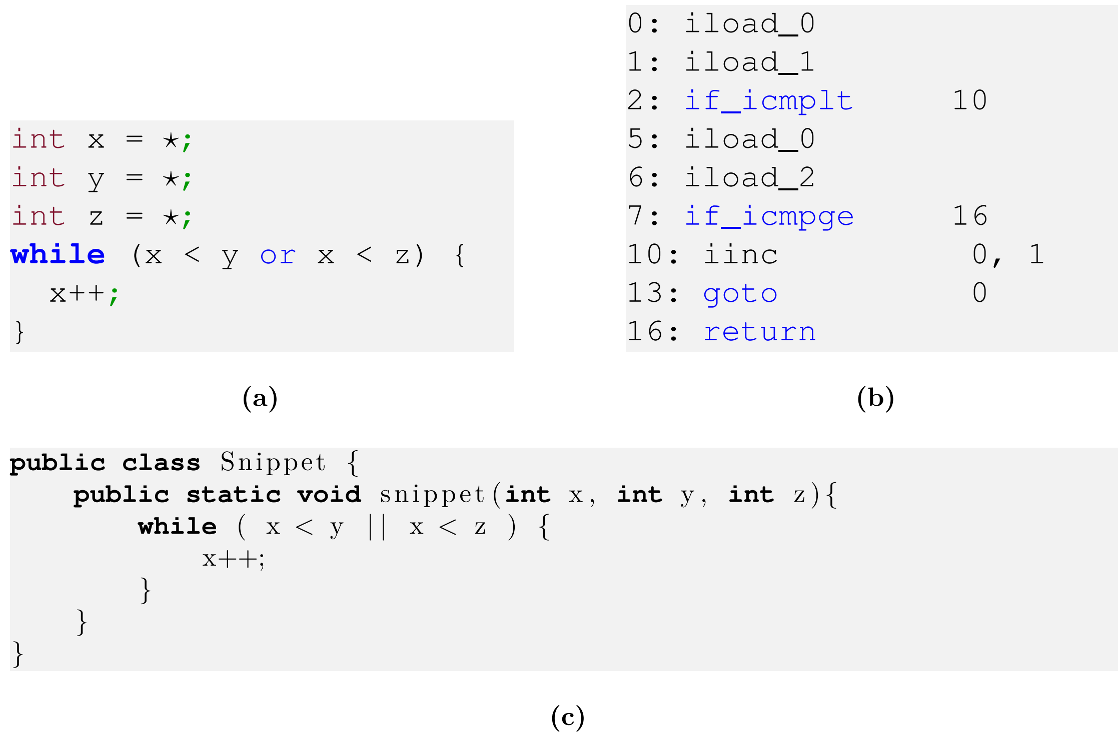


Figure 2.3: Figure 2.3c, shows the Java class and function represented by the snippet shown in Figure 2.3a. Finally, Figure 2.3b shows the Java Bytecode we obtain by compiling the Java class in 2.3c.

2.3.1 Modelling Programs

In order to reason about a program, we want to be able to express its behaviour abstractly. Hence, we define a *program state* s of a program P as a pair $s = \langle l, v \rangle$ where l is a location in P and v is a valuation function assigning values $v(i)$ to each variable i in P. Within the Java context as mentioned above, it is helpful to think of the locations as locations or lines in Bytecode. When analysing programs, we will implicitly use operational semantics [29] and the *program transition relation* induced by it. A *program run* is a possibly infinite sequence of states $s_0, s_1, \ldots$ such

that s_0 is a legal start state and $(s_i, s_{i+1}) \in T$ where T is the program transition relation. The location l_0 of a *legal start state* or *initial state* $s_0 = \langle l_0, v_0 \rangle$ has to be the start location of the program. The valuation v_0 is the initial valuation and can be thought of as the *program arguments*.

To analyse and model programs and their behaviour, it is not uncommon to transform them to different alternative representations. Usually, one can analyse the *control flow graph* [30] of a program or transform it to *single static assignment* [31] form. These constructions and translations are standard practices that have been in use for decades [32] and are implemented in many libraries that we make use of.

Loop Head Transitions: Let S be the set of all states of a program P and let $T \subseteq S \times S$ be the corresponding program transition relation. We define the set of *loop heads* H to consist of all locations of loop guards in the program. We will call H the set of all loop head locations. Since the set of locations is finite, the set of loop head locations is finite as well. Further, we define a loop head transition relation as follows:

Definition 2.3.1 (Loop Head Transition)**.** Let H be the set of all loop heads of a program. We define the loop head transition relation $T_l \subseteq S \times S$ for the loop head $l \in H$ such that

$$(\langle l, v_0 \rangle, \langle l, v_{k-1} \rangle) \in T_l$$

if and only if there exists a program run

$$\ldots, \langle l, v_0 \rangle, \langle l_1, v_1 \rangle, \ldots \langle l_{k-2}, v_{k-2} \rangle, \langle l, v_{k-1} \rangle, \ldots \quad \text{where} \quad l_i \neq l \quad \text{for} \quad 1 \leq i \leq k-2.$$

We will write T_H as a shorthand to denote all loop head transition relations. In particular, if H is the singleton set $\{h\}$ then $T_H = T_h$. We use the notation $T_h(x, y)$ to denote $(x, y) \in T_h$.

Informally, the loop head transition relation defines the transition from one iteration of a loop to the next iteration of the same loop. Importantly, the loop head transition relation is vital in termination analysis, as we will see soon.

```
int x = ⋆;
int y = ⋆;
int z;
z = y;
while (x > z) {
  x = x - 2;
}
```

(a) Single while loop with 3 variables, decreasing one of them in each iteration.

```
(assert (= J9 (> x z)))
(assert (= x1 (- x 2)))
(assert (=> J9 (= x' x1)))
```

(b) SMTLIB assertions modelling the loop head transition relation.

Figure 2.4: Part of a java function with corresponding bytecode.

Loop Summaries: Loop summaries are a set of constraints or logical formulas that capture the behaviour of the loop head transition relation T_H. As an example, consider the loop snippet in Figure 2.4a. Using the first-order theory of linear arithmetic, we can describe the loop head transition relation $T_H(x, y, z, x', y', z')$ where x, y, z and x', y', z' are the variable assignments (i.e. states) before and after the transition respectively:

$$
\begin{aligned}
T_H(x, y, z, x', y', z') :=& \\
& b = x > z \wedge \\
& x_1 = x - 2 \wedge \\
& b \implies x' = x_1
\end{aligned}
$$

In Figure 2.4b, we show an equivalent formula using the syntax of SMTLIB. The SMT formula was generated automatically. Hence, the variable names are an artefact of the single-static assignment encoding and the contrived formulas are an artefact of the procedure.

Auxiliary Invariants: During loop summarisation, we construct invariants from the loop bodies. However, in many instances, we also require invariants which originate outside of the loop body or guard. An *auxiliary invariant* A_h for a loop with loop head $h \in H$ is an invariant that holds before entering the loop and remains true during the execution of the loop. We use the shorthand notation A_H to denote a set of auxiliary invariants for all loop heads in H. To generate a set of auxiliary

invariants A_H we use simple heuristics by considering, for example, conditional statements, assignments, etc. We also use an SMT solver to check that these are consistent with respect to the loop summary. As an example, consider the loop and its corresponding loop head transition relation T_H shown in Figure 2.4. Notice that T_H does not relate y to y' or z to z'. Furthermore, it does not tell us anything about the relation between y or z. This is because neither of these variables gets changed within the loop body, and they are therefore not included in the loop summary. Nevertheless, it is important to know that neither y nor z change in the loop and that y gets assigned to z before the loop. We therefore add the auxiliary invariant

$$\begin{aligned} A_H(x, y, z, x', y', z') := & \\ & z = y \wedge \\ & y = y' \wedge \\ & z = z' \end{aligned}$$

In the subsequent sections, we will use $T_H(s, s')$ to refer to the first-order formula describing the transition from state s to s' and $A_H(s)$ to refer to the auxiliary invariants for the loop H expressed in terms of all variables in the formula. Note that the conjunction $A_H(s) \wedge T_H(s, s')$ could also be considered an *inductive invariant* for the loop head transition relation.

2.3.2 Program Termination

We say a program is *terminating* if all its runs are finite, and non-terminating if there exists a run of infinite length. Importantly, non-termination can only occur within loops, which is why the loop head transition T_H plays an important role in termination analysis. This relation is made formal in Theorem 2.3.1.

Theorem 2.3.1 (Non-Termination in Terms of Loop Head Transitions)**.** If a program P with loop heads H is non-terminating then there must be some $h \in H$

and an infinite program run $s_0, s_1, s_2, \ldots$ with an infinite subsequence[1] $s_{i_0}, s_{i_1}, s_{i_2}, \ldots$ such that $T_h(s_{i_j}, s_{i_{j+1}})$ for all s_{i_j} in the subsequence.

This theorem can be constructed from the proof in [33, Theorem 7.14] or the definition of Hoare'e *while* rule [12] to total correctness [34]. As a consequence, to show termination of P with loop heads H, it is sufficient to show that there cannot be an infinite subsequence as in Theorem 2.3.1. The standard technique of doing so is by means of providing a ranking function. To this end, we will use the notion of a well-founded relation.

Definition 2.3.2 (Well-founded relation)**.** Let $R \subseteq X \times X$ be a binary relation on X. We say R is a well-founded relation on X if and only if every non-empty subset of X has a minimal element with respect to R.

Notably, in literature one can find many different and equivalent definitions of well-founded relations which have been studied and formalised [35–37]. The most natural example of a well-founded relation is the set of natural numbers with $>$. In the context of linear integer arithmetic, it is worth noting that $(\{x \in \mathbb{Z} : x \geq k\}, >)$ for some fixed k is also well-founded. In addition, consider the lexicographic order defined as follows

Definition 2.3.3 (Lexicographic Order)**.** Let $(Q_0, \succ_0), \ldots, (Q_{n-1}, \succ_{n-1})$ be ordered sets with $Q = Q_0 \times \cdots \times Q_{n-1}$. We define the lexicographic order $>_{\text{LEX}} \subseteq Q \times Q$ such that $x >_{\text{LEX}} y$ if and only if there exists an $i \in \{0, \ldots, n-1\}$ such that

$$\forall j < i.\ x_j = y_j, \text{ and}$$

$$x_i \succ y_i.$$

The following theorem provides us with an additional important well-founded order when considering the lexicographic order.

[1] A subsequence of a given sequence is a sequence that is constructed by *only* deleting elements of the given sequence. If all elements in the subsequence are consecutive in the original sequence, we call it a substring.

Theorem 2.3.2 (Well-foundedness of $>_{\text{LEX}}$)**.** If $(S_0, \succ_0), \ldots, (S_{n-1}, \succ_{n-1})$ are well-orders then $>_{\text{LEX}}$ is a well-order on $S_0 \times \cdots \times S_{n-1}$.

More details about the product of lexicographic orders can be found in [35, Chapter 4]. Finally, having defined well-founded orders we can introduce the definition of a ranking function.

Definition 2.3.4 (Ranking Function)**.** Let $T \subseteq S \times S$ be a transition relation and let $(Q, \succ)$ be a well-founded relation. We say a function $f : S \mapsto Q$ is a ranking function for T if and only if

$$\forall x, y \in S.\ T(x, y) \implies f(x) \succ f(y).$$

Intuitively, ranking functions map program states to values that (i) decrease after every loop iteration and (ii) are bounded from below [17]. Combining Definition 2.3.4 with Theorem 2.3.1 we get Theorem 2.3.3, which shows how ranking functions are used to show termination of a program.

Theorem 2.3.3 (Ranking function shows termination)**.** Let T_H be the loop head transition relation of a program P. If there exists a ranking function f for T_H then P is terminating.

Hence, ranking functions are certificates of termination: if a ranking function exists, then the program terminates for every possible input. If a program has multiple loop heads, proving termination becomes more complex. In this case, many different techniques have been developed. One way, in particular, is by providing a lexicographic ranking function. In such cases, the ranking function will map to a vector of elements which have to decrease by the lexicographic order $>_{\text{LEX}}$.

3 Literature Survey

This thesis spans a wide range of topics as it seeks to apply machine learning to function synthesis. Both are active areas of research with large bodies of work preceding it. Furthermore, function synthesis has a wide range of applications one of which – termination analysis – will be the centre of focus in later chapters of this thesis. In this chapter, we discuss and summarise research that is related to the work presented in this thesis. The purpose of this is to position the contributions of this thesis within the context of other research and to emphasise the novelty of said contributions. Hence, we will present related work in function synthesis and termination analysis and draw our focus to prior research that applies machine learning to either of those fields.

At the end of subsequent chapters, we will reiterate *closely* related literature. These sections contain more details about related work with comparisons to the contributions of the respective chapter in order to highlight their novelty.

3.1 Function Synthesis

As mentioned in the introduction, function synthesis has already been used for circuit synthesis by Alonzo Church [38] in the sixties of the last century. A couple of years later Waldinger and Lee [39] as well as Green [40] showed how

theorem proving using resolution proofs can be applied to program synthesis. Subsequent work sought to apply new results in computational logic to the theorem proving setup [41–43]. In [44], a comparison of some of these synthesis methods is conducted. Nowadays synthesis has evolved into multiple rich sub-fields of study with applications in many different areas. This is, in part, due to the fact that the term "function synthesis" is very generic and may incorporate anything from theorem proving in the context of type theory to automatic data wrangling in spreadsheets. A survey article by Gulwani et al. [45] characterises synthesis techniques using the following three dimensions:

User Intent: This dimension describes the means by which a *specification* for a synthesis target is given. Examples of different formats include logical specifications, I/O examples, natural language, etc.

Search Space: The *space* of functions or programs where the potential *solutions* lie. In the case of program synthesis, this can be defined by the syntax of a programming language. In function synthesis, this can be a set of expressions defined by a context-free grammar.

Search Technique: The search *method* that is employed to *search* through the *space* of candidate solutions to find a function that *satisfies* the given *specification*. Some synthesis engines employ enumerative solvers that systematically list all programs in the *search space*. Other techniques include declarative methods that follow the divide-and-conquer principle and, most recently, neural-guided searches that use different AI methods.

Past research has been carried out in all of these dimensions and we recommend the aforementioned survey for a detailed presentation thereof. In addition to program synthesis, there has been a surge in "synthesis adjacent" tasks, such as program repair or automatic generation of documentation, for example. In line with the main topic of this thesis, in the following, we will look at work that applies some form of machine learning to synthesis and synthesis adjacent problems.

3.1.1 Programming-By-Example

This is a domain of program synthesis where the specification, that is *user intent*, is given in the form of a list of input-output examples (I/O examples). This has found plenty of applications in string editing, as well as data manipulation and wrangling [46]. The format is also very conducive to spreadsheet formula generation such as Excel's Flash Fill [5, 47–49]. PBE has also been applied to functional programming with data structures [50]. In [51], I/O examples are used in a refinement step to reduce the search space. One advantage of the PBE domain is the fact that training data is abundant and it is easy to automatically generate arbitrary amounts of additional training data by first generating functions and then generating I/O pairs. As a result, many machine learning methods that rely on large amounts of training data have been applied to PBE. Notable machine learning models used in this domain include Seq2Tree models [52], Recurrent Neural Networks (RNN) [53], Graph Neural Networks (GNN) [54], Recursive-Reverse-Recursive Neural Networks (R3NN) [55], and Multi-Layer Perceptrons [56]. In the domain of synthesis from sketches (i.e. programs with holes) [57, 58], LSTMs that encode I/O examples have been used to guide the search for solutions [59]. In [60], the authors train weights for probabilistic grammars. A similar approach is used in [61] where a Markov-Chain-Monte-Carlo algorithm is employed to sample programs from the probabilistic grammar. In both cases, the weights are trained from I/O examples. Usually, deep learning models are employed in combination with more traditional symbolic methods. For example, DeepCoder [62] uses deep learning to predict what functions are likely to be used in the synthesis target. In [63], the authors use reinforcement learning to guide a deductive synthesis algorithm. Similarly, the authors in [64] also use reinforcement learning to guide a synthesis engine for string editing programs as well as 2D and 3D graphics programs. In the experimental evaluation, the authors use the same PBE-style string editing problem set introduced in [65] where the authors apply machine learning to library learning. Library learning is a task where one seeks to iteratively synthesise a set of useful functions (i.e. library) from a set of given primitives. This task has also

been tackled by DreamCoder [66] using a wake-sleep set-up to iteratively synthesise library functions using neural networks. In [67], the authors investigate the use of machine learning models for syntax checking that run alongside the synthesis procedure using neural networks. The authors also make use of an RL set-up. In bottom-up synthesis procedures, intermediate information can be generated during search. This is exploited in BUSTLE [68], where neural networks are combined with bottom-up program search. This is evaluated on string manipulation problems which lend themselves to the generation of intermediate results. Following the bottom-up search approach, CrossBeam [69] uses different neural architectures to iteratively select operations and argument lists to iteratively combine smaller programs into one larger program. The selection process is done given I/O examples in conjunction with the operations. One of the advantages of bottom-up synthesis is the fact that intermediate steps form complete programs and their behaviour (e.g. I/O examples) can therefore be used to guide the next steps. This fact is utilised in [70] which was later extended to lambda terms with lists [71]. In [72], the authors present a Monte-Carlo simulation based algorithm to inductively synthesise programs by combining primitive functions with arguments. This work is related to the contents of Chapter 4. However, the experimental evaluation only presents the synthesis of a single function, quick-sort. Furthermore, the neural networks used to guide the search are once again trained using input-output examples. Recently, the PBE domain was extended to include *implication constraints*. These have been applied to predicate synthesis where I/O examples are given as positive and negative examples. In this context, implication constraints are of the form *if x is a positive example then so is x'*. The paper [73] describes the synthesis of logical predicates utilising neural networks in a similar way as we do in Chapter 6. Notably, the authors train a neural network with a loss function that is adjusted to include implication constraints. Subsequently, the neural network is translated to a symbolic expression representing the synthesised predicate. This technique is extended to predicates on lists defined by recursive functional programs [74]. In the experimental evaluation, the authors synthesise 11 predicates including `sorted` and `max`.

It is evident that the programming-by-example subset of program synthesis has been addressed by many AI based methods. These have proven to be very effective. This is good news, given the wide range of applications of the PBE domain. However, there are important applications of synthesis, especially in program verification and mathematics, where specifications are abstract and usually stated using some logic. These generally, cannot be transformed to a PBE domain. In this thesis we will exclusively consider such domains.

3.1.2 Synthesis of Equivalent Programs

In some applications of synthesis, a program or function is already given, and the goal is to synthesise an equivalent program. The most obvious example of an application would be compilers. Compilers can be considered as procedures that given a program in a high-level language, e.g. C++, synthesise a program in a low-level language like assembly, LLVM, or binary. In the case of source-to-source compilation, both target and source language could be high-level. Similarly and related to compilers is the area of optimisation and super-optimisation where given a program one seeks to synthesise a program that uses fewer resources (e.g., memory, energy, or time) [75, 76]. Given the importance of good compilers and the difficulty of compiler optimisation, it is not surprising that machine learning has been applied in many cases [77–79]. In [80], the authors present a survey of different techniques and areas where ML has been applied to compilers. This has become even more relevant with the availability of different computer architectures. Furthermore, a specialised algorithm with the sole purpose of synthesising sorting functions has been able to find highly optimised code for sorting arrays of small fixed sizes up to 5 [81]. Recently, machine learning has been applied vectorization of loops. For example, NeuroVectorizer uses deep neural networks in conjunction with a reinforcement learning loop to synthesise programs with better vectorisation and interleaving factors [82] than the original programs. Similarly, techniques for lifting programs from a general purpose language to a domain-specific language (e.g. tensor computation DSL) have become popular. In [83] the authors apply

standard synthesis and verification techniques to lift FORTRAN code to a stencil DSL. Similarly, C2TACO [84] presents a technique for synthesising TACO code, which is a DSL for tensor operations.

The field of compiler optimisation that incorporates machine learning is vast and has a long history. We will refrain from additional in-depth analysis of related work in this area and instead refer to the detailed survey [80]. The synthesis domain where equivalent programs serve as specifications has the advantage that they can continuously query the reference program as an oracle. Furthermore, it is possible to rely on heuristics and properties derived from the syntax of the reference program. These are key distinctions to other synthesis domains that lead us to believe that its applicability to computer-aided verification and formal methods is limited.

3.1.3 Abstract Logical Specifications

As previously mentioned one of the main contributions of this work is a learning based search algorithm for SyGuS. Most state-of-the-art SyGuS solvers are based on Oracle Guided Inductive Synthesis (OGIS) [58, 85], which alternates between a search phase that enumerates the space of possible programs and an oracle that returns feedback on candidate programs. In this context, CounterExample Guided Inductive Synthesis (CEGIS) [20] is an instance of OGIS that leverages the efficiency of rewrite engines by using counter-examples of failed candidate programs to improve the speed of subsequent verification steps. This technique has become standard practice and we also make use of this in our tool (cf. Section 2.2.1). In addition, solvers also use other techniques to guide the search algorithm and prune the search space. These range from simple methods such as conflict clause learning [86] to utilising rewriting to avoid checking the same term twice [21]. In [87], the authors apply a divide-and-conquer approach in combination with CEGIS. A probabilistic approach is followed by Euphony [88] where a probabilistic grammar is trained in order to bias the search towards more likely solutions. In [89], the authors present a method that works with graph neural networks that are trained iteratively in a reinforcement learning loop. This technique is applied

to the restricted setting of circuit synthesis. Circuit synthesis is a domain of synthesising Boolean functions where reference programs are usually given. Hence, this setup can be thought of as "Synthesis of Equivalent Programs" as discussed before. In expression search, Monte-Carlo simulations have been used [90] as a search technique. Finally, an AlphaZero-style Monte-Carlo tree search algorithm in combination with tree neural networks (TNNs) has been applied to combinator synthesis and synthesis of Diophantine equations [91].

3.1.4 Natural Language and Pre-Trained Models for Code

Another common program synthesis task is the automatic generation of programs given a natural language specification [92]. In some cases, specifications can be posed using natural language in combination with I/O examples [93–95]. Importantly, problems where user intent is given using natural language have gained popularity with the advent of Large Language Models (LLMs). As a result, many pre-trained models (PTM) combining natural languages and programming languages have been developed. These include, but are not limited to, CodeBERT [96], CodeT5 [97], CodeGen [98], Codex [99], PaChiNCo [100], and Synchromesh [101]. With these models, program synthesis adjacent tasks such as code completion [7], variable naming [102], code translation, code search, code repair [103], automatic code documentation, and competition programming [104] have gained popularity [105]. An evaluation of such models based on their effectiveness across different programming languages can be found in [106]. In [6], LLMs are applied to Python programming tasks written in natural language designed to be solvable by entry-level programmers. In [107] and [108], the authors also consider the problem of synthesising programs from given natural language descriptions. They present a method that combines reinforcement learning with a pre-trained language model. LLMs have also been applied to the previously discussed PBE synthesis domain. For example, in Jigsaw [109] the authors combine large language models with PBE synthesis problems. Similarly, [110] generate GPT queries from I/O examples to generate functions for string processing. Related to the previously discussed area of

compiler optimisation, [111] present a pre-trained model for compiler optimisation. In particular, they aim for code size reduction. Many pre-trained models have up to billions of parameters trained on large data sets. In contrast, Joshi et al. [112] describe significantly smaller (60 million parameters) models trained on Excel spreadsheet formulas with significantly less training data. This model is applied to formula repair, auto-completion, and syntax reconstruction. Even smaller neural networks in combination with syntactic enumeration are applied to last minute repair of low-code formulas as found in Excel [103]. Learning models and techniques for the aforementioned synthesis adjacent tasks are usually trained on large code data sets comprised of, for example, publicly available projects from GitHub. These data sets are even used to train simpler learning models on code token prediction [113]. It is evident that, with the improvement of LLMs, their applicability to code related tasks only benefits. Nevertheless, their applicability to formal methods remains to be seen. Most importantly, this pertains to domains where training data is scarce and "ground truth" is hard to come by. This is particularly problematic in synthesis problems within abstract domains, which are very common in computer-aided verification and logic.

3.1.5 Invariant and Condition Synthesis

Invariants, pre-conditions, and post-conditions are simply functions that return Boolean values (i.e. predicates). Usually, in program analysis, one seeks to find predicates that are true throughout the entire program or at certain parts of the program. For example, loop invariants are predicates that hold throughout the execution of a loop while pre-conditions and post-conditions are predicates that are true before, respectively after a certain block of code. Since invariants and conditions are simply functions that return Boolean values, they can be formulated as function synthesis problems. Usually, these synthesis problem specifications consist of a logical description of a program, and the goal is to find a set of predicates that are non-trivial. Non-trivial here can mean "strong enough" to prove a certain property of the program. Automatic invariant generation (i.e.

synthesis) is an active field of research that seeks to apply many different techniques from theory and practice. We will only survey closely related work with a focus on techniques that apply machine learning to invariant synthesis. The research described in Chapter 5 is closely related to [114] where invariants for Horn clauses are synthesised using Syntax-Guided Synthesis. The invariants have the form of inequalities and the specification is intended to prove termination of loops. Machine learning has also been applied to invariant synthesis. For example, in [115], the authors use handcrafted predicates in conjunction with decision tree learning to obtain pre-conditions for C functions. Decision trees are also used in [116, 117], and [118]. In the latter case, the decision tree is built over a set parameterised predicates. As a default set of predicates, the authors propose inequalities over the program variables. In the end, the learning algorithm learns Boolean combinations of the predicates as well as bounds on variables. Decision trees are used in a similar way in [119] but the predicates are learned using a linear classifier. In [120], random sampling and CEGIS are used in conjunction with Support Vector Machines (SVM) to synthesise invariants. The SVMs are used as classifiers to separate the positive and negative samples. In a similar vein, [121] uses SVMs to synthesise interpolants. More recently, graph neural networks with reinforcement learning have been applied to invariant synthesis [122, 123]. Since invariant synthesis suffers from data scarcity, the authors take existing loop invariant specifications and mutate them in ways that are guaranteed to keep the solution the same. This allows them to artificially generate more training data for the neural networks. As mentioned previously, [73] also applies neural networks to invariant synthesis using the PBE domain. Similarly, in [124], the authors use gated continuous logic networks (G-CLNs) to learn SMT formulas from traces. Most recently, LLMs have also been applied to invariant synthesis [125]. In [126], a reproducibility study is performed by creating a framework that allows for a comparison of multiple invariant generation methods. Finally, in [127] the authors present an AlphaZero-style agent to synthesise invariants using graph transformer networks.

In conclusion, invariant synthesis is an important problem in computer-aided verification. The field can be viewed as a domain of function synthesis and many people have successfully applied function synthesis techniques to it. Much like synthesis of equivalent programs discussed before, the specifications often include a full program or loop for which invariants need to be synthesised. This has the advantage that heuristics and information can be obtained from these specifications that may not be available in the more general setting of function synthesis. Utilising this domain specific knowledge is something we will also explore in this thesis.

3.1.6 Summary

In the above, we have described multiple synthesis techniques and categorised them by their specification type. We also briefly looked at large language models and how they are being applied to problems that are not specifically program synthesis but closely related. In presenting related research, we pointed out specific work that is closely related to the contributions of this thesis in terms of area of application or applied technique. These involved machine learning based techniques for functions and program synthesis. Table 3.1 presents a visual categorisation of that work separated by specification type (i.e. user intent by the previously mentioned synthesis dimensions) and machine learning techniques utilised. Note that the table does not include, techniques that do not use machine learning models, LLMs for synthesis adjacent tasks (e.g. code documentation), and synthesis of equivalent programs. We previously discussed each of these but Table 3.1 is supposed to only show closely related work. Note that some entries in the table are duplicates as they may utilise and combine different techniques or specification types. This thesis makes novel contributions to the domain of synthesis with abstract logical specifications (shaded green and blue). In particular, the work presented in Chapter 4 fits into the light blue cells while the contributions of Chapter 6 can be placed into the green cell.

	Simple	NN	PTM	RL
PBE	[48, 61, 88]	[52–56, 59, 62–68, 70–74, 95]	[93, 109]	[63, 64, 72]
Natural L.	[92, 95]	[52, 54, 69]	[6, 93, 94, 100, 104, 107–109]	[107, 108]
Logical	[88]	[89, 91]		[89, 91]
Partial		[59, 74, 103]		
Invariants	[115–121]	[73, 122–124, 127]	[125]	[122, 123, 127]

Table 3.1: Related work that applies some form of machine learning to synthesis categorised in Programming-By-Example (PBE), Natural Language (Natural L.), abstract logical constraints (Logical), and partial programs (Partial). We differentiate the following AI techniques: simple learning models (e.g. Bayesian models like probabilistic grammars), neural networks (NN), pre-trained models (PTM), and reinforcement learning (RL).

3.2 Termination Analysis

This thesis also presents contributions to the field of termination analysis. As previously discussed, termination analysis is an important problem in theoretical computer science with a rich history. The problem of termination concerns many different models of computation [27, 128]. Although general termination in sufficiently powerful models of computation remains undecidable, theoretical research continues to describe decidable fragments. Such fragments are often defined by restricting the use of operations [129–134], or disallowing the use of variables (in the context of term rewriting) [135], and other means.

On the more practical side of things, termination analysis is an important topic in computer-aided verification. In particular, we are interested in proving that a certain program or function terminates. The aforementioned theoretical results in many cases have carried over to termination analysis tools that seek to automatically prove termination of programs and other models of computation.

Owing to the undecidability of the problem in general, most techniques restrict the scope of the analysis in some way. These restrictions include linear ranking functions and programs [136–138], semi-definite programs [139], semi-algebraic systems [140], or formulae drawn from specific SMT theories [141]. To deal with complex program loops, lexicographic ranking functions [142–144], piecewise ranking functions [145, 146], disjunctively well-founded transition invariants [147–150], and implicit ranking functions have been used [151]. Ranking functions have been synthesised symbolically, using Farkas' lemma and template-based guess-and-check strategies [114, 152]. More recently, the tool `DynamiTe` [153] uses SMT solving to discover ranking functions based on execution traces. To prove conditional termination, abstract interpretation by under-approximation and loop summarisation methods have been used [154–158].

Other methods perform termination analysis by translating programs to alternative models of computation and show that the resulting model is terminating. This requires a guarantee that termination of the translated program implies the termination of the original program. Models used for this purpose include term rewriting systems [159–161], constraint logic programs [162], recurrence relations [163], Horn clauses [114], and Büchi Automata [164, 165]. This approach has also been used to show termination of programs that make use of cyclical data structures [166].

3.2.1 Machine Learning for Termination Analysis

In the last years, several termination analysis approaches that incorporate machine learning technologies have been presented. Early methods learn linear ranking functions from execution traces by constructing a linear regression problem whose solutions describe a loop bound [167]. Recently, machine learning models such as Support Vector Machines (SVM) have been used for termination analysis [168, 169]. Methods based on SVMs have been applied to single or nested loop programs defined using conjunctions of continuous functions for the guard, and deterministic assignments defined as continuous functions [169]. In [170], decision trees in

combination with CEGIS is used to synthesise piecewise linear ranking functions with constraint solving. Recently, deep learning was used in [171] to synthesise ranking functions for a setting with restricted loops. A similar data-driven approach employed efficiently checkable templates to learn loop bounds [172]. The authors propose a portfolio of methods and templates for this purpose. A heuristic approach to termination analysis based on deep learning has been proposed in [173]. This approach uses graph neural networks on a program's abstract syntax tree to estimate the likelihood of termination and non-termination. Using an attention mechanism, this method also produces line numbers that are likely to cause non-termination. This approach is envisioned to be part of a debugging workflow where potential issues are highlighted for consumption by a programmer or another analyser. Therefore, this method does not provide any formal proof of termination or non-termination. A standard technique for non-termination analysis is the generation of recurrence sets [174]. Intuitively, these are sets of states which can be proven to form a "circle" in the program execution. Recently, decision trees have been used to generate these recurrence sets [175]. The recurrence set is represented using the decision tree which is iteratively trained using feedback from an SMT solver. After each training iteration, the SMT solver is queried as an oracle to check if a correct recurrence set has been synthesised.

Finally, when considering dynamical systems, neural networks have been used to prove stability [176], using Lyapunov functions. Similarly, in the setting of probabilistic programs neural networks have been used to prove almost-sure termination [177].

4

Reinforcement Learning in Syntax-Guided Synthesis

As defined in Chapter 2, syntax-guided synthesis (SyGuS) is a synthesis setting where users provide a background first-order theory, a specification as a semantic constraint, and a context-free grammar as a syntactic constraint. However, not being able to make strong assumptions about the type of syntactic or semantic specification makes for a difficult target for machine learning where the main issue is data scarcity. The inherent difficulty of function synthesis also makes it hard to artificially generate training data. Hence, careful consideration is required in how and where to apply machine learning, as a wrong method may not be feasible due to the lack of data. In this chapter, we offer the following two solutions:

- Monte-Carlo based tree search algorithm for solving SyGuS problems
- Generating new SyGuS problems from pre-existing SMT problems.

For the first part, we describe function synthesis as a *single player game* with states, actions, winning conditions, and a game tree that corresponds to the SyGuS grammar. This allows us to regard the function synthesis procedure as an agent traversing the game tree searching for winning states, where each state is a node representing an expression in sentential form in the language of the given grammar.

In this setting, we can employ “smart” tree search techniques developed in AI research to explore the enormous state space. We use a Monte-Carlo based tree search similar to that used by AlphaZero [22] where machine learning models are used to guide a search agent through the tree, prioritising branches that are more likely to lead to a correct solution. These machine learning models predict *policies* and *values* of states and actions in the game. The policy is a function that estimates the likelihood of success when choosing an action in the given state. Similarly, the value estimates the *quality* of a given state regardless of action. While searching through the game tree our algorithm uses the policy and value to calculate the upper-confidence bound for trees (UCT) of each node we reach. The UCT is used to balance the exploitation of learned policies and values, with the exploration of new branches. We use simple syntactic features to describe states and actions so that we can apply this technique to SyGuS theories with abstract specifications like LIA. The machine learning models are iteratively trained through a reinforcement learning loop from data gathered in the previous iterations. This has the advantage that we do not require a priori labelled training data.

The standardised problem sets for SyGuS are limited. We hypothesise that this is due to the relative immaturity of the field. Hence, to further mitigate the data scarcity issue, we present an algorithm to generate SyGuS problems from pre-existing satisfiability modulo theories (SMT) problems. This is done by utilising the fact that SyGuS and SMT problems share a similar syntactic framework in conjunction with applying first-order unification and anti-unification. We apply this method to the pre-existing `LIA` and `QF_LIA` SMT benchmarks to generate new SyGuS problems. Finally, we use these problems in combination with standard benchmarks from the SyGuS competition [23] for an experimental evaluation of the developed techniques.

In particular, the main contributions of this chapter are:

- We frame SyGuS as a tree search problem and present a Monte-Carlo tree search (MCTS) based synthesis algorithm. This algorithm uses machine learned policy and value predictors and the Upper-Confidence Bound for Trees (UCT) for balancing exploration and exploitation.

- We train policy and value predictors using data generated from a reinforcement learning loop, where data from all previous iterations is collected to train the models for the current iteration. These predictors work for abstract logical specifications and do not restrict the use of syntactic specifications.

- We present a method for generating SyGuS problems from SMT problems using anti-unification and unification in order to overcome a lack of training data. This approach is applicable to generating training data for synthesis problems with logical specifications.

- We evaluate the algorithm and the reinforcement learning setup on a combination of pre-existing benchmark sets from the SyGuS competition [23] as well as our newly generated problem set. Both consist of problems that use abstract logical specifications.

4.1 Background

In this section, we briefly introduce some background that is specific to this chapter. None of the topics discussed in this section are novel contributions of this thesis and are considered standard procedures in their respective fields.

4.1.1 Decision Tree Models

Decision trees are one of the simplest machine learning models that can be used for classification as well as regression problems [178]. In the latter case, they are also known as regression trees. They fall under the category of supervised models where the training data has to be labelled. This means that a set of input/output pairs called ground truth needs to be given as training data. Decision trees can be thought of as mechanised versions of a series of nested *if-then-else* queries. Like other models, regression trees are functions mapping an input, called *feature vector*, to a real number.

Gradient Boosted Trees: Boosting is a technique for combining multiple learning models, called weak learners, into one model. In the case of tree models as weak learners, the resulting model is called a gradient boosted tree [179]. The prediction of such a boosted model can be described as a combined prediction of the weak learners. One simple method for combining these weak learners is by using a weighted average. Hence, the prediction of a gradient boosted tree model can be seen as taking a weighted average of the predictions of a set of regression tree models. During training the weak learners are trained alongside the weights with which the average is taken. As usual in machine learning, in the training phase, a certain loss function is being minimised along the training data. In our experiments, we make use of XGBoost [180], a library for highly performant gradient boosted trees where the default loss function is the *root mean square error*.

4.1.2 Unification and Anti-Unification

Unification and anti-unification are problems commonly arising in different areas of computer science [128, 181, 182], including function synthesis [183]. The following definitions as well as a more detailed analysis of unification can be found in [184, 185] while anti-unification is discussed in more detail in [186–188].

Let Σ be a signature containing, constant symbols, and functions, and we let $\mathcal{V}$ be the set of first-order variables. We let $\mathcal{T}(\Sigma, \mathcal{V})$ be the set of first-order terms inductively defined from Σ and $\mathcal{V}$. A *substitution* is a mapping $\sigma \colon \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ with a finite domain and its application to a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, written $t \cdot \sigma$, is defined inductively:

$$t \cdot \sigma = \begin{cases} \sigma(t) & \text{if } t \in \mathcal{V} \\ f(t_0 \cdot \sigma, \dots, t_{n-1} \cdot \sigma) & \text{if } t = f(t_0, \dots, t_{n-1}) \end{cases} \tag{4.1}$$

We allow f to be a constant symbol without arguments in the second case leading to $f \cdot \sigma = f$. Now, we can introduce the unification problem for first-order terms.

Definition 4.1.1 (Syntactic Unification)**.** A substitution σ is called a *unifier* of two terms $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$ if

$$s \cdot \sigma = t \cdot \sigma.$$

The Syntactic *unification* problem for two terms s, t, written $s \stackrel{?}{=} t$, asks for a unifier σ for s and t.

Example 4.1.1 shows a unification problem and the resulting unifiers. In the setting of first-order terms unification is decidable and practical [189].

For two terms $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$ we say that s is more general than t, if there exists a substitution σ such that $s \cdot \sigma = t$, written $s \lessapprox t$. This allows us to define the anti-unification problem as follows.

Definition 4.1.2 (Syntactic Anti-Unification)**.** A term $g \in \mathcal{T}(\Sigma, \mathcal{V})$ is called a generalisation of $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$ if there exist σ_0, σ_1 such that

$$g \cdot \sigma_0 = s \text{ and } g \cdot \sigma_1 = t.$$

Furthermore, we call g the least general generaliser (LGG) if for all generalisers g' of s and t we have $g' \lessapprox g$. Syntactic anti-unification describes the problem of finding a least general generaliser g for a set terms $T \subseteq \mathcal{T}(\Sigma, \mathcal{V})$.

Example 4.1.1 shows an instance of a least general generaliser of two terms. An algorithm for anti-unification was first presented in [190]. In the setting of first-order terms anti-unification, much like unification, is decidable and practical.

Example 4.1.1 (Unification and Anti-Unification)**.** Let us consider the signature Σ with constant symbols $\{c, 1, 3, 5\}$ and the binary infix function symbols $\{\odot, \ominus, \bullet\}$ together with the variables $\mathcal{V} = \{v, w, x, y, z\}$. Consider the following unification problem for the two terms in $\mathcal{T}(\Sigma, \mathcal{V})$:

$$(5 \odot x) \bullet (3 \ominus c) \stackrel{?}{=} (y \odot 1) \bullet z$$

Using the standard syntactic unification algorithm we obtain the unifier

$$\{x \mapsto 1, y \mapsto 5, z \mapsto (3 \ominus c)\}.$$

This can be verified by applying the substitution to each of the two terms individually resulting in the term $(5 \odot 1) \bullet (3 \ominus c)$ for both terms. On the other hand, if we apply anti-unification to the two terms

$$(5 \odot x) \bullet (3 \ominus c) \text{ and } (y \odot 1) \bullet z,$$

we obtain the term

$$w \bullet v$$

where w and v are variables not occurring in any of the original terms. This term is called the "least general generaliser" (LGG). By applying unification to the LGG and either of the terms above we obtain a substitution that turns the LGG into the term we unify with. For example, unifying $(5 \odot x) \bullet (3 \ominus c)$ with $w \bullet v$ gives us the substitution $[w \mapsto (5 \odot x), v \mapsto (3 \ominus c)]$ which, when applied to the LGG gives us the term we unified with. Note that the LGG is *always* unifiable with the terms that were anti-unified.

4.2 Motivating Example

```
(set-logic LIA)

(synth-fun F ((x Int) (y Int)) Int
  ((I Int) (B Bool))
  ((I Int (x y 0 1 (+ I I) (- I I) (ite B I I)))
   (B Bool ((and B B) (or B B) (not B) (=  I I) (<= I I)
      (>= I I))))))

(declare-var x Int)
(declare-var y Int)

(constraint (>= (F x y) x))
(constraint (>= (F x y) y))
(constraint (or (= x (F x y)) (= y (F x y))))

(check-synth)
```

Figure 4.1: `SyGuS-IF` problem for a function computing the maximum of two numbers.

As per Section 2.2.1, a SyGuS problem is a tuple (T, G, ϕ, F). Using the SyGuS input format (SyGuS-IF), Figure 4.1 shows a synthesis problem for a function F that computes the maximum of two integers. First, we declare that we are working in the logic $T = \texttt{LIA}$ with a function variable F that takes two integers

as arguments and returns an integer. Further, we stipulate that the synthesised function must be in the language defined by the following grammar G:

$$I \rightarrow x \mid y \mid 0 \mid 1 \mid (I + I) \mid (I - I) \mid (\text{ite}\ \ B\ I\ I)$$
$$B \rightarrow (I \wedge I) \mid (I \vee I) \mid (\neg I) \mid (I = I) \mid (I \leq I) \mid (I \geq I)$$

Semantically, we want that for all x and y, $F\ x\ y$ needs to be larger than or equal to both, x and y. Furthermore, $F\ x\ y$ also needs to equal either x or y. Formally, we get the following constraint:

$$\begin{aligned}\phi = \forall x\ y.\ & F(x, y) \geq x \wedge \\ & F(x, y) \geq y \wedge \\ & (x = F(x, y) \vee y = F(x, y))\end{aligned}$$

Naive Solution: The simplest enumerative method for solving such a SyGuS problem would be to enumerate all terms constructed with $1, 2, 3, \ldots$ steps using the rules defined by G. In each step, one would have to split the terms into complete and partial functions (i.e. sentences and terms in sentential form as defined in Section 2.2.1). For each complete function f in each step, we use an off-the-shelf SMT solver to prove $\texttt{LIA} \models \phi[F \mapsto f]$. The problem with this approach is the combinatorial explosion of the number of terms after n steps. In fact, even a grammar as simple as G leads to 18582 terms after 5 steps and almost 20 million after 7. Table 4.1 shows the number of complete and partial functions of up to 7 steps. This exponential blow-up has been studied as early as 1963 by Chomsky and Schützenberger [191] and more recently in [192, 193]. Note that one canonical solution to this SyGuS problem is the term ite $(x \leq y)\ y\ x$ which can be constructed in 7 steps in the following way:

$$I \rightarrow \text{ite}\ \ B\ I\ I \rightarrow \text{ite}\ (I \leq I)\ I\ I \rightarrow$$
$$\text{ite}\ (x \leq I)\ I\ I \rightarrow \text{ite}\ (x \leq y)\ I\ I \rightarrow \text{ite}\ (x \leq y)\ y\ I \rightarrow$$
$$\text{ite}\ (x \leq y)\ y\ x$$

Depth	complete	partial	total
1	0	1	1
2	4	3	7
3	0	48	48
4	64	753	822
5	0	18582	18582
6	4096	541022	545118
7	30720	19239282	19270002

Table 4.1: Number of functions at depth up to 7 split into complete and partial programs with total being the sum of both. We start with the starting symbol of the grammar at depth 1.

We also need to take into account that, for each complete program, a call to an SMT solver is also required. It should become apparent that enumerating all potential solutions becomes infeasible very quickly. The main contribution of this chapter is a method based on machine learned models to prune this search space.

4.3 The Game of Function Synthesis

At the core of any enumerative SyGuS solver is a method for systematically listing all possible terms generated by a given grammar G. In our case, this is the leftmost reduction strategy (cf. Section 2.2.1). Starting with the starting symbol, we expand the leftmost non-terminal and replace it with the corresponding right-hand sides in G, resulting in a new set of terms. Some of the terms in the resulting set are complete functions with no non-terminals while others are partial functions containing non-terminals. For each of the partial functions, we can again expand the leftmost non-terminal to obtain a new set of functions. On the other hand, each of the complete functions is a potential solution candidate that can be checked using an SMT solver. Example 4.3.1 shows how this process enumerates the expressions of a given grammar.

Example 4.3.1 (Leftmost Enumeration procedure)**.** Consider the following gram-

mar from Section 4.2

$$I \rightarrow x \mid y \mid 0 \mid 1 \mid (I + I) \mid (I - I) \mid (\text{ite } \ B\ I\ I)$$
$$B \rightarrow (I \wedge I) \mid (I \vee I) \mid (\neg I) \mid (I = I) \mid (I \leq I) \mid (I \geq I)$$

with the start non-terminal I. Since at the start, we only have the start symbol, I, we get all right-hand expressions of the first rule in the first step of our enumeration:

$$x, y, 0, 1, (I + I), (I - I), (\text{ite } \ B\ I\ I)$$

This gives us four complete functions, x, y, 0, and 1 and three partial functions $(I + I)$, $(I - I)$, and $(\text{ite } \ B\ I\ I)$. To enumerate the next set of functions, we take the partial set and replace the leftmost non-terminal. Hence, for the term $(I + I)$ (the leftmost non-terminal is in red) we get the following set:

$$(x + I), (y + I), (0 + I), (1 + I), ((I + I) + I), ((I - I) + I), ((\text{ite } \ B\ I\ I) + I)$$

This works analogously for the terms $(I - I)$ and $(\text{ite } \ B\ I\ I)$. The only difference for the term $(\text{ite } \ B\ I\ I)$ is the fact that the leftmost non-terminal is B and we therefore have to apply the second rule resulting in the following set:

$$(\text{ite } (I \wedge I)\ I\ I), (\text{ite } (I \vee I)\ I\ I), (\text{ite } (\neg I)\ I\ I)$$
$$(\text{ite } (I = I)\ I\ I), (\text{ite } (I \leq I)\ I\ I), (\text{ite } (I \geq I)\ I\ I)$$

This process can be succinctly described as a tree search. For that, we introduce the notion of a *Grammar Tree* in Definition 4.3.1.

Definition 4.3.1 (Grammar Tree)**.** Given a context-free grammar $G = (V, \Sigma, R, S)$, the grammar tree $\texttt{Tree}(G)$ is defined recursively as follows:

- S is the root (i.e. node with no parent)
- for every node r, u is a child of r if $r \rightarrow^{L} u$.

A node without child nodes is called a *leaf* node. We denote the set of all leaf nodes of $\texttt{Tree}(G)$ with $\texttt{Leaves}(G)$.

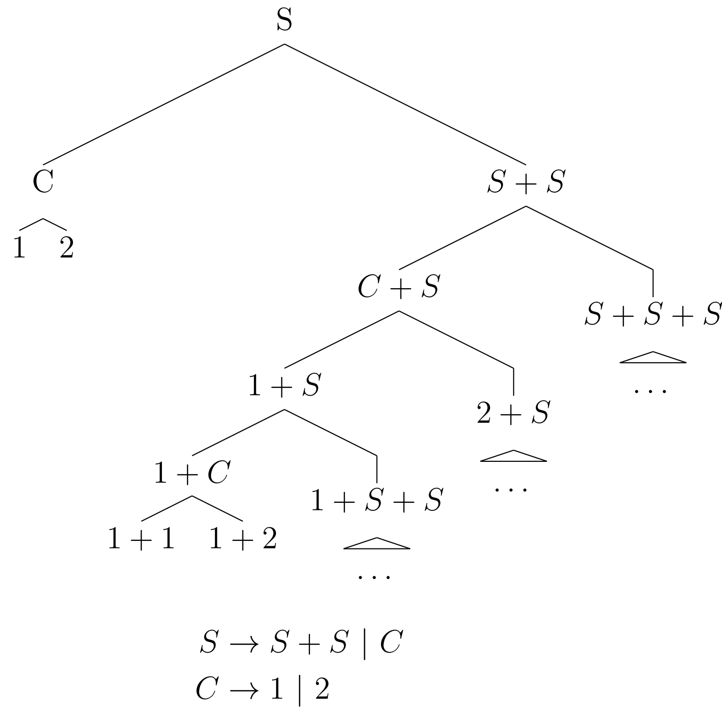


Figure 4.2: Grammar describing the set of terms of addition of 1 and 2 with the corresponding grammar tree.

Figure 4.2 shows an example of a simple grammar with a corresponding grammar tree. Formally, given a grammar $G = (V, \Sigma, R, S)$ and a corresponding grammar tree $\texttt{Tree}(G)$, the following theorem relates nodes in $\texttt{Tree}(G)$ to words in L^G.

Theorem 4.3.1. Let G be a context-free grammar with a corresponding grammar tree $\texttt{Tree}(G)$. The following relations between $\texttt{Tree}(G)$ and G hold.

1. Every leaf node of $\texttt{Tree}(G)$ corresponds to a term in L^G, and vice versa.

2. Every internal node of $\texttt{Tree}(G)$ corresponds to a term of G in sentential form, and vice versa.

Theorem 4.3.1 shows that we can systematically search for a term of a language L^G by conducting a tree search on $\texttt{Tree}(G)$. In the following, we will use a Monte-Carlo based method that was used by AlphaZero and subsequent work to search the grammar tree for correct expressions.

4.3.1 The Game

To describe our synthesis algorithm, we will use the notion of a game that is played on the grammar tree $\texttt{Tree}(G)$ of the SyGuS problem $P = \langle \tau, F, \phi, G \rangle$. Each game state consists of a static and a dynamic component. The static component is the specification ϕ which does not change throughout the game. The dynamic component consists of terms generated by G and therefore nodes in $\texttt{Tree}(G)$. Hence, any state s of the synthesis game is a pair $\langle \phi, v \rangle$ where v is a node of $\texttt{Tree}(G)$. The actions in state $\langle \phi, v \rangle$ correspond to the possible choices of child nodes of v in $\texttt{Tree}(G)$. If v is a complete function (i.e. leaf node) we have reached a final state and no further actions can be taken. If v is a solution to P, we have reached a winning state, otherwise the state is losing. Formally, we define the synthesis game as follows:

Definition 4.3.2 (Synthesis Game)**.** Given a SyGuS problem $P = \langle \tau, F, \phi, G \rangle$ and the tree $\texttt{Tree}(G)$ with vertex set V and edge set E, with leaf nodes $\texttt{Leaves}(G)$ the synthesis game is the tuple $(S, S_w, S_L, s_0, A_{s \in S})$ where

- $S = \{\langle \phi, v \rangle \mid v \in V\}$,
- S_w is the set of winning states $\{\langle \phi, v \rangle \mid v \in \texttt{Leaves}(G).\ \tau \models \phi[F \mapsto v]\}$,
- S_L is the set of losing states $\{\langle \phi, v \rangle \mid v \in \texttt{Leaves}(G).\ \tau \not\models \phi[F \mapsto v]\}$,
- $s_0 \in S$ is the start state $\langle \phi, r \rangle$ where r is the root of $\texttt{Tree}(G)$,
- $A_{s \in S}$ is the set of actions each leading from s to exactly one successor state $\langle \phi, c \rangle$ where c is a child node of s.

In subsequent sections, we will present an agent that incorporates machine learned heuristics for solving this game. These models are trained on representations of states $s \in S$. Crucially, these representations also include the static component ϕ, which is why we include it in each state even though ϕ never changes throughout the synthesis game.

4.4 Monte-Carlo Grammar Tree Search

Similar to [194] and [91], we adapt AlphaZero's [22] Monte-Carlo simulation based tree search algorithm. The input is a SyGuS problem $P = \langle \tau, F, \phi, G \rangle$ and the output is either a solution $f \in L^G$ that satisfies the constraints or a general failure (i.e. timeout). The algorithm traverses through the grammar tree of G and consists of four main phases: *Big-Step*, *Rollout*, *Expansion*, and *Backpropagation*. The big-steps are the outermost loop and represent the agent committing to a single step move from which backtracking is not possible. During rollout and expansion, the agent repeatedly searches for the first unexpanded node along the "best" path. Finally, backpropagation updates the statistics at the end of the rollout by propagating them back along the path that was taken.

During the search, we keep a record of the visit count $\mathrm{N}(v)$ and cumulative value $\mathrm{W}(v)$ of every node in the tree. The default visit count is 0. Note that we differentiate between the value $\nu(v)$ of a node v and the cumulative value $\mathrm{W}(v)$. The cumulative value is the accumulation of the node's value $\nu(v)$ with all values of the expanded subtree of v. These are accumulated and updated during the backpropagation phase. Hence, unlike the cumulative value $\mathrm{W}(v)$, the value $\nu(v)$ does not change during the search and can be thought of as a function in a mathematical sense. Similarly, we also have a policy function $\pi(e)$ for each edge e in the grammar tree. Policies and values are supposed to estimate the quality of an action and state, respectively. In the simplest case, these could be a constant or a simple heuristic. In Section 4.5 we will discuss policy and values in more detail as these are the points where we will employ machine learned heuristics. In the remaining, we also make use of the following functions in the pseudocode.

`most_visited_child`(v): This function returns the immediate successor of v that has the highest visit count among all immediate successors, with a random tiebreak.

`expand`(v) and `is_expanded`(v): These functions add the vertex v to the set of expanded nodes and check whether a given vertex v is in that set, respectively.

`verify`(P, v): Given a node v where f is the corresponding expression and a problem P, this function uses techniques discussed in Section 2.2.1 to check if f constitutes a solution to P. Formally, this function checks if $\tau \models \phi[F \mapsto f]\}$.

Furthermore, we set bounds on the iterations of the different loops. In particular, we limit the number of big-steps and rollouts by `MAX_BIGSTEPS` and `MAX_ROLLOUTS`, respectively. These can be viewed as hyperparameters of the algorithm. In detail, the four phases are as follows:

Big-Step: This phase is shown in Algorithm 1. This is the outermost loop of the search algorithm that is executed at most `MAX_BIGSTEPS` times. The search starts at the root of the grammar tree by setting the *active node* accordingly. In each iteration, we start the rollouts from the current active node, which in turn initiate the remaining phases. If a solution is found during the rollouts, we terminate and return the solution. Otherwise, once all rollouts are completed, we perform one *action* by updating the active node to the most visited immediate child (i.e. the action that was taken most often) for the next iteration. Intuitively, the most visited child is the root of the "most explored" subtree, and it therefore, makes sense to follow a path to that node in the next big-step.

Algorithm 1: `Big-Steps`

```
Data: SyGuS problem P = ⟨τ, F, φ, G⟩
Result: fail or solution to P
ac_node ← root(Tree(G));
for i ← 0 to MAX_BIGSTEPS do
    r ← rollout(ac_node, P);
    if r is solution then
        return r;
    else
        ac_node ← most_visited_child(ac_node)
return fail
```

Rollout and Expansion: In this phase, we perform rollouts and expansion steps starting from a given node in the grammar tree. The purpose of these phases is to collect as much data as possible about the subtree starting at `active_node`. To this end, we *check potential solutions* where appropriate, evaluate intermediate steps and apply *backpropagation* to update tree statistics so that Algorithm 1 can make the best decision possible in choosing the next node. The procedure for these steps is shown in Algorithm 2. Each time the procedure is called, we

Algorithm 2: `Rollout`

```
Data: active_node and P
Result: fail or solution to P
for i ← 0 to NUM_ROLLOUTS do
    current_node ← active_node;
    sub_path = [current_node];
    while is_expanded(current_node) do
        current_node ← best_successor(current_node);
        append(sub_path, current_node)
    if current_node ∈ Leaves(P) then
        if verify(P, current_node) then
            return current_node;
        else
            W(current_node) ← 0;
    else
        expand(current_node);
        W(current_node) ← ν(current_node)
    backpropagate(sub_path, W(current_node))
```

perform `NUM_ROLLOUTS` rollouts. During each rollout, we search for the first unexpanded node along the path of "best" successors in the grammar tree starting from `active_node`. Note that choosing the best successor means choosing the best action in the synthesis game (cf. Definition 4.3.2). In the simplest case, *best* successor is just a random selection of an immediate child node. In Section 4.5 we present a different notion of "best" to improve the search guidance that takes policy and value (i.e. $\pi(\cdot)$ and $\nu(\cdot)$) into account. If the first unexpanded node (i.e. `current_node`) is a leaf node, we check if the corresponding function is a solution to P. In the affirmative, we return the solution, otherwise, we set the cumulative value (i.e.

reward) of the node to 0. If the current node is an internal node, we expand the node. Expansion amounts to adding `current_node` to the list of expanded nodes so that `is_expanded(current_node)` is true. Further, we also set the cumulative value W(`current_node`) to the value ν(`current_node`) of the node. As the current node is the first and only node of its subtree to be expanded, the cumulative value is now equal to the value ν(`current_node`). Finally, at the end of every rollout we call the backpropagation procedure with the value of the `current_node` as well as the path taken from `active_node` to `current_node`.

Backpropagation: This is the final phase of each rollout. The purpose is to propagate the value of the newly expanded node up through the tree and to update the visit counts of each node that was visited along the way. The updating of these statistics is important as they are used to determine the "best" successor in subsequent rollouts (i.e. the return value of `best_successor(·)`). The pseudocode of this phase is shown in Algorithm 3. Note that we exclude the last node on the path as that is the node we expanded in the current rollout.

Algorithm 3: `Backpropagation`

```
Data: path and val
for node In path[:-1] do
    N(node) ← N(node) + 1;
    W(node) ← W(node) + val;
```

4.5 Search Guidance with Policy and Value

In the previous section, we used the notion of a "best" successor node to guide the search. Specifically, we used the function `best_successor`(v), which selects the best child node of v. In this section, we will make this notion of "best" explicit. In the simplest case, this may just be a random selection of child nodes. However, we introduce functions that are supposed to give better estimates of how likely a path leads to a correct candidate. We also take into account that these estimates can be mistaken. Therefore, we also introduce a heuristic to balance the *exploitation* of

paths with a high likelihood of success with the *exploration* of rarely visited parts of the search tree that might have erroneously lower success estimates.

4.5.1 Policy

The first function to estimate the quality of the search state is called policy. Intuitively, a policy estimates the quality of an action when starting from a given state. Hence, in the synthesis game (cf. Definition 4.3.2), a policy function is a function

$$\pi : S \times \bigcup_{s \in S} A_s \to \mathbb{R}.$$

The policy of a state-action pair represents the likelihood of success when committing to action a from state s. For readability, we will use the notation $\pi(s, s')$ where $s, s' \in S$ to denote the policy of the action a that leads from state s to s'. It should be noted that a policy should only evaluate legal actions in a given state, instead of all actions in the game. This behaviour can be simulated by letting $\pi(s, a) = 0$ for actions a that are illegal in state s.

4.5.2 Value

The second heuristic we use estimates the value of a state. Hence, the value function,

$$\nu(s) : S \to \mathbb{R}$$

map a state to a "quality" measure of that state. A perfect value function estimates 0 for all losing states S_L and 1 for all winning states $s \in S_W$ while giving the ratio of all *winning final states* to *all final states* reachable from the given state. In other words, the function should predict the probability of ending up in the winning state when starting from the given state.

4.5.3 Upper Confidence Bound for Trees

The most naive strategy for solving the synthesis game would be a (heuristic) best-first tree search algorithm, where we always select nodes and actions with the highest values and policies. However, if the value and policy are not perfect (i.e.

bias against subtrees that contain solutions) we might end up repeatedly exploring a path that does not lead to a successful node. Conversely, if we only look for unseen paths in the search tree, we do not use the information provided by the value and policy functions to make informed decisions, rendering them useless. Since perfect value and policy functions are impossible to achieve (cf. Chapter 2), we have to strike a balance between these two stages – *exploration* of rarely visited subtrees and *exploitation* of heuristics. That is, we want to explore relatively unknown parts of the tree to learn more information about their quality while also exploiting the information already obtained. To this end, we use the upper confidence bound for trees (UCT) [195] as a heuristic to find the "best" child node during the rollout phase. This heuristic combines the cumulative value and policy with the visit counts of the parent and child node to balance exploration and exploitation. Given a parent node $p \in S$ and child node $c \in S$ the UCT value of the child node when starting in the parent node is calculated as follows:

$$\text{uct}(p, c) = \frac{\mathrm{W}(c)}{\mathrm{N}(c)} + \gamma * \pi(p, c) * \sqrt{\frac{\log \mathrm{N}(p)}{\mathrm{N}(c)}} \tag{4.2}$$

The UCT is the sum of two values; the exploitation value $\frac{\mathrm{W}(c)}{\mathrm{N}(c)}$ which calculates the average value per visit of the child node and the exploration value $\pi(p, c) * \sqrt{\frac{\log \mathrm{N}(p)}{\mathrm{N}(c)}}$. The latter of which is multiplied with a constant γ that defines the relation between exploration and exploitation (i.e. if $\gamma = 0$ the exploration term is 0 and we only consider exploitation). The second term weights the policy $\pi(p, c)$ (i.e. probability of success when choosing an action that leads from parent p to c) with the term

$$\sqrt{\frac{\log \mathrm{N}(p)}{\mathrm{N}(c)}}. \tag{4.3}$$

When the visit count for p increases and the visit count for c stagnates, (4.3) increases and thus the exploration value also increases. Conversely, if the child c visit count increases comparatively to the parent visit count, the term decreases in value, and thus the exploration value decreases.

Recall the use of `best_successor(s)` in Section 4.4 as a function that is supposed to select the "best" child node of a given node s. We now define this to be the node with the highest UCT score among all child nodes. More precisely,

$$\texttt{best_successor}(s) = \underset{c \in \texttt{successors}(s)}{\arg\max} \ \mathrm{uct}(s, c),$$

where $\texttt{successors}(s)$ is the set of all child nodes of s. This selection criteria is used in the rollout phase of the Monte-Carlo tree search. Hence, this is the stage where we do rollouts without committing to any choice (i.e. without doing a Big-Step).

4.6 Features

To facilitate the use of machine learning models, we have to represent the inputs to these models as feature vectors. In our case, we want to use machine learning models to predict policy and value. Hence, we need to represent states and actions of the synthesis game using feature vectors. We use syntactic formula and term features similar to those that were developed and used in first-order logic theorem provers [194, 196–198]. Term walks constitute the most basic building blocks of our feature vectors. In the following, we show how we use term walks to represent search state and actions which are the domains of policy and value functions.

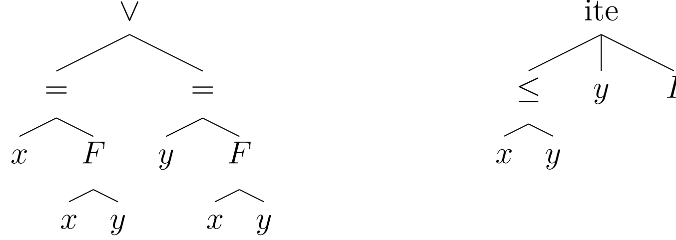
4.6.1 Term Walks

Term walks are the most basic syntactic features and the building blocks of subsequent definitions. Using the standard definition of a *directed walk* from graph theory, we introduce the following:

Definition 4.6.1 (Directed Tree Walk)**.** Given a tree $\langle V, E \rangle$, a directed tree walk of length k is a sequence of nodes $v_0, v_1, v_2 \ldots, v_{k-1}$ such that there is always an edge $e \in E$ connecting v_i and v_{i+1} for $i \in \{0, \ldots, k-2\}$.

By considering the syntax tree of a term, we can analogously talk about term walks as the tree walks of the syntax tree. Furthermore, we can consider the multiset of term walks of a specific size. This is shown in the following example.

Example 4.6.1 (Term Walks)**.** Consider the terms $(x = F(x, y) \vee y = F(x, y))$ and (ite $(x \leq y)$ y I) from Section 4.2. The terms are taken from the specification ϕ and from an intermediate search state, respectively. Note that the latter is an incomplete function. The two terms have the following tree representation.



Using a simple recursive algorithm traversing these trees, we can easily enumerate all term walks of size 2. For the first term, we get the following multiset of term walks of size 2

$$\{[\vee, =], [\vee, =], [=, x], [=, F], [F, x], [F, y], [=, y], [=, F], [F, x], [F, y]\}$$

and for the second term we get

$$\{[\text{ite}, \leq], [\text{ite}, y], [\text{ite}, I], [\leq, x], [\leq, y]\}.$$

In the following section, we will use the multiset of term walks of size 2 to characterise states and actions of the synthesis game.

4.6.2 Representing the Search State and Actions

Recall from Section 4.3.1, a synthesis state $s = \langle \phi, v \rangle$, consists of a static component ϕ, and a dynamic component v which is either a partial or a complete program. Both ϕ and v correspond to first-order terms and formulas. Therefore, we can use term walks as introduced in Section 4.6.1 to characterize the search state using a bag-of-words model as follows:

Definition 4.6.2 (Term/Formula Feature Vector)**.** Let ϕ be a term or formula that we want to represent with a feature vector v. W is the multiset of all term walks of length 2 and L the multiset of all constants and variables occurring in ϕ. We fix a

number M called the *hash base* and let v be of dimension M initialised with 0 in every entry. The feature vector v of ϕ is defined as follows:

$$v[\texttt{hash}(t) \mod M] = v[\texttt{hash}(t) \mod M] + \#_W(t) \quad \text{for all } t \in W,$$
$$v[\texttt{hash}(c) \mod M] = v[\texttt{hash}(c) \mod M] + \#_L(c) \quad \text{for all } c \in L,$$

where $\texttt{hash}(t)$ is the hash value of t and $\#_W(t)$, $\#_L(c)$ are the multiplicities of t and c in the multisets W and L.

Example 4.6.2 shows how a bag-of-words feature vector can be generated. Note that this is a common feature generation technique used in machine learning. Using this in conjunction with term walks, we can now create feature vectors for terms and formulas. To characterise a state $s = \langle \phi, v \rangle$, we concatenate the feature vectors for ϕ and v. This means that the state s is represented by a feature vector of dimension $2 * M$ where M is the hash base used in Definition 4.6.2. The first half of the feature vector is the bag-of-words representation of ϕ and the second half the bag-of-words representation of v. As previously mentioned the static component, ϕ, never changes throughout the search allowing us to cache the feature vector of ϕ.

Finally, we also need to represent actions that can be taken in a state as feature vectors. Unlike other approaches to synthesis, SyGuS introduces the ability for users to specify the grammar rules. Hence, we cannot represent one action by the corresponding grammar rule, since the grammar rules might very well change from one synthesis problem to another. To overcome this we simply characterise an action by its resulting term. This means that to characterise an action that moves from $s = \langle \phi, v \rangle$ to $s' = \langle \phi, v' \rangle$ we take the feature vector of v'. Hence, to represent the domain $S \times \bigcup_{s \in S} A_s$ of the policy function we simply take the concatenation of the representations of ϕ, v, and v'. In combination, we get a feature vector of dimension $3 * M$ to represent elements in $S \times \bigcup_{s \in S} A_s$. Again, we can cache the feature vector of ϕ to avoid redundant computations.

A Note on Performance: Due to the speed of tree traversal and node selection in our algorithm, the calls to the learning models are comparatively slow. Hence, it is vital for performance gains to ensure that the generation and representation of feature vectors for states and actions can be implemented efficiently. These concerns also carry over to the choice of machine learning models. This is also a factor when applying machine learning to theorem proving [197, 199].

Example 4.6.2 (Bag-Of-Words Vector)**.** Consider the feature vector depicted in Figure 4.3. We have a vector of dimension M with indices ranging from 0 to $M-1$. At position 1 and 544, we have entries 7 and 4, respectively. This means there are 7 features with hash congruent to 1, and 4 features with hash congruent to 544 modulo M. Note that all entries with value 0 indicate that no such features are present.

0	1	...	542	543	544	545	...	M-2	M-1
0	7	...	0	0	4	0	...	0	0

Figure 4.3: Bag-Of-Words feature vector of dimension M with 7 words with index 1 and 4 with index 544.

4.7 Training Models and Reinforcement Learning

We have now described where we use machine learned heuristics and how we represent search states and actions. In this section, we will present a reinforcement learning algorithm with which we train these models.

A big issue in applying data driven techniques to synthesis is that training data is hard to come by. One of the main reasons we looked into applying AlphaZero's MCTS to SyGuS is that it lends itself to a reinforcement learning loop that iteratively generates training data for later iterations. In other words, the models can use the data generated from previous iterations to "learn from experience". Algorithm 4 shows the reinforcement learning loop we use in our setup. This loop runs on a set B of SyGuS problems called the training set. In the first iteration, the policy

Algorithm 4: Reinforcement Learning Loop

```
Data: set of SyGuS problems B
ν ← default_value;
π ← default_policy;
policy_data ← {};
value_data ← {};
for i ← 0 to RL_ITERATIONS do
    for P ∈ B do
        r ← search(P, π, ν);
        policy_data ← policy_data ∪ get_policy_data(r);
        value_data ← value_data ∪ get_value_data(r);
    π ← train_policy_model(policy_data) ;
    ν ← train_value_model(value_data) ;
```

and value functions π and ν are set to default functions. In our case, the default policy is simply the constant 1 and the default value of a state s is simply $0.95^{|\mathrm{NT}(s)|}$ where $|NT(s)|$ is the number of non-terminals in the dynamic part of state s. In each iteration we first run the search procedure as described in Section 4.4 for each problem P in B using the policy and value functions π and ν. After either successful or unsuccessful completion of the search, we use the functions `get_policy_data` and `get_value_data` to generate training pairs from the runs. These pairs are added to the policy and value training data. Finally, at the end of each iteration, we use the data from previous iterations to train new policy and value functions which are used in the next iteration.

The ground truth training pairs from a search run r are generated as follows:

`get_policy_data`: For policy data, we only care about successful runs (i.e. where the search procedure was able to find a solution). We take the path p in the grammar tree of the successful run, leading from the root node to the leaf node that represents the solution. For state s and action a that was taken to reach the next state s' in p, we create the data pair $(\langle s, a \rangle, n)$ where n is the number of visits of s' in relation to the sum of all visits of all child nodes of s including s'.

`get_value_data`: As value training data, we consider both successful and unsuccessful runs. For the successful runs, we again take the path p in the grammar tree of the successful run leading from the root node to the leaf node that represents the solution. For each state s on the path, we take the pair $(s, 0.9^D)$ where D is the distance (i.e. length of the remaining path) from s to the final and winning state in the path.

For the unsuccessful runs, we take the sequence p of *Big-Steps* that was taken starting from the root of the grammar tree. And for each state s in p we take the pair $(s, 0)$ as training data.

The presented reinforcement learning setup presents one way of solving the aforementioned data scarcity problem. By iteratively collecting data in each iteration, we are able to collect more and more data from previous successful runs. This setup also has the advantage that we do not require a priori ground truths. In the experimental evaluation presented in Section 4.9, we show how the policy and value functions improve their precision throughout the iterations.

4.8 Generating New SyGuS Problems

Previously, we introduced a reinforcement learning loop as a means to overcome the lack of ground truth. In each iteration, the RL algorithm runs the search procedure on a training set. Consequently, the larger the training set, the better the algorithm is likely to perform. Randomly generating SyGuS problems as data is not practical: if we randomly generate the specification ϕ, it is highly unlikely that a solution F exists; and if we randomly generate the solution F, we need to find a way to infer a meaningful specification ϕ that admits f and does not give away the answer. For this reason and due to the relative immaturity of the field, the number of benchmarks available in the linear arithmetic category of the SyGuS competition is relatively small (less than 1000). In contrast, for each of the background theories, there are a lot more first-order SMT problems than there are SyGuS problems. Since SMT and SyGuS share the same background theories, it is logical to look at the former

to generate problem sets for the latter. In this section, we will describe a novel technique based on syntactic unification and anti-unification that does exactly that.

4.8.1 SyGuS Generation Algorithm

We now present the algorithm that turns an SMT problem $Q = \langle \tau, \phi \rangle$ into a SyGuS problem $P = \langle \tau, F, \psi, G \rangle$. Without loss of generality, we assume that Q is τ-valid since if Q is $UNSAT$, we can turn it into a valid problem by negating it. Furthermore, if Q is SAT but not valid, we can use the resulting model to obtain a valid problem by substitution. Note that, if Q is a hard problem, we might not know if a problem is $SAT/UNSAT$. In our experiments, we found this to be rarely the case and we just discarded these problems. Due to the abundance of SMT problems, this is not an issue. With these assumptions, the following gives a step-by-step algorithm of how we generate a SyGuS problem P from a given valid SMT problem Q.

1. Heuristically select a set S of non-overlapping sub-terms of Q
2. Compute LGG l of S with fresh variables $x_0, \ldots, x_{n-1}$ not occurring in ϕ.
3. For each $t \in S$, let $\sigma_t = \{x_0 \mapsto u_0, \ldots, x_{n-1} \mapsto u_{n-1}\}$ be the solution[1] to the unification problem $l \stackrel{?}{=} t$.
4. Replace each term $t \in S$ in ϕ with $F(x_0, \ldots, x_{n-1}) \cdot \sigma_t$ to obtain ψ.
5. Let G be a grammar producing all terms in τ using arguments $x_0, \ldots x_{n-1}$.
6. Return SyGuS problem $P = \langle \tau, F, \psi, G \rangle$.

The execution of this procedure is shown in Example 4.8.1 and an application to a problem from the SMT-LIB benchmark database can be found in Appendix A. As shown in Theorem 4.8.1, the SyGuS problem resulting from this algorithm is guaranteed to be solvable. Furthermore, we get a concrete solution to this problem.

[1] Note that such a solution must exist by the Definition 4.1.2.

Theorem 4.8.1. Let $Q = \langle \tau, \phi \rangle$ be a valid SMT problem and $P = \langle \tau, F, \psi, G \rangle$ be the corresponding SyGuS problem obtained from the aforementioned algorithm. The resulting problem P is solvable and the LGG generated in Step 2 is a solution.

Proof. To show that P is solvable we provide a term l such that $\tau \models \psi[F \mapsto l]$ and $l \in L^G$. We show that the LGG produced in Step 2 is such a term. Since anti-unification only introduces new variables $x_0, \ldots, x_{n-1}$ and no constants or functions, we have that $l \in L^G$. We now show that $\tau \models \psi[F \mapsto l]$. Let t be an arbitrary term in S. Observe that by Definition 4.1.1 and Step 3 of the algorithm we have $l \cdot \sigma_t = t \cdot \sigma_t = t$. The second equality is obtained by combining the fact that the domain of σ_t is $\{x_0, \ldots x_{n-1}\}$ and that t does not contain any of these variables (by Step 2). Furthermore, we get that $(F(x_0, \ldots, x_{n-1}) \cdot \sigma_t)[F \mapsto l] = l \cdot \sigma_t = t$ which is the term that $(F(x_0, \ldots, x_{n-1}) \cdot \sigma_t)$ was replaced with in Step 4. Since this holds for all $t \in S$ we have $\psi[F \mapsto l] = \phi$. Combining this with the fact that Q is valid (i.e. $\tau \models \phi$) we have $\tau \models \psi[F \mapsto l]$. □

Note that the solution to the resulting SyGuS problem need not be unique. The algorithm does not allow for a lot of parameterisation except for the first step. For example, if the set of terms S is selected "badly" the resulting problem might be solvable with the identify function, which is not particularly challenging or interesting. Indeed, the heuristics with which terms are selected have a large impact on the quality of the resulting problem. If the wrong subterms are chosen, the resulting LGG might be trivial (e.g. For the terms $\{5+4, 9*x\}$ the LGG is basically the identity function). In our case, we select sub-terms with the following criteria:

- size of the resulting LGG
- terms of type `int`

The first criterion rules out many cases where the identity function is a solution to the resulting problem. Furthermore, we found that allowing sub-terms of type `bool` often leads to trivial solutions such as `True` or `False`. Hence, we also select by the second criterion. The usage of more advanced heuristics, maybe even machine learned heuristics remains to be investigated.

Example 4.8.1 (SyGuS problem Generation)**.** Consider the following SMT problem stated in SMTLIB syntax:

```
(set-logic LIA)

(assert 10 * x = (2 * x) + y)
(assert x * 3 + 5 = 8)

(check-sat)
```

We can use a SMT solver to obtain the satisfying assignment $x = 1, y = 8$. By applying this assignment to the problem we get the following constraints:

$$10 * 1 = (2 * 1) + 8$$
$$(1 * 3) + 5 = 8$$

We can now choose any set of non-overlapping sub-terms. For example, let us choose $(2 * 1) + 8$ in the first and $(1 * 3) + 5$ in the second assertion. Applying Plotkin's anti-unification algorithm to these terms we get the least general generaliser (LGG)

$$u_1 * u_2 + u_3,$$

with fresh variables u_1, u_2, and u_3. By applying unification to $(2 * 1) + 8$ and $u_1 * u_2 + u_3$ we get the unifier $\sigma = [u_1 \mapsto 2, u_2 \mapsto 1, u_3 \mapsto 8]$. Similarly, we get the unifier $\sigma' = [u_1 \mapsto 1, u_2 \mapsto 3, u_3 \mapsto 5]$ by applying unification to $(2 * 1) + 8$ and $u_1 * u_2 + u_3$. Now, we can replace the sub-terms selected in the first step with a fresh second-order variable F with arguments u_1, u_2, and u_3 in the original assertions. The unifiers σ and σ' tell us what arguments have to be applied. In particular, we get $F(u_1, u_2, u_3) \cdot \sigma = F(2, 1, 8)$ for the first and $F(u_1, u_2, u_3) \cdot \sigma' = F(1, 3, 5)$ for the second term. The resulting assertions are

$$10 * 1 = F(2, 1, 8),$$
$$F(1, 3, 5) = 8.$$

Now all we have to do is select a grammar G such that $u_1 * u_2 + u_3 \in L^G$. The simplest such grammar is the free grammar that generates all terms in LIA. Let ϕ be the conjunction of the assertions: $10 * 1 = F(2, 1, 8) \land F(1, 3, 5) = 8$. The

tuple $\langle \texttt{LIA}, F, \phi, G \rangle$ now describes a SyGuS problem with the solution $u_1 * u_2 + u_3$. Written explicitly in the SYGUS input format we have generated the following problem:

```
(set-logic LIA)
(synth-fun F ((u Int) (v Int) (w Int)) Int)

(constraint (= (10 * 1) (F 2 1 8)))
(constraint (= (F 1 3 8) 8))

(check-synth)
```

4.8.2 New Problem Set

We apply the generation algorithm to SMT problems taken from the Linear Integer Arithmetic (LIA) and Quantifier-free Linear Integer Arithmetic (QF_LIA) tracks of the SMT competition [200]. From this, we generate 8186 new SyGuS problems. We run `cvc5` [201], the state-of-the-art SyGuS solver, on these problems and group the problems into the following categories, in approximate order of complexity:

Basic (B) problems that `cvc5` could solve and where a valid solution is a function that returns a single constant or variable.

Straight-line (S) problems that `cvc5` could solve and are not basic but where a valid solution does not need control flow (i.e., there is a valid solution that performs simple mathematical operations and contains no if-then-else statements).

Control-flow (C) problems that `cvc5` could solve and where the solution has control flow (i.e., the solution contains if-then-else statements).

Unsolved (U) problems that `cvc5` could not solve with a 120s time-out.

Table 4.2 reports the number of problems in each of these categories in our new data-set and in the original data-set from the SyGuS competition. The SyGuS competition data comprises all benchmarks in LIA from the General and Invariant Synthesis tracks. Where a benchmark lacks a grammar restriction, we augment the

Data set	#B	#S	#C	#U	Total
SyGuS-competition	290	93	38	536	957
Generated generated data	3760	2495	1693	220	8168
Filtered generated data	–	211	513	220	944

Table 4.2: Number of problems by category: basic problems (B), straight-line problems(S), problems with control-flow (C), and unsolved problems (U).

benchmark with the same default grammar as our generated data. The SyGuS data contains more unsolved benchmarks than ours, but we note that these often include multiple benchmarks that are slight variations on a theme (for instance, there are 10 unsolved benchmarks that require synthesizing a function that calculates the maximum of the input variables.

In total, we generate 8186 new SyGuS LIA problems. We remove any basic problems and filter the remaining problems to reduce the number of very similar problems. We use a heuristic based on the file names and duplicate comments as these are usually indicative of the source of the SMT benchmark and matching in both correlates with very similar problems but perhaps with different constant values. This gives a total of 944 new SyGuS LIA problems. In combination with the existing benchmarks from the SyGuS LIA competition [23] we have a problem set consisting of 1901 SyGuS LIA problems. This almost doubles the total available number of SyGuS LIA problems, and increases the number of solvable benchmarks with control flow by $> 10\times$. As the number of publicly available first-order problems increases year-on-year, our approach can be applied to automatically generate more SyGuS problems even if the number of SyGuS problems does not grow as much.

4.9 Experimental Evaluation

We implemented a SyGuS solver based on the techniques presented in this chapter. We evaluate these techniques in an experimental evaluation using standard SyGuS benchmarks ("old" problems) as well as newly generated benchmarks as discussed in Section 4.8 ("new" problems) to answer the following research questions:

RQ1 Can MCTS with RL be used in function synthesis?

RQ2 Does our data generation improve the performance?

RQ3 How does MCTS compare to other techniques?

The implementation is written in C++ using z3 [202] as an SMT solver to check the correctness of potential candidate solutions. As policy and value estimators we use gradient boosted trees as provided by the XGBoost [180] library. These are continuously trained in a reinforcement learning loop as presented in Section 4.7. Running the experiments multiple times with different hyperparameters ranging from (10 to 30) we found that the best tree depths are 20 and 25 for value and policy. We did not experience any significant improvements by changing other hyperparameters. In the MCTS we do 30 big-steps and 6500 rollouts with a decay factor of 0.98^B where B is the number of big-steps. These were optimized with regard to a 100s timeout. Generally, higher numbers are better but also require more time. As a hash base for the feature vectors, we use $2^{12} - 3 = 4093$. For training, we only take data obtained from the previous 4 iterations instead of all previous iterations (we tried values between 2 and 20). Previously, we had a Python-based implementation where we also experimented with other machine learning models such as K-Nearest Neighbours and Linear Regression as well as larger hash bases. However, due to the high prediction latency as well as overall slowdown we were unable to improve on the baseline. In either case, tree models outperformed other machine learning models.

Benchmarks & Setup: We use the SyGuS LIA data set consisting of 1901 problems described in Section 4.8. For each of the experiments, we create a 75 : 25 training/testing split on the data sets. The data set, as well as the code with Dockerfile and instructions on how to install dependencies, compile the code, and run experiments can be found in the supplementary material. The experiments were run on Amazon Elastic Compute Cloud (EC2) on a `r6a.8xlarge` instance with an AMD EPYC 7R13 CPU and 256GB of Memory running Amazon Linux 2 with kernel `6.1.34-58.102.amzn2023.x86_64`. We ran each search procedure with a timeout of 100 seconds. Our implementation introduces non-determinism

in two sections: training of the machine learning models and a random tie-break in child selection when two or more child nodes have the same scores. We ran the experiments 5 times with fixed seeds.

4.9.1 Results

We now present the experimental results to answer the aforementioned research questions. The main results are shown in the Tables 4.3 and 4.4.

Can MCTS with RL be used in function synthesis? We ran experiments on the union of all benchmark sets resulting in 1425 training and 476 testing problems. The results are presented in Table 4.3. The baseline is the bare algorithm with a default value and policy, where no policy or value estimators have been trained yet. In this version, we solve 471.6 (33.1%) and 163.4 (34.3%) problems on average in the training and testing sets, respectively. Note that we only report on the mean as with the default policy and value functions the algorithm is almost deterministic (the only randomness comes from a tie-break mechanism). From the second iteration onwards policy and value functions are trained on the data obtained from the training set in the previous iterations (no data is collected from the testing set). If we only consider the best iteration of each experimental run we solve, on average, 859.8 on the training and 289 on the testing set. This is an *improvement* of 27.2 and 26.4 percentage points compared to the first iteration for training and testing sets (i.e. the baseline). The best iterations of each experimental run differ from each other with a standard deviation of 3.97 problems on the training set and 1.87 problems on the test set. When considering the part of the testing set exclusively consisting of the old problems, we see an improvement of almost 11 percentage points over the baseline. In conclusion, our method solves a significant amount of testing and training problems and the reinforcement learning setup improves on the results significantly. This leads us to answer RQ1 in the *affirmative*.

action	data	total	baseline		Solved in Best Iteration					cvc5	
		#	#	%	min	max	# mean	% mean	stdev	#	%
train	old+new	1425	471.6	33.1%	852	865	859.8	60.3%	3.97	834	58.5%
test on	old+new	476	163.4	34.3%	287	292	289	60.7%	1.87	295	62.0%
test on	old	249	49.4	19.8%	75	78	76.4	30.7%	1.14	115	46.2%

Table 4.3: Table showing the experimental results comparing baselines to the best learned iterations. The first and second rows show experimental results on training and testing data respectively. The last row shows the results of the testing data when only considering the subset of problems that originate from the old set. In either case, a problem can only occur in exactly one of test set and train set.

Does our data generation improve performance? Doing a more fine-grained analysis of the experiments, we find that the testing set in the previous experiments contains 249 problems from the old data set. Of these, between 75 (30.1%) and 78 (31.3%) are solved with a mean of 76.4 (30.7%) and a standard deviation of 1.14. This is shown in the third row of Table 4.3. To compare, we conduct two additional experiments with the following data set-ups [2]:

1. Train with *old* and test with *old* problems
2. Train with *new* and test with *old* problems.

Once again, we ran each of the two experiment set-ups five times. The results are shown in Table 4.4, where the first two rows show the first experiment and rows three and four show the second experiment. For the first experiment, we find that training and testing exclusively on the old data set improves the performance by approximately 15 percentage points on the training and the testing set. In particular, we go from solving 16.6% of problems in the baseline to solving 30.7 on the test set. In the second experiment, we find that training with the new data set and testing with the old data set does not lead to significant improvements on the testing set. In particular, we only improve by 2.4 percentage points. That is, we go from solving 151.8 problems in the first iteration to solving 175.4 problems on average in the best iteration. Notably, the impact on the training data is a lot more significant. When training with the entire set of new problems we go from

[2]Recall, *Old* data describes the standard SyGuS benchmark problem sets and *new* data is the data we generated from SMT problems as described in Section 4.8.

action	data	total	baseline		Solved in Best Iteration					cvc5	
		#	#	%	min	max	# mean	% mean	stdev	#	%
train on	old	717	112.2	15.7%	205	224	210.8	29.4%	7.6	315	41.0%
test on	old*	240	39.8	16.6%	69	79	73.6	30.7%	3.58	108	45.0%
train on	new	944	479.6	50.8%	874	886	879.6	93.2%	3.37	727	77.0%
test on	old	957	151.8	15.9%	169	184	175.4	18.3%	5.97	402	42.0%
test on	old*	240	38.6	16.1%	45	50	47.4	19.8%	2.51	108	45.0%

Table 4.4: Experimental results with different training and testing sets. Testing and training problems do not overlap. The two old sets indicated with an asterisk are the exact same set.

solving 50.8% of the problems in the first iteration to solving 93.2% of all training set problems. In conclusion, we find that the new training data does not improve our performance on the old test set significantly. This holds in particular when compared to training on the old data as shown in the first two rows of Table 4.4. However, when considering the overall experiment with the combined data sets as in Table 4.3 we can say that by adding the new data set we can solve many new problems while also not diminishing the performance on the old problems. Hence, overall, we can solve more problems. This leads us to answer RQ2 in the *negative* when only considering the old data set. However, when looking at the overall data set that includes both, old and new data, we answer RQ2 in the *affirmative*.

How does MCTS compare to other techniques? We compare our approach to `cvc5`, the SyGuS solver which performed best in the most recent SyGuS competition [23]. In Tables 4.3 and 4.4, we also present the results of running `cvc5` [201] on each of the data sets with a 100s timeout. In the experiment with combined data sets (i.e. Table 4.3), `cvc5` solves 834 problems in the training set and 295 in the test set. In comparison, our tool solves, on average, 25.8 problems more on the training set (in the best iteration) and 6 fewer on the test set. Further, we find that we solve 41 test set and 141 training set problems on which `cvc5` fails, these are called *unique problems*. When looking at the problem sets separately as in the experiments presented in Table 4.4 we see that `cvc5` outperforms our tool on the old set while lagging behind on the new set. In particular, when

considering the old training data set `cvc5` solves over 41% of the old problems while our tool solves 29.4% in the best iteration. On the testing set this difference is more significant. The story is different on the newly generated problems. In this setup our method solves 93.2% in the best iteration where `cvc5` only solves 77%. These results lead us to claim that MCTS based function synthesis with RL compares favourably to `cvc5` on the newly generated data set while lagging behind `cvc5` on the old pre-existing data set.

4.9.2 Discussion

We presented an experimental evaluation to answer research questions regarding the overall efficacy of MCTS based synthesis with RL (RQ1), the usefulness of our SyGuS problem generation technique (RQ2), and the competitiveness of our technique (RQ3). In this section, we discuss the results of the evaluation and limitations of our approach.

Efficacy: We answered the first research question in the affirmative, meaning that Monte-Carlo tree search based function synthesis is effective in practice. On data sets comprising newly generated as well as pre-existing benchmarks, our technique improves by around 27 percentage points on training *and* testing sets when comparing the first iteration to the best iteration. Running each of the experiments 5 times to account for nondeterminism we find that the results are fairly stable with a standard deviation of less than 4 solved problems per experiment. We also plot the number of solved problems for each reinforcement learning iteration of each experiment in Figure 4.4. Each line represents one experimental run showing the number of solved testing problems on the y-axis for each iteration on the x-axis. In each experiment, we observe a gradual improvement at a comparable rate up until iteration 7. After that, the performance seems to flatten out. This indicates that the reinforcement learning loop indeed improves with each iteration at the beginning. The graph also shows regressions in some iterations of some

experiments. However, they never go as low as the first iteration and seem to be fixed in succeeding iterations.

We have previously indicated the relative speed with which simple enumeration procedures operate. This has had a significant impact on our choice of learning model as we had to compete with the speed of the baseline enumeration. This contrast is again shown in Figure 4.5. This graph plots the number of solved testing problems within a certain time budget for each iteration of one experiment. Hence, each line represents one iteration of the RL loop. The dashed line represents the first iteration with the default policy/value functions that do not use any machine learning. This iteration, which is the baseline enumerator, is extremely fast and solves the vast majority of its problems within a very short period of time. In fact, for the first few seconds of each search the baseline enumerator seems to outperform all iterations with machine learned search guidance. These catch up after around 20 seconds, where they start to lift off. Between 20 and 60 seconds the machine learned guidance solves many additional problems while the baseline flattens out after 20 seconds and does not solve any additional problems anymore.

SyGuS Problem Generation: To answer RQ2 we conducted two additional experiments with which we tried to evaluate the usefulness of the newly generated data set. The results presented in Table 4.4 indicate that exclusively training on the new data set does not lead to significant improvements on the old data set. This indicates that knowledge learned from the problems in the new data set does not carry over as well to the old data set. One of the reasons for this may be that the old data set consists of mostly handwritten problems while the new set has been automatically generated to some extent. Another explanation might be that the new data set is simply significantly easier. This is supported by the fact that when training on the new data set we solve up to 93.2% of the training set. However, `cvc5` only solves 77% of these problems, indicating that they might still be "interesting" problems worth learning from. In contrast, training and testing exclusively on the old data set lead to results that are comparable to

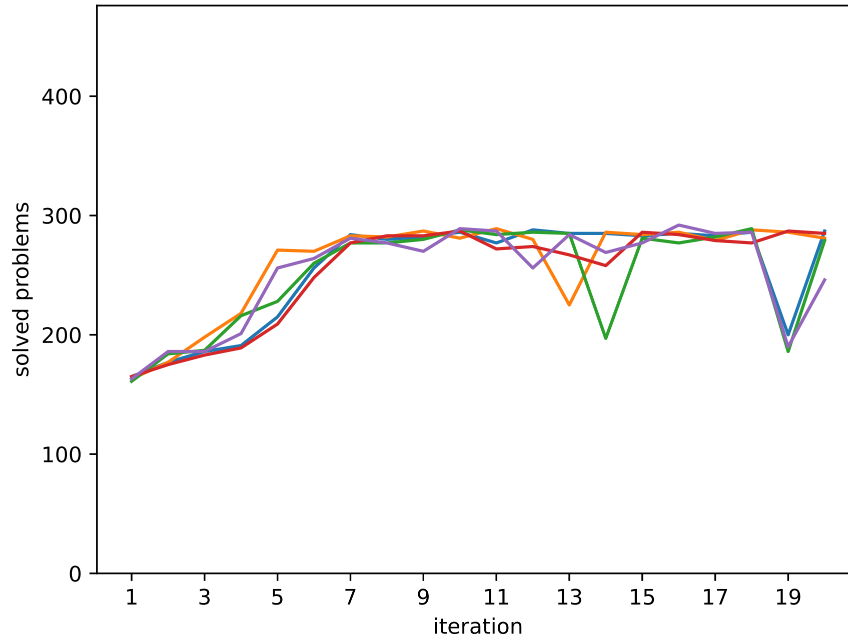


Figure 4.4: Number of solved problems in the testing set for each iteration in each of the 5 experiment runs. Each line represents one experimental run.

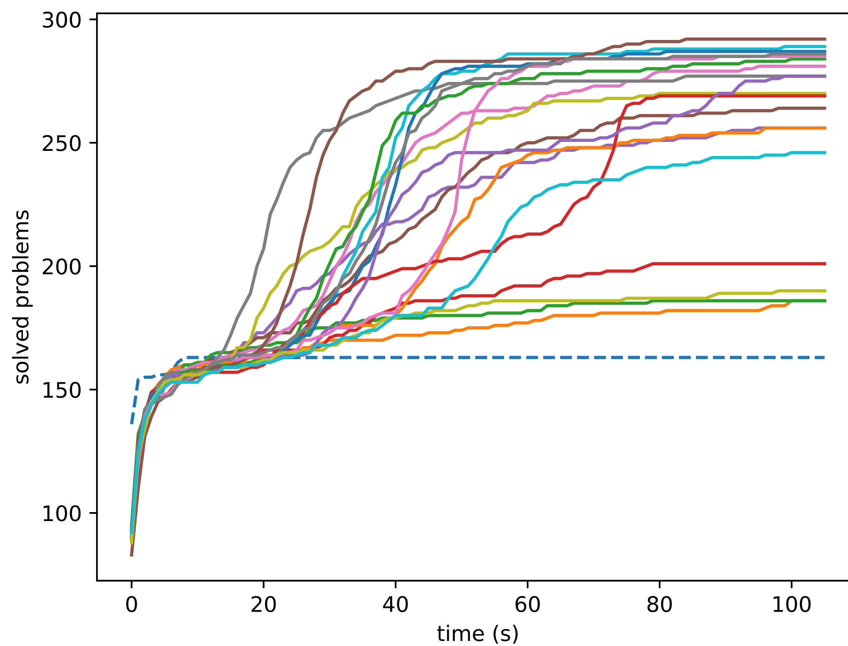


Figure 4.5: Number of solved problems in the testing set within a certain time budget in each iteration. The dashed blue line is the first iteration with a default value/policy.

the overall experiment with training/testing on the combined data sets presented in Table 4.3. In the overall experiment, we can solve 30.7% of the old problems in the test set while training exclusively on the old problems we also solve 30.7% of the old problems in the test set. This leads us to say that, while the newly generated problems only marginally help solve problems from the old set (i.e. improvement of around 2.4%), they do help in solving a wider range of problems. In particular, by combining new and old problems in the training set, we can solve more problems of the new problems while still solving 30.7% of the old problems. Hence, combining the two data sets leads us to solve a wider range of problems.

Comparison: To compare our tool to the state-of-the-art in SyGuS, we also ran `cvc5` on all of the problem sets. Unfortunately, we are unable to compare to Eupony [88] as the available code does not compile. In the combined data sets we showed that we can solve more problems than `cvc5` on the training data while performing comparably on the testing data. We also find that we solve 41 and 141 unique problems on the testing and training sets, respectively. By considering the new and old data sets separately, we see that `cvc5` performs significantly better on the old data set while our tool shows a superior performance on the new data set. We hypothesize this is because our new benchmark set is not typical of the existing SyGuS benchmarks (this is also demonstrated by `cvc5`'s failure to solve many of these examples). Furthermore, the old data set has been freely available and `cvc5` has been tested on and developed alongside this problem set. At this point, it should be pointed out that `cvc5` is a mature software artefact consisting of over 400k lines of code incorporating many decision procedures and domain-specific heuristics to improve function search. In comparison, our implementation exclusively uses the learned search heuristics presented in Sections 4.4 and 4.5. We emphasise that in all experimental setups, our agent exclusively learns from its own previous iterations without external guidance or solutions. Notably, we do not train with any information from the test set.

Limitations: The main limitation is that our approach relies on the existence of high quality training data. This means that, if this data is not available for whatever reason, this set-up does not work. For example, if the set of training problems only contains problems that cannot be solved in the first iteration, we are not able to train the policy and value estimators for the succeeding iterations. Hence, we run into a type of bootstrapping issue where we are unable to "get off the ground". Similarly, if the training data is significantly different from the testing data the policy and value estimators might not perform very well. Such a behaviour can be seen in the second experiment presented in Table 4.4 where the training on the new data set does not carry over to a significantly improved performance on the old data set. Again, we emphasise that having a diverse training set leads to an overall better performance.

As discussed, the Monte-Carlo tree search based solver that we presented is still an enumerative solver. Hence, the main drawbacks of this search technique still apply. In particular, if we have grammar trees with a large branching factor, finding expressions of large sizes is still difficult as they are deeper in the grammar tree. While we have shown that machine learned guidance can help prune the search space and prioritise "good" branches, this fundamental limitation of enumerative search still applies.

4.10 Threats to Validity

In the previous section, we presented an experimental evaluation showing that Monte-Carlo tree search can be applied to SyGuS. As always, the validity of such an empirical evaluation can be questioned. In this section, we discuss the main threats to the validity of our experiments.

Benchmark Bias: One part of the experiments was conducted using the standard SyGuS benchmark set. The field of Syntax-Guided Synthesis is relatively immature, this has led to small problem sets compared to more established fields like SMT, for instance. This also means that the sources of benchmark problems are not as diverse since it has not yet been applied to as many domains as it could be. This

could lead to our methods breaking down once applied to more diverse problems. We mitigate this issue in two steps. First, we not only consider standard `LIA` function synthesis problems, but we also translate `LIA` invariant synthesis problems to function synthesis problems. Finally, in Section 4.8 we also present an algorithm for generating SyGuS problems from SMT problems, allowing us to make use of the more mature field of SMT solving. Both of these measures help to mitigate a certain benchmark bias by ensuring a certain diversity in the problem set. However, it remains to be seen if the efficacy shown in the experimental evaluation carries over to later iterations of the benchmark database.

New Benchmark Set: We conduct the experimental evaluation on a mix of newly generated and standard benchmark problems. Generating new problem sets always introduces a bias. In our case, one point where we introduce a bias is in step 1 of the algorithm presented in Section 4.8. By considering the entire SMT `LIA` data set, we mitigate some of the biases introduced. Furthermore, by choosing terms with “large” LGGs, we use a *neutral* measure. Finally, we also introduce a bias by discarding problems with *trivial* solutions. This classification is done using the SyGuS solver `cvc5`. While `cvc5` is a very mature state-of-the-art solver, it is still not immune to biases which would carry over to our problem set. To some extent, the bias introduced by adding this data set is mitigated by considering the union of both, old and new, data sets.

Variance in Experiments: Like most machine learning algorithms, we also introduce randomness in multiple locations. In particular, in the initialisation of the machine learning model during training, and as a tie-break when two actions have the same score. Like any empirical analysis that uses nondeterminism, we could have just gotten lucky in our evaluation. To reduce this threat, we first determinised the experiments by fixing an arbitrary seed. This also helps reproduce the experiments. In addition, we ran the experiments multiple times and reported on the average and standard deviation of the experiments.

Hyperparameters Finally, the algorithm we evaluated has multiple hyperparameters. These are part of the MCTS as well as the machine learning models we used. We used a subset of the data to tune the hyperparameters. The selection of this subset may introduce some bias. Furthermore, problems with a different character that could, for instance, originate from a different set of applications may perform worse on the hyperparameters we used. This is also mitigated to some extent by considering standard benchmark sets as well as newly generated problems. However, the robustness in other domains remains to be seen.

4.11 Related Work

In Chapter 3, we introduced the established characterisation of function synthesis algorithms using the following three dimensions [45]: *User Input*, *Search Space*, and *Search Technique*. The technique we will present in this chapter can be characterised as follows:

Intent: Abstract logical specifications given by the user in first-order logic with axioms, functions, and constants defined in the first-order theory of linear integer arithmetic.

Search Space: Any syntactic restriction that is allowed under the standard specified by `SyGuS-IF` using a context-free grammar.

Search Technique: Monte-Carlo based tree search of the possible expressions defined by the grammar. As search heuristics, we use machine learned policy and value functions in combination with upper-confidence bounds to balance exploration and exploitation.

As discussed in Chapter 3, machine learning has been applied to function synthesis before. In the majority of cases, this was restricted to the domain of Programming-By-Example (PBE). In contrast, when considering abstract specifications like the work presented in this chapter, training data is hard to come by. In the following, we discuss work closely related to the contributions of this chapter.

In [91], the author uses Tree Neural Networks (TNNs) to synthesise combinators and polynomials. Notably, the paper has served as an inspiration for the research presented in this chapter as it also applies AlphaZero style Monte-Carlo Tree search. The TNNs are estimators for policy and value, which, in our case, are gradient boosted trees. In addition, [91] differs from our work in more fundamental aspects. First, the grammars describing the search spaces are fixed and consist of 5 and 2 rules for combinators and polynomials, respectively. These fixed rules are baked into the search procedure. This restricted setting is not applicable to SyGuS where grammars can and do vary from problem to problem and usually consist of significantly more than 5 rules. In particular, the default grammar for SyGuS `LIA` consists of 17 rules which makes for a considerably larger search space. The second difference lies in the specifications of the synthesis targets. In particular, for the combinator synthesis, the specification for a target F is given as an equation $F x_0, \ldots, x_{n-1} = h[x_0, x_{n-1}]$. Similarly, for polynomials, the specifications are given as diophantine sets modulo 16. Both specification designs are very restrictive and not applicable to SyGuS. Furthermore, they allow for a straightforward automatic generation of arbitrary amounts of training data. This permits the use of TNNs that require significant amounts of training data.

Similarly, in [127] the authors also use an AlphaZero style agent to synthesise invariants. The invariants are constructed using a nondeterministic program that features a *choose* operator to select an invariant from a set of invariants that are generated using abductive reasoning. The choice operator is then refined using the machine learned heuristics provided by graph transformer networks. A notable difference to the work presented in this chapter is that the networks are trained in a reinforcement learning loop with a teacher/solver setup. The teacher is trained to generate invariant synthesis problems and the solver is trained to solve these problems. Both models are trained in an alternating fashion.

Reinforcement learning in combination with graph neural networks (GNNs) is used in [89]. In particular, a graph embedding is learned from graph representation of the semantic and syntactic specification. Similar to us, they represent grammar

expansion rules as actions of a game and train machine learning models (GNNs in their case) to act as policy estimators. In contrast, however, they use a simple recursive depths-first-search where the expansions are selected based on the policy predictions of the GNN. No rollouts or balancing of exploration and exploitation is performed as far as we know. Furthermore, they apply their experiments to the domain of circuit synthesis which is a subset of SyGuS where the goal is to synthesise boolean functions. In this setting, the specifications are given as an equivalent program. Hence, the domain of circuit synthesis falls into the subset of "Synthesising Equivalent Programs" discussed in Chapter 3 [203].

A Bayesian approach is followed by Euphony [88] where weights for the grammar rules are trained. A higher weight indicates that applying this rule is more likely to be successful. These trained weights are subsequently used in an A-star style search algorithm to search for expressions that satisfy the synthesis specification. This stands in contrast to our MCTS based search algorithm. Furthermore, Euphony's grammar weights are pre-trained and require a corpus of training data problems with solutions. This stands in contrast to our set-up where a reinforcement learning loop is used to iteratively create training data for subsequent iterations. This has the advantage that we do not require a priori ground truths as training data.

Finally, in expression search, Monte-Carlo simulations have been used as a search technique [90]. Notably, this work significantly differs from our contributions as no learned heuristics are used for guidance. Furthermore, this follows a more traditional Monte-Carlo tree search, while we follow the approach of AlphaZero [22].

5 Application of SyGuS: Termination Analysis

As discussed in Chapter 1, function synthesis has many applications, especially in program analysis. In this chapter, we will take a closer look at one application in particular – *Termination Analysis*. Termination analysis addresses the question of whether or not a program halts. While undecidable in general, tools and techniques that work in practice have been developed in industry and academia, as discussed in Section 3.2. When proving termination using *ranking functions*, termination analysis becomes a second-order existential problem [8]. In other words, proving termination can be stated as a *synthesis problem* where the goal is to synthesise a ranking function. Ranking functions for programs map program states to a well-founded order such that there is a strict decrease with respect to that order between two consecutive states. We call two states that are related by the transition relation induced by the program *consecutive*. If we fix the well-founded order to be the set of non-negative integers, a ranking function $f : S \to \mathbb{Z}$ is a function satisfying

$$f(x) > f(y) \quad \text{and} \quad f(x) \geq 0$$

for all consecutive program states x and y. The first constraint encodes the *strict decrease*, while the second ensures *boundedness from below*. Finding such a ranking

function f for a given program P is a problem of the form

$$\exists f.\ \forall \vec{x}.\ \phi,$$

where ϕ is a quantifier-free formula with constraints and assumptions encoding

- *states* of P,
- the *transition relations* of P,
- invariants of P, and
- ranking function properties strict decrease and boundedness (as above).

As it turns out these second-order existential formulas are subsumed by the problem definition of SyGuS as introduced in Section 2.2.1. It therefore seems natural to extract ranking function synthesis problems as SyGuS problems and use `SyGuS-IF` as an interface to different solvers. This is one of the contributions of this chapter.

We introduce an automatic construction of a formula ϕ from a Java program P that encodes the aforementioned properties. We employ a large amount of well-known standard procedures from program analysis to achieve this. Using ϕ, we construct a SyGuS problem such that any solution is a correct ranking function for P – proving its termination. We implement this automatic translation in a tool called `RFSynth` and use `cvc5` as well as the reinforcement learning based solver presented in Chapter 4 to solve the resulting SyGuS problems. To evaluate this approach, we run an experimental evaluation on a set of termination benchmarks that comprises different problems from literature as well as problems we found to be challenging for state-of-the-art tools.

In summary, the novel contributions of this chapter are:

- We present a technique for generating SyGuS problems with semantic specifications that model ranking functions from given programs.
- We provide an implementation called `RFSynth` that generates `SyGuS-IF` problems from Java programs.

- We identify a set of termination problems that state-of-the-art termination provers have difficulties solving. These usually involve disjunctions and non-linear loop guards.

- We then test and compare `RFSynth` with `cvc5` as a backend to state-of-the-art termination tools by applying them to standard termination problem sets. In addition, we also run our experiments on the aforementioned set of termination problems containing disjunctions and non-linear loop guards.

5.1 Background on Program Analysis

In Chapter 2, we already discussed how to generate first-order logic formulas T_H and A_H describing loop transition relation and auxiliary invariants, respectively. With these, we can now reason about programs. It is important to note, however, that, due to the inherent difficulty of program analysis, formulas describing T_H and A_H are usually over-approximations. The relationships between the different sets of states are shown in Figure 5.1. S is the set of all possible variable assignments to the program variables (i.e. states). The set L is *exactly* the set of all states that are possible in the given program. Since the exact characterisation of L is usually undecidable, we approximate this, using the intersection of T_H and A_H. The striped area represents the states legal by the conjunction (i.e. intersection) of T_H and A_H but that are not possible in the program we analyse. Hence, this area represents the difference between the over-approximation and the actual set. This approach is sound, as any property we prove about $T_H \wedge A_H$ also holds for L. In software verification, common properties of interest include but are not limited to, safety, reachability, as well as a lack of common bugs such as under and overflow, memory leakage, etc.

The verification task we are interested in is termination as introduced in Section 2.3.2. In Chapter 3, we discussed multiple tools that implement different techniques for proving termination. We will focus on termination analysis via *ranking functions*. A ranking function as in Definition 2.3.4 is a function mapping

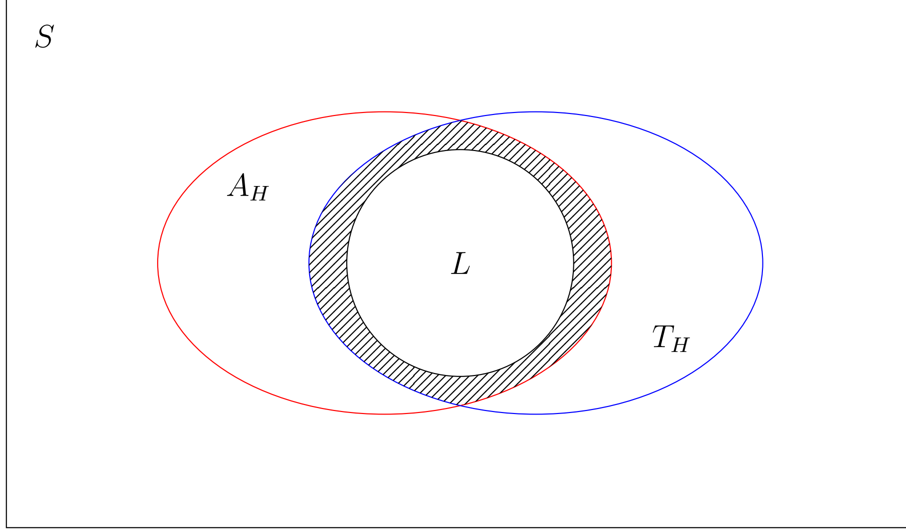


Figure 5.1: Venn diagram of all states S, all possible states according to the program L, all states defined by the transition constraints T_H, and all states defined by the auxiliary invariants A_H.

states to a well-founded order. To facilitate verification procedures, we take the set of integers as the co-domain of the ranking function. Since the set of whole numbers is not well-ordered, we will further impose a non-negativity restriction. In combination, we now have that a function $f : S \to \mathbb{Z}$ is a ranking function for a program P with states S, loop head transition relation T_H, and auxiliary invariants A_H if

$$\forall s, s'.\ T_H(s, s') \wedge A_H(s) \implies f(s) > f(s') \wedge f(s) \geq 0. \tag{5.1}$$

Usually, s and s' are restricted to be elements of S. However, we will assume that this is covered by the construction of T_H and A_H, meaning that if $s, s' \in S$ then $T_H(s, s') \wedge A_H(s)$. In Formula 5.1, we use $>$ as the default order on the integers. Once we work with multiple loop headers (i.e. where $|H| > 1$) we will also introduce lexicographic ranking functions that map to tuples of non-negative integers rather than single non-negative integers. In this case, we replace $>$ by the lexicographic order $>_{\text{LEX}}$. In either case, the following theorem is a concrete version of Theorem 2.3.3 connecting loop head transitions, auxiliary invariants, and ranking functions to the termination of programs.

Theorem 5.1.1. Let P be a program with states S, loop head transition relation T_H with $|H| = 1$, and auxiliary invariants A_H. If there exists a ranking function f satisfying Formula 5.1 then P is terminating.

Proof. Let us assume the existence of a ranking function f satisfying Formula 5.1. We will show by contradiction that P must be terminating. Hence, assume that P is non-terminating and therefore has an infinite run $s_0, s_1, s_2, \ldots$ with all $s_i \in S$. By Theorem 2.3.1 there exists an infinite subsequence $s_{i_0}, s_{i_1}, s_{i_2}, \ldots$ such that $T_H(s_{i_j}, s_{i_{j+1}})$ holds for all s_{i_j} in the subsequence. Furthermore, by the definition of auxiliary invariant (cf. Section 2.3.1) we have that since s_{i_j} occurs inside a loop the auxiliary invariants hold (i.e. $A_H(s_{i_j})$ is true). We therefore have that since f is a ranking function satisfying Formula 5.1 every s_{i_j} in the subsequence satisfies

$$f(s_{i_j}) > f(s_{i_{j+1}})$$

and

$$f(s_{i_j}) \geq 0.$$

Now, consider the following infinite sequence of integers $f(s_{i_0}), f(s_{i_1}), f(s_{i_2}), \ldots$. This sequence is strictly monotonically decreasing and therefore has no least element. Furthermore, each element is non-negative. This contradicts the well-foundedness of the non-negative integers. □

For lexicographic ranking functions the argument is analogous but with the additional application of Theorem 2.3.2. In either case, Theorem 5.1.1 shows that proving the existence of a ranking function is a sufficient condition for proving termination of a program. This theorem builds the foundation of the remainder of this chapter, as we will focus on different techniques of synthesising these ranking functions.

5.2 SyGuS Problems for Termination Analysis

Let P be a program that we want to show terminating. For simplicity, we will also assume that P is restricted to unbounded integer and boolean values without nested loops. In addition, let T_H and A_H be the corresponding first-order formulas describing P's loop head transition relation and auxiliary invariants as described in Section 5.1. We now construct the following SyGuS problem Q:

Definition 5.2.1 (SyGuS Ranking Function Synthesis)**.** We define the ranking function synthesis problems in terms of a SyGuS problem $Q = (T, G, \phi, f)$ as follows:

- T is the theory `ALL`.
- G is the free grammar $*_G$ (cf. Section 2.2.1) generating all legal terms in T.
- ϕ is equivalent to Formula 5.1.
- f is a function variable of type $S \to \mathbb{Z}$.

Any solution to Q satisfies the semantic constraint ϕ and must therefore be a valid ranking function. Hence, by Theorem 5.1.1, this means that the original program P must be terminating. Note that if no solution is found, we cannot draw any conclusions about the termination of P since T_H and A_H are over-approximations as explained in Section 5.1.

Assuming functions to extract auxiliary invariants and transition relations have been implemented, Definition 5.2.1 gives rise to Algorithm 5 that simply constructs the corresponding SyGuS problem. We implement a tool, `RFSynth`, that implements Algorithm 5 and all required sub-procedures on input Java programs. As output, `RFSynth` produces a `SyGuS-IF` problem. Example 5.2.1 shows how this procedure works in practice. One of the advantages of constructing `SyGuS-IF` problems is that we can use any tool that accepts this standardised format as input. This includes `cvc5` as well as the reinforcement learning based tool presented in Chapter 4.

Algorithm 5: `Generate Ranking Synthesis`

Data: Program P

Result: SyGuS problem (T, G, ϕ, f)

$A_H \leftarrow$ `auxiliary_invariants`(P);

$T_H \leftarrow$ `loop_head_transition`(P);

$\phi = \forall s, s'.\ T_H(s, s') \wedge A_H(s) \implies f(s) > f(s') \wedge f(s) \geq 0$;

return (`ALL`, $*_G$, ϕ, f);

```
(set-logic ALL)

(synth-fun f ((x Int)(y Int)(z Int)) Int)

(declare-var xP Int)
(declare-var z!3 Int)
(declare-var x!15 Int)
(declare-var y!0 Int)
(declare-var x!M4 Int)
(declare-var zP Int)
(declare-var J9 Bool)
(declare-var yP Int)

;; Loop head transition
(assume (= J9 (<= x!M4 z!3)))
(assume (= x!15 (+ (- 2) x!M4)))
(assume (=> (not J9) (= xP x!15)))
(assume (not J9))

;; auxiliary invariants
(assume (= z!3 y!0))
(assume (= y!0 yP))
(assume (= z!3 zP))

;; ranking constraints
(constraint (> (f x!M4 y!0 z!3) (f xP yP zP)))
(constraint (>= (f x!M4 y!0 z!3) 0))

(check-synth)
```

Figure 5.2: Ranking function synthesis problem stated in SyGuS-IF.

Example 5.2.1 (Ranking Function Synthesis as SyGuS problem)**.** Consider the following code snippet that we already used in Section 2.3.1 where `x, y, z` are integers with a non-deterministic initialisation.

```
int x = *;
int y = *;
int z = *;
z = y;
while (x > z) {
  x = x - 2;
}
```

The resulting ranking function synthesis problem according to Definition 5.2.1 stated in `SyGuS-IF` is shown in Figure 5.2. After setting the logic to `ALL` we declare the synthesis target `f`. Since the program has three integer variables, we describe any state with a triple of integers $(x, y, z) \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$. Therefore, `f` is a function of type $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. Further since we do not want to restrict f syntactically, we omit the grammar specification. After the target definition, we need to declare the universally quantified variables that we need to describe states, transition relations, and invariants. To encode Formula 5.1, we use the `assume` and `constraint` commands provided by the `SyGuS-IF` standard. To this end, we first add the loop head transition relation relating the variables before the transition `x!M4, y!0, z!3` to the variables `xP, yP, zP` after the transition. These expressions correspond to the expressions obtained in Section 2.3.1 packed into `assume`-commands. Similarly, we add the auxiliary invariants to the list of assumptions. Finally, we add the ranking constraints: strict decrease between two states in the transition relation, and boundedness by 0. For the former, we use the variables describing the state before and after the transition:

```
(constraint (> (f x!M4 y!0 z!3) (f xP yP zP)))
```

For the latter, we impose a non-negativity as follows:

```
(constraint (>= (f x!M4 y!0 z!3) 0))
```

The resulting problem in the `SyGuS-IF` format can be given to any compatible SyGuS solver. In particular, using `cvc5`, we obtain the following solution:

```
(
  (define-fun f ((x Int) (y Int) (z Int)) Int (- x z))
)
```

In conclusion, we have proven the input program to be terminating and provide the ranking function $x - z$ as a certificate of termination.

5.3 Termination Analysis Problem Sets

As mentioned, termination analysis is an important problem in program analysis with historical roots at the very inception of computer science as a field of study. These days, many program analysis tools support termination as one of the base analysers and new techniques are continuously developed. To test the efficacy of new techniques as well as compare tools, annual competitions are organised. The two main competitions that feature termination analysis are

- Competition on Software Verification (SV-COMP) [204], and
- Termination Competition (TermComp) [205].

The former is a competition focused on software verification that also features competitions in other problems from software verification, such as memory safety, overflows, safety, etc. The latter is geared towards termination and complexity analysis of programs in general purpose programming languages as well as other models of computation. These include term rewriting and integer transition systems. Both competitions offer continuously growing benchmark sets collected from different sources. These problem sets allow researchers to test the efficacy of newly developed techniques and tools.

To evaluate the techniques presented in this thesis we collect a set of terminating benchmark programs to test our tools against. First, we collect problems from pre-existing problem sets of TermComp and SV-COMP. In particular, we focus on the two problem sets `Aprove_09` from TermComp and `term-crafted-lit` from SV-COMP. The latter comprises problems from literature on termination analysis, which are given as C code. Both problem sets are publicly available and cover a wide

variety of termination and non-termination problems as well as software verification in general. From these sets, we discard the non-terminating programs as they do not have ranking functions. We further restrict the set to only contain programs that

- have deterministic loops,
- use integer and boolean types,
- have a maximum of two nested loops, and
- have no function calls in the loops.

Deterministic loops in this case means that everything except for the initialisation of the program needs to be deterministic. After dropping problems due to the aforementioned constraints, we are left with 72 problems from `term-crafted-lit` (originally 159) and 38 problems (originally 76) from `Aprove_09`. Both sets contain problems with nested loops. We translate the problems into Java by hand. In addition, we also split the main functions' bodies into an initialisation and loop part. Note that this purely syntactic change does not alter the difficulty of determining termination of a program.

Existing tools in many cases are developed and continuously tested along the benchmark sets from the competitions. Hence, they are very good at solving these benchmarks. In fact, some problems might come from literature that introduces techniques implemented in some of the tools. As a result, there is a bias where termination analysers might perform significantly better on competition problems than "in the wild". To challenge them on "unseen" problems as well as showcase notable strengths and weaknesses we also create an additional problem set, called `NEW`. The problems in this set mostly exhibit the following two weaknesses that we identified in state-of-the-art tools;

- disjunctive loop guards, and
- non-linear loop guards with linear behaviour in the bodies.

```
int m = *;
int n = *;
int a = 0;
int b = 0;
while (a * b <= n or a * b <= m) {
    a = a + 1;
    b = b + 1;
}
```

Figure 5.3: Code snippet that has a non-linear and disjunctive loop guard but where the body exhibits "linear" behaviour (i.e. an increase by one for `a` and `b`).

An instance of a loop that exhibits behaviour that falls into both of these categories is shown in Figure 5.3. The loop has a guard that is disjunctive as well as non-linear. Both disjuncts contain the non-linear expression `a * b` where neither `a` nor `b` are constants. At the same time, the loop body has linear behaviour as each iteration only sees one increment for each `a` and `b`. The set `NEW` contains a total of 14 problems showcasing non-linear loop guards with linear bodies and disjunctive loop guards. In Appendix B we present additional problems from this dataset.

To summarise, in the remainder of this thesis, we will test termination analysis techniques on the following benchmark sets

- `Aprove_09` (TermComp),
- `term-crafted-lit` (SV-COMP), and
- `NEW` (created by us).

In total, we have 124 problems 18 of which contain nested loops. The exact numbers are shown in Table 5.1.

	`TermComp`	`SV-COMP`	`NEW`	Total
no nesting	31	61	14	106
nested	7	11	0	18
total	38	72	14	124

Table 5.1: Termination analysis benchmark sets with the number of problems without and without nesting for `Aprove_09` (TermComp), `term-crafted-lit` (SV-COMP), and `nuTerm-advantage` (nuTerm).

5.4 Experimental Evaluation

In this section, we will evaluate SyGuS based ranking function synthesis. We conduct an experimental evaluation using problems discussed in Section 5.3 without nested loops and answer the following research questions:

RQ1 Can SyGuS be used to synthesise ranking functions in termination analysis?

RQ2 Does SyGuS based ranking function synthesis advance the state of the art?

To answer the first question, we use the prototype `RFSynth` to generate SyGuS problems corresponding to the ranking function specification. Once extracted, we use `cvc5` to solve the resulting SyGuS problems. In addition, we also apply ϕ`ser`, the reinforcement learning based solver developed in Chapter 4 to the problems. In this case, we also add an explicit theory grammar to the SyGuS problem definition. We omit this when using `cvc5` to allow the tool to perform all possible optimisations. To answer *RQ2*, we compare ν`Term` to `Ultimate` [164], `Aprove` [159], and `DynamiTe` [153], which, collectively, represent the state of the art in termination analysis.

Setup: The experiments were conducted on Linux Kernel 5.15 running on an Intel Core i7 5820K at 3.3 GHz with 16 GB RAM. We ran all tools with a timeout of 60 seconds for each problem. Since `Ultimate` [164] and `DynamiTe` [153] do not support Java code as input we ran the experiments on the C versions of the problems.

5.4.1 Results

Can SyGuS be used to synthesise ranking functions in termination analysis? In total, `cvc5` produces ranking functions for 73 out of the 106 problems without nested loops. A more detailed breakdown of the results is shown in Table 5.2. We solve 22 of `TermComp`'s `Aprove_09` problem set, 44 (72.1%) of `SV-COMP`'s `term-crafted-lit` problem set, and 7 of our own problems. In contrast, using our own MCTS based solver ϕ`ser`, we solve 64 problems, overall. Note that in both cases we use the same algorithm to generate the SyGuS problems. The only

difference is that in one case we use cvc5 as a backend solver and in the other, we use ϕser. Given that overall, using cvc5, we solve 73 problems which is an efficacy of 68.9% we answer RQ1 in the affirmative.

	TermComp		SV-COMP		NEW		Total	
# tot.	31		61		14		106	
RFSynth/cvc5	22	71%	44	72.1%	7	50%	73	68.9%
RFSynth/ϕser	20	64.5%	38	59%	6	42.9%	64	62.3%
AProVE	27	87.1%	54	88.5%	3	21.4%	84	79.2%
Ultimate	27	87.1%	52	85.3%	1	7.1%	80	75.5%
DynamiTe	27	87.1%	40	65.6%	7	50%	74	69.8%

Table 5.2: Results of running RFSynth with cvc5 and ϕser as well as other state-of-the-art tools on all benchmarks without nested loops.

Does SyGuS based ranking function synthesis advance the state of the art? To compare with the state of the art, we also ran Ultimate [164], AProVE [159], and DynamiTe [153] on the same benchmarks. The results are also shown in Table 5.2. We observe that RFSynth in either configuration performs worse compared to the other tools on the TermComp problem set. Meanwhile, on the SV-COMP data set, we solve 44 and 38 problems using cvc5 and ϕser as SyGuS solvers, respectively. On this problem set AProVE comes in first (54 solved problems), with Ultimate close second (52 solved problems) while DynamiTe only solves 40 problems. Finally, on the NEW set, we solve 7 and 6 problems while DynamiTe also solves 7. Here, the remaining tools scored significantly lower, with AProVE solving 3 and Ultimate only solving 1. Compared to other tools, we see that SyGuS based termination analysis does not outperform state-of-the-art tools on any of the data sets. Nevertheless, we were able to outperform DynamiTe on the SV-COMP data set. Furthermore, on the NEW set we solve significantly more problems than either AProVE or Ultimate. However, we are unable to identify a class of problems where SyGuS based termination analysis clearly outperforms competing termination analysis tools. This leads us to conclude that using SyGuS solvers to produce ranking functions leads to comparable results but does not advance the state of the art. Thus, we answer RQ2 in the negative.

5.4.2 Discussion

The experimental evaluation showed that SyGuS based techniques can be used to synthesise ranking functions effectively. Using cvc5 as a SyGuS solver, we were able to solve 68.9% of all problems. In this approach, we have a clear separation of fronted (i.e. generation of SyGuS problem) and backend (i.e. solving of SyGuS problem). The two parts use SyGuS-IF as an interface language. This approach can be seen as a double-edged sword: On one hand, due to the fact that these backend solvers need to be applicable to general SyGuS, domain specific knowledge about ranking functions cannot be exploited. This drawback will be addressed later. On the other hand, we can use different backend solvers for the resulting problems that incorporate many optimisations, techniques, and theory specific tricks. For example, cvc5 is an enumerative solver, making it easy to solve SyGuS instances with small solutions even if the problem exhibits “complex” (e.g. non-linear) behaviour. Such an instance can be seen in the following loop:

```
int n = *;
int a = 0;
while (a * a * 4 <= n ) {
  a = a + 1;
}
```

Although the loop guard is non-linear, the ranking function is linear. In fact, cvc5 produces the following ranking function:

$$n, a \mapsto n - a$$

The resulting ranking function is very short, making it easy for cvc5 to find despite having to use non-linear arithmetic while Ultimate and AProVE are unable to prove this function terminating. Interestingly, cvc5 is also able to synthesise complex ranking functions with branching (i.e. if-then-else expressions). For example, we are able to synthesise the ranking function

$$n, a, b \mapsto \text{ite } (a \leq b) \ (n - a) \ (n - b)$$

for the following code snippet:

```
int n = *;
int a = 0;
int b = 0;
while (a * a <= n or b <= n) {
    a = a + 1;
    b = b + 1;
}
```

This problem is part of the `NEW` set and none of the competing tools were able to show this terminating. Nevertheless, unique solutions – solutions that are only solved by our tool – are rare. This is no surprise as the results presented in Section 5.4.1 show that this SyGuS based approach has only limited efficacy compared to other tools and does not improve on the state-of-the-art in any particular class of problems. One of the reasons for the poor performance compared to competing tools can be attributed to the maturity of those tools. `AProVE` and `Ultimate` in particular, are sophisticated pieces of software that have been in active development for over a decade. However, SyGuS based termination analysis exhibits more fundamental limitations that are inherent to the approach. The main drawbacks can be summarised using the following categories:

Lack of Domain Knowledge: This is the main drawback that we also mentioned earlier. SyGuS solvers are usually designed to support the whole specified standard. This allows for a general applicability of the solvers. However, the downside of this is that the solvers cannot make use of domain specific knowledge that might make the problem at hand more tractable. Conversely, techniques developed for general SyGuS might not be applicable to the domain of ranking function synthesis. This applies, for example, to the single invocation fragment [206] that `cvc5` is good at handling but cannot be applied to ranking function synthesis. In Chapter 6, we will address this issue and devise a machine learning based method that exploits domain knowledge in ranking function synthesis.

Insufficient Invariants: When generating the SyGuS problem, we model the program behaviour with auxiliary invariants A_H and transition invariants T_H.

These invariants need to be generated automatically. This can be problematic, as invariant synthesis in itself is a difficult problem and a matter of active research. This dependence is a major drawback as it requires us to solve a hard problem – invariant generation – before we solve the resulting synthesis problem, which is also a hard problem.

Large Size of Solutions: As discussed in Chapter 3, most SyGuS solvers are based on enumerative search. In our case, we use the free grammar (cf. Definition 5.2.1), which means that enumerative solvers will have difficulties finding expressions of large depth (i.e. large term size). Similarly, if the synthesis target has a large number of arguments, we also impede enumerative solvers. As a result, if input programs have many variables, enumerative solvers will also struggle to enumerate expressions. This problem persists even if the "correct" ranking function is simple (i.e. linear). In some cases, static analysis could be used to mitigate these issues. However, this would add another layer of complexity.

5.5 Threats to Validity

In this chapter, we developed a technique for ranking function synthesis for termination analysis based on SyGuS. We test this approach by evaluating an implementation of the presented techniques on a set of pre-existing benchmarks as well as a newly created set of problems. Such an empirical approach comes with threats to its validity. Apart from the obvious – incorrect implementations – there are also more subtle biases that can occur. In the following, we discuss some threats to the validity of this type of experimental evaluation.

Existing Benchmark Bias: All our claims depend on the choice of benchmark programs which were selected from pre-existing sets as discussed in Section 5.3. The problems we consider only make use of a small subset of Java and may not be representative of software that termination analysis may be applied to. Usually, programs are more complex and require more sophisticated techniques to analyse

their behaviour. For example, the programs we analyse do not make use of complex data types or concurrency. However, a comparison with state-of-the-art software in termination analysis shows that even these restrictions lead to benchmark sets that pose difficulties for state-of-the-art tools.

New Benchmark Bias: To evaluate the performance of our method, we create a new benchmark set `NEW` which falls under the same restrictions as the old benchmark set. These benchmarks were created by us specifically to showcase some weaknesses of existing tools. This introduces a bias against existing tools and techniques. To allow for a fair comparison, we also report on results for each of the data sets individually. Furthermore, most problems in the `NEW` problem set have disjunctive conditions and non-linear behaviour. While we believe that both features are important in commodity software, it remains to be quantified how large the benefit of supporting these features is on a larger repository of software.

Invariant Generation: As mentioned in Section 5.4.2, one of the drawbacks of the presented method is that we require an adequate set of transition and auxiliary invariants. The results of an end-to-end termination analysis are dependent on the quality of these invariants, and the tools we compare with use a variety of different approaches to solve this problem. Our tool uses simple methods based on heuristics for generating invariants as discussed in Section 5.1. Applying our method to a wider range of programs may require more sophisticated techniques for invariant generation.

5.6 Related Work

Unsurprisingly, we are not the first to make the connection from ranking function synthesis to SyGuS. Apart from other related work presented in Chapter 3, the contributions in this chapter build on [8] where a decidable fragment of synthesis problems is used to synthesise ranking functions. The synthesis problems are stated in a subset of the C programming language and the search space of functions is

a language based on a reduced instruction set (RISC). To compare, the authors also translate a small subset of the benchmarks to `SyGuS-IF` *by hand*. Hence, a comparison to our work is not tractable.

Similarly, in [114] the authors use a grammar driven technique to synthesise loop bounds for Constrained Horn Clauses (CHC). These loop bounds are stated using a grammar allowing linear expressions over the program variables. Once synthesised, a CHC solver is used to check the validity of the proposed loop bound. Neither of these synthesis based termination analysis techniques automatically generates `SyGuS-IF` problems from source programs. In contrast, we present an end-to-end translation from Java programs to SyGuS problems. This allows us to use highly engineered and optimised off-the-shelf solvers supporting `SyGuS-IF`, such as `cvc5`. By minimising the restrictions, we allow the backend solver to perform as many optimisations as possible.

Many other techniques that do not make use of SyGuS have been developed for ranking function synthesis and termination analysis. We refer to Chapter 3 for more details.

6
Neural Termination Analysis

In the previous chapter, we showed how function synthesis in the form of SyGuS can be used in proving termination of computer programs. However, an experimental evaluation has shown rather modest results. We hypothesise that one of the main limitations of this approach is the inability to utilise domain specific knowledge. In this chapter, we will present a solution to ranking function synthesis that builds on this hypothesis and exploits domain specific knowledge. By using the target programs to generate execution traces we are able to obtain an arbitrarily large amount of data describing the behaviour of the program. This allows us to employ neural networks to help synthesise ranking functions. This is also a solution to the data scarcity problem, as mentioned in Chapter 4, for the restricted setting of ranking function synthesis. The main contribution of this chapter is the development of a framework called *Neural Termination Analysis* (NTA) that is based on the principle that

finding a valid proof is harder than checking that a given proof is valid.

In our case, we use machine learning to "guess" a ranking function, followed by symbolic reasoning to "verify" it. Formally, this means that instead of proving the existence of a ranking function with the *second-order* formula

$$\exists f.\ \forall \vec{x}.\ \phi$$

as in the previous chapter, we train a neural network to act as a candidate ranking function $\hat{f}$. Once training has concluded, we verify the proposed ranking function by proving the simpler *first-order* formula

$$\forall \vec{x}.\ \phi[f \mapsto \hat{f}]. \tag{6.1}$$

The candidate ranking functions are trained on data generated from execution traces. To check whether a given candidate *neural ranking function* (NRF) $\hat{f}$ is correct (i.e. proving the aforementioned first-order formula), we use a satisfiability modulo theory (SMT) solver in conjunction with standard program analysis techniques discussed in Section 5.1. Thus, Neural Termination Analysis as a framework combines the three phases; data generation, learning, and verification to effectively train neural networks to act as ranking functions. The novelty of this work derives from the learning phase and how the other phases play into it. We provide concrete neural architectures and loss functions for training monolithic and lexicographic ranking functions. We implement these ideas in a proof of concept tool called ν`Term` to conduct an experimental evaluation with termination benchmarks established in Chapter 5. This demonstrates that the method is effective: most loops can be proven to be terminating using very small neural networks with at most a thousand tracing runs. Using a neural network with just one hidden layer and a straightforward training routine, this method discovers neural ranking functions for over 78% of the benchmarks. Our method subsumes a broad range of existing termination analysis strategies: not only do we discover linear ranking functions, but also ranking functions for problems that require piece-wise linear or lexicographic termination arguments [152, 164]. We observe that the ability of neural networks to represent non-linear functions enables termination proofs for programs that go beyond what the state of the art can handle. In particular, programs that use disjunctive or non-linear loop guards are proven terminating just as easily by our new technique. To summarise, the novel contributions of this chapter are:

- The designing of Neural Termination Analysis as a method of using machine learning for termination analysis.

- We present machine learning primitives such as loss functions and neural network architectures that are suitable for NTA.

- We implement NTA with a tracing set-up, neural network architectures with training procedures, and verification procedures using SMT.

- We test the efficacy of NTA on a standard set of benchmarks. By incorporating problems that state-of-the-art tools struggle to solve, we present the strengths of NTA.

6.1 Motivating Example

Many sound but incomplete techniques for determining termination of programs have been developed. As discussed in Chapter 3, these methods range from applying Farkas' lemma over recurrence relations all the way to term rewrite systems. Notably, in Chapter 5, we use ranking functions to prove termination of programs. It turns out that, for various reasons, many of these methods have trouble proving termination of programs that involve non-linear constraints or disjunctive loop guards. As an example of such a loop guard, consider the snippet in Figure 6.1. It is not

```
int x = ⋆;
int y = ⋆;
int z = ⋆;
while (x < y or x < z) {
  x++;
}
```

Figure 6.1: Loop with a simple increment in the loop body and disjunction in the loop guard.

difficult to convince yourself that this loop will terminate for every initialisation of the variables x, y, and z. However, actually proving that it terminates by providing a ranking function might not be as straightforward as it seems. Even though the loop guard only involves a disjunction of linear constraints, a linear

ranking function is insufficient to prove that this loop terminates. Naively, one might believe that the mapping

$$x, y, z \mapsto -x$$

is a correct ranking argument because it decreases with every loop iteration; however, it is not bounded from below. Hence, for any lower bound for x we can find a legal program state where x is below that bound. Similarly, for the mapping

$$x, y, z \mapsto y + z - x,$$

we can again always obtain a value that is smaller than any given constant coefficient with an adversary initialisation of x, y, and z, assuming they can be any unbounded integer. In fact, under this assumption, no linear combination of x, y, and z is a valid ranking function for this loop. However, by increasing the space of possible functions, we can find the following ranking function for the above loop:

$$x, y, z \mapsto \max\{y - x, 0\} + \max\{z - x, 0\}. \tag{6.2}$$

This function decreases in every iteration and is non-negative for every valuation of x, y, and z, meaning it is bounded from below by zero. Hence, this constitutes a correct ranking function that proves Figure 6.1 terminating. It turns out that synthesising this ranking function is already out of reach and too complex for the methods developed in Chapter 5. However, if we are given the ranking function from (6.2), it is easy to prove that it is a solution to the resulting SyGuS problem and therefore a correct ranking function. For example, the `define-fun` expression in Figure 6.2 represents the function (6.2) by making the max operation explicit. Using SMT solvers it is easy to prove that the presented solution is correct. This once again underlines the principle that verifying a solution candidate is significantly easier than coming up with a correct solution.

The main contribution of this chapter is a method for using neural networks to "guess" ranking function candidates, followed by formal methods to "check" these candidates. To this end, consider that neural networks are simply a different way

```
(
(define-fun f ((x Int) (y Int) (z Int)) Int
  (+
    (ite (> (- y x) 0) (- y x) 0))
    (ite (> (- z x) 0) (- z x) 0))
)
```

Figure 6.2: `SyGuS-IF` solution for the ranking function synthesis problem produced by the snippet in Figure 6.1.

of representing functions. And since ranking functions are just that – *functions*, we can use neural networks to represent them. For example, the ranking function described in 6.2 is equivalent to the neural network shown in Figure 6.3. In the remainder of this chapter, we will introduce and evaluate methods for synthesising such ranking neural networks. We argue that neural networks are a powerful model

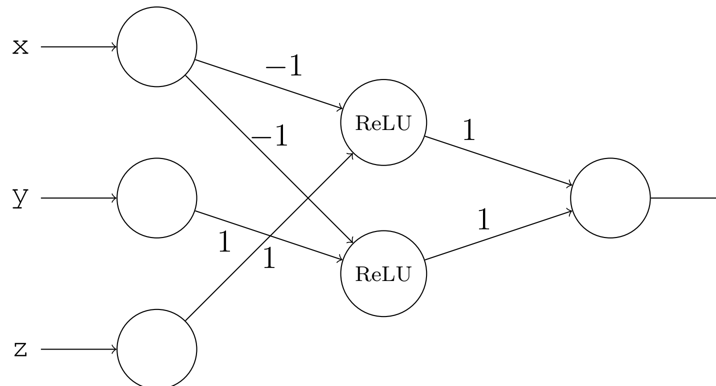


Figure 6.3: Neural network representing the function $\max\{y - x, 0\} + \max\{z - x, 0\}$.

to represent ranking functions of non-trivial programs. For example, loops that include disjunctions in their loop guards are not rare. Similar behaviour can be induced by conditional control flow, early breaks or by throwing exceptions. Suffice it to say that the following loop is semantically equivalent to the previous loop with a disjunction in the loop head:

```
while (true) {
  if (x >= y and x >= z)
    break;
  x++;
}
```

Moreover, we also argue that decoupling the process of guessing ranking functions from checking them enables us to effectively discover termination arguments for programs that involve non-linear constraints. As it turns out, neither `AProVE` nor `Ultimate` can determine with limited time that the following loop terminates:

```
while (x*x*x < y) {
  x++;
}
```

By contrast, our method learns and verifies the ranking function $\max\{\mathsf{y} - \mathsf{x}, 0\}$ in less than a second.

6.2 Neural Termination Analysis Framework

We propose neural termination analysis as a framework for termination analysis using neural networks as ranking functions. Figure 6.4 depicts a rough outline of the framework consisting of the three stages: tracing, learning, and verification. Neural networks are first trained to act as ranking functions using execution traces and subsequently verified using the program code. Thus, the three stages of *Neural Termination Analysis* are

1. the tracing/sampling strategy,
2. the neural network architecture with training setup, and
3. the verification procedure.

Notably, these steps are independent of each other. This means that changes to the learning setup only require a change in the second step. This may occur, for example, when considering different network models, learning frameworks, and training setups. Similarly, considering different input languages or alternative verification procedures only requires a change in the verification and (potentially) tracing procedure.

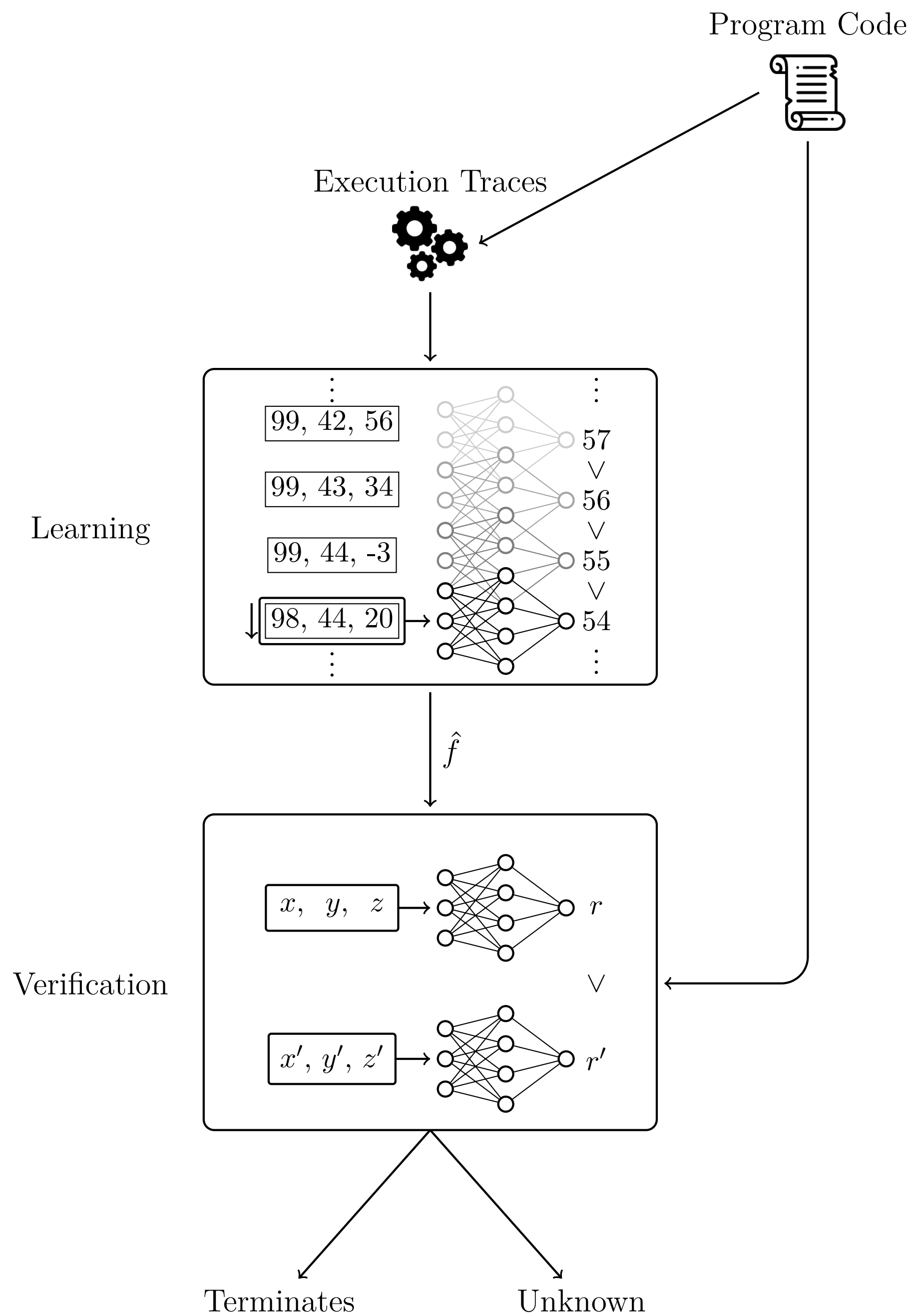


Figure 6.4: Schema of the entire Neural Termination Analysis framework consisting of tracing, training, and verification.

Tracing: In the first step depicted in Figure 6.4, we collect execution traces for a given program P that we want to show terminating. These execution traces are subsequently used as training data to train the neural ranking functions. These traces are obtained by first generating test input data for P and then taking memory snapshots during the execution of P with the generated test input data. Since the training phase exclusively works with these execution traces, it is important that they adequately represent the behaviour of the program P. We will discuss this step in more detail in Section 6.3.

Learning: In the second step, we use the execution traces from the previous step to train neural networks to become neural ranking functions. Like conventional ranking functions, neural ranking functions also need to be decreasing along consecutive program states and bounded from below. The training procedure minimises a loss function that punishes neural networks that do not decrease over the sampled observations. Hence, the decrease of the neural ranking functions is induced during the training of the networks. The second condition, boundedness from below, is induced through the network architecture. We use neural architectures with Rectified Linear Units (ReLU) as activation functions. These ensure that the neural network's output is bounded from below by 0. If the training is successful, we are left with a neural network that satisfies the conditions of a ranking function for all collected execution traces. In Section 6.4 we discuss this step in more detail.

Verification: Finally, we now have a neural network that, if training has concluded successfully, satisfies the ranking conditions for all collected execution traces. However, we also need to assert that the neural ranking function generalises to *all possible executions*, not just the sampled executions from the first step. To this end, we encode the neural network symbolically and use formal verification procedures discussed in Section 5.1 to check the correctness of the neural ranking function using SMT solvers. Upon an affirmative result, we conclude that the program terminates; upon a negative result, we return an inconclusive answer, that is, the program may or may not terminate. The verification procedure guarantees that our method is sound.

The combination of a particular tracing and sampling strategy, neural architecture, and verification procedure is an instance of our approach, which offers a flexible and extensible termination analysis framework that combines tracing methods, neural networks, and symbolic reasoning. In the following, we will describe each of the three stages in that order. We will draw our main focus to the second step – learning – as that constitutes the main contribution of this chapter.

6.3 Tracing and Data Generation

The first stage of NTA is concerned with generating and preparing data from the target program such that it can be used for training neural ranking functions later on. The input of this stage is a program that we want to show terminating, while the output is a family of sets $D = \{D_{l_0}, \ldots, D_{l_{m-1}}\}$ where each D_{l_i} is a set of feature vector pairs (o_t, o_{t+1}) representing related states t and $t+1$ at location l_i. These state pairs are obtained by

1. sampling input data for the given program,
2. monitoring the program's execution, and
3. taking memory snapshots every time a loop head is reached.

Finally, some data wrangling is performed to get the final data set D. In the following, we will describe each of the steps in more detail. The work in this section, while laborious in its engineering, does not describe any novel research contribution. We will therefore not go into much detail and refer to Appendix C for more engineering details.

6.3.1 Input Sampling

In order to generate program runs and traces, we need to define an initial state. In other words, to execute a function, we need to provide arguments (i.e. inputs) to the function. Since these traces are later used to train ranking functions, we want them to contain as much "information" about the loop's behaviour as possible. One way to

quantify the quality of a set of traces is by considering their coverage. As an example, consider the following loop guard from the motivating example in Section 6.1:

```
while (x < y or x < z)
```

If our data was collected from program runs with initial states where $x \geq z$ is always the case, then it would appear as if the loop only had the guard `x < y` instead of the whole disjunction `x < y or x < z`. Similar issues can arise when considering loop bodies with some form of branching (e.g. if-then-else constructs). Therefore, to avoid introducing any biases, we randomly sample input data from a normal distribution. This simple technique is facilitated by the fact that we work with programs that fall under the restrictions described in Section 5.3. Hence, we do not have to consider the case of complex data types and how to randomly generate them.

Related Area of Research: Many sophisticated techniques for automated generation of inputs optimising for coverage or other properties have been developed. Such techniques include but are not limited to usage of symbolic execution and model checking [207–209], fuzzing [210, 211], and reachability analysis [212]. Some techniques also use random sampling as a main driver behind input generation [213, 214]. In comparison, our generation of input data using random sampling is fast and does not require any static analysis. In this work, we refrain from using more sophisticated techniques. The effect that applying these techniques has on the results remains future work.

6.3.2 Generating Traces

In this section, we define execution traces and describe how they are generated from a given program P using arguments generated with the aforementioned techniques. As in Section 2.3.1 we let H be the set of all loop head locations of P.

Intuitively, a program trace is simply a representation of states in specific program locations (in our case these are the loop head locations) occurring in a run of that program. A program run as defined in Section 2.3.1 is a sequence of states related by the program's transition relation. Therefore, a program trace

can be thought of as a representation of a sub-sequence of a program run. We use the term "representation" since abstract program states defined in Section 2.3.1 are not amenable to learning models. Hence, we need to find a representation or feature vector of these states that can be used by learning models. We address this issue by constructing an *embedding*, defined by means of an observation function. An observation function $\omega : S \to \mathbb{R}^n$ extracts vectors of numerical values (i.e. feature vectors) from program states. Since the programs we consider only contain numerical data, we can assume that ω simply maps a state $s = \langle l, v \rangle$ to a vector containing the values of the variables in memory as follows,

$$\omega(\langle l, v \rangle) = [v(\texttt{var}_0), \dots, [v(\texttt{var}_{n-1})]. \tag{6.3}$$

The symbols $\texttt{var}_0 \dots, \texttt{var}_{n-1}$ are all variables occurring in the program in some fixed order. A trace is only a sub-sequence of an entire run of a program. Depending on what we want to analyse we may only consider states in certain locations. For termination analysis in particular, as described in Section 2.3.2, we care about states at loop head locations so that we can find a ranking for the loop head transition relation. We therefore define a trace for each loop head as follows:

Definition 6.3.1 (Program Trace)**.** Let $r = [s_0, s_1, \dots]$ be a finite run of a program P with initial state $s_0 = \langle l_0, v_0 \rangle$ and let l be the location of the loop head with guard G[1]. Now consider the sub-sequence $r_l = [s_{i_0}, s_{i_1}, \dots]$ of r consisting of all states in r with location l. Let L be the set of all *maximal* substrings of r_l for which G continuously holds. Formally, L consists of substrings $[\langle l, v_{i_k} \rangle, \langle l, v_{i_{k+1}} \rangle, \dots, \langle l, v_{i_{k+n}} \rangle]$ of r_l such that,

$$\begin{aligned} s_{i_{k-1}} &\not\models G, \\ s_{i_{k+n+1}} &\not\models G, \\ s_{i_{\{k,\dots,k+n\}}} &\models G. \end{aligned}$$

We now define the *program trace* $\mathcal{T}_{l,v_0}$ for loop l with input values v_0 as the set

$$\mathcal{T}_{l,v_0} = \{[\omega(a_0), \omega(a_1), \dots] \mid [a_0, a_1, \dots] \in L\}.$$

[1]We use the notation $s \models G$ to denote that a state s satisfies a guard G.

Intuitively, a program trace for location l is simply a program run that only considers states in l embedded into $\mathbb{R}^n$. Definition 6.3.1 seems more complicated than necessary. Indeed, for single loops, a simpler definition would suffice but the definition also covers the case of nested loops. Examples 6.3.1 and 6.3.2 showcase this difference. In spite of this seeming complexity, collecting program traces for single loop as well as nested loop programs is straightforward and can be done by means of monitoring the program execution. This procedure is comparable to a debugger with breakpoints at the location l. Hence, it is no surprise that in our implementation we use tooling that is also used for building debuggers. For more details on the implementation, we refer to Appendix C.

6.3.3 Data Wrangling

In the following, we describe a procedure to generate a data set consisting of pairs of feature vectors that can be used to train neural ranking functions. The procedure is quite intuitive as Examples 6.3.1 and 6.3.2 show and allows for a simple implementation.

As input, this procedure receives a target program P as well as a loop head l. As a result, we generate a set D_l of training pairs (o_t, o_{t+1}) where o_t and o_{t+1} are observations at iteration t and $t+1$ of the loop at location l. However, we create a separate set for each loop head location. Hence, the following steps describe how we produce the data set D_l for each loop head $l \in H$:

1. Generate a set V of input arguments for P using input sampling.

2. Collect a set of program traces $\mathcal{T}_{l,v}$ for each $v \in V$ and let

$$\mathcal{T}_l = \bigcup_{v \in V} \mathcal{T}_{l,v}.$$

3. Define a set of pairs D_l such that

$$D_l = \{(o_i, o_{i+1}) \mid [o_0, \dots, o_n] \in \mathcal{T}_l \text{ for each } \; i \in \{0, \dots, n-1\} \text{ with } n \geq 2\}.$$

The first two steps make use of input sampling and tracing, as introduced earlier. The third and final step creates pairs from the traces using a *sliding window* of size two, discarding traces of length 1. Using this sliding window, we create the set of observation pairs (o_t, o_{t+1}) at loop iterations t and $t+1$. Finally, we apply the procedure above to all loop heads in H to obtain the whole training data set,

$$D = \{D_l \mid l \in H\}.$$

The collection of sets D concludes the tracing phase of neural termination analysis as defined in Section 6.2. This set is now given as input to the learning stage. For a more intuitive understanding of the set D, consider Example 6.3.1 for the case of a program with a single loop and Example 6.3.2 for a program with two nested loops.

Example 6.3.1 (Generating Training Pairs for Single Loop)**.** Consider the loop in Figure 6.1. For this program, we define an embedding $\omega\colon S \to \mathbb{R}^3$ that observes the values of `x`, `y`, and `z` every time a run hits the entry location of the loop. We first sample initial arguments $(5, -3, 10)$ and produce the trace,

$$[(5, -3, 10), (6, -3, 10), (7, -3, 10), (8, -3, 10), (9, -3, 10)].$$

Note, that with these initial values, we always only satisfy the second disjunct of the loop guard. We repeat the process with the sampled arguments $(-2, 3, -5)$ resulting in the trace

$$[(-2, 3, -5), (-1, 3, -5), (0, 3, -5), (1, 3, -5), (2, 3, -5)].$$

Unlike the previous sampled arguments, these arguments lead to the first disjunct being satisfied. In the end, we arrive at the following set of 10 pairs of consecutive observations at loop head location l:

$$\begin{aligned} D_l = \{&((5, -3, 10), (6, -3, 10)), ((6, -3, 10), (7, -3, 10)), ((7, -3, 10), (8, -3, 10)), \\ &((8, -3, 10), (9, -3, 10)), ((-2, 3, -5), (-1, 3, -5)), ((-1, 3, -5), (0, 3, -5)), \\ &((0, 3, -5), (1, 3, -5)), ((1, 3, -5), (2, 3, -5))\} \end{aligned}$$

```
int i = ⋆;
int j = ⋆;
int k = ⋆;
while (i < k) {
  j = 0;
  while(j < i) {
    j++;
  }
  i++;
}
```

Figure 6.5: Code snippet with nested loops and three variables.

Example 6.3.2 (Generating Training Pairs for a Nested Loop)**.** Consider the program in Figure 6.5 with two nested loops. We associate the location of loop heads one and two with index 1 and index 2, respectively. We define the embedding $\omega\colon S \to \mathbb{R}^3$ and mapping states to the values of `i`, `j`, and `k`. Taking one trace with initial arguments $(0, 0, 3)$, we obtain the following trace for loop head 1:

$$[(0, 0, 3), (1, 0, 3), (2, 1, 3)]$$

The second loop head is a loop that is nested inside the first loop. Here, it becomes apparent why Definition 6.3.1 is as complex. In particular, we see that the trace $T_{2,(0,0,3)}$ contains three different *runs* of the loop, once for each iteration of the outer loop:

$$\mathcal{T}_{2,(0,0,3)} = \{[(1, 0, 4)], [(2, 0, 4), (2, 1, 4)], [(3, 0, 4), (3, 1, 4), (3, 2, 4)]\}$$

Finally, from this trace, we collect the following data sets using the sliding window of size two and discarding the trace $[(1, 0, 4)]$ of length one:

$$D_1 = \{((0, 0, 3), (1, 0, 3)), ((1, 0, 3), (2, 1, 3))\}$$

$$D_2 = \{((2, 0, 4), (2, 1, 4)), ((3, 0, 4), (3, 1, 4)), ((3, 1, 4), (3, 2, 4))\}$$

6.4 Learning Neural Ranking Functions

Neural ranking functions (NRFs) are neural networks that are trained to rank input vectors. Given a set of pairs describing a sequence that needs to be ranked we

train a neural network such that the predicted value (i.e. output of the neural network) of the first element is ranked above that of the second element. This ranking is trained by minimising a loss function that ensures that the neural network decreases by a discrete amount between every training pair. Within the context of neural termination analysis, the pairs are exactly the pairs obtained from execution traces as described in Section 6.3. We enforce the boundedness of the resulting NRFs using the architecture of the neural network. In this section, we present network architectures as well as two strategies for training neural ranking functions. The first is used to train a monolithic ranking argument, that is, a neural ranking function that outputs one value that decreases along the traces. The second strategy generalises the first and trains a lexicographic ranking argument, that is, a neural ranking function that outputs multiple values that decrease lexicographically. We use the second technique to synthesise rankings for nested loops.

6.4.1 Architecture for Neural Ranking Functions

The network architecture we use for neural ranking functions are simple feed-forward neural networks. These are functions $f\colon \Theta \times \mathbb{R}^n \to \mathbb{R}^m$ with n inputs, m outputs, and parameterised by $\theta \in \Theta$. The neural networks are defined in terms of interconnected neurons partitioned into one input, one output, and k intermediate layers which are also known as hidden layers. Each hidden layer is defined as a parameterised function,

$$\sigma(W, b; x) = \mathrm{ReLU}(Wx + b). \tag{6.4}$$

where W is a matrix of weights and b a vector of biases. The term $Wx + b$ defines an affine transformation of the input, while ReLU is a non-linear transformation defined as

$$\mathrm{ReLU}(x) = \max\{x, 0\}.$$

If x is a vector as in the definition of hidden layers then this is the element-wise application to each of the h outputs (i.e. neurons) in the respective hidden

layer as follows:

$$\mathrm{ReLU}(x_0, \ldots, x_{h-1}) = (\max\{x_0, 0\}, \ldots, \max\{x_{h-1}, 0\}). \tag{6.5}$$

Hence, an entire neural network consisting of multiple hidden layers can be defined as a parameterised function where the parameters of the network are a sequence of weight matrices and bias vectors $\theta = (W^{(0)}, b^{(0)}, \ldots, W^{(k-1)}, b^{(k-1)})$, each pair corresponds to a hidden layer. For neural ranking functions, we impose that the output layer has no bias and its weights $W^{(k)}$ are not trainable and non-negative. In combination, we define a neural ranking function f parameterised with θ:

$$f(\theta; x) = W^{(k)}\sigma(W^{(k-1)}, b^{(k-1)}; \cdot) \circ \cdots \circ \sigma(W^{(0)}, b^{(0)}; x) \tag{6.6}$$

Figure 6.6 shows a graphical interpretation of a neural ranking function with output dimension $m = 1$. Importantly, defining the network architecture of neural ranking functions as above allows us to derive the following theorem:

Theorem 6.4.1. The output of a neural ranking function is non-negative. Formally, we have

$$\forall x\ \theta.\ f(\theta; x) \geq 0.$$

Proof. From the definition of ReLU we have that the term $\sigma(W^{(k-1)}, b^{(k-1)}; \cdot) \circ \cdots \circ \sigma(W^{(0)}, b^{(0)}; x)$ is non-negative. In conjunction, with the fact that $W^{(k)}$ is defined to be non-negative, we have that $f(\theta; x) \geq 0$. □

Notably, we also get Theorem 6.4.1 if we relax the restrictions on $W^{(k)}$ and instead apply a ReLU to the output neuron. Importantly, this shows that a neural ranking function is always bounded from below by zero. Thus, $f(\theta; x)$ satisfies the first requirement of being a ranking function *by construction*, as the boundedness is baked into the architecture.

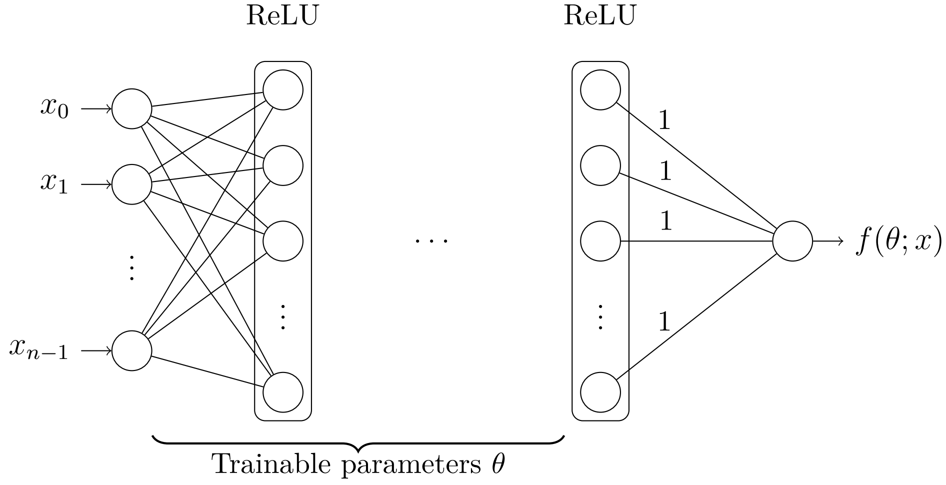


Figure 6.6: Architecture for monolithic NRFs with n input neurons where the last layer consists of constant weights 1.

6.4.2 Monolithic Ranking Loss

A monolithic neural ranking function is a special case where the output dimension $m = 1$, such as in Figure 6.6. This is used in case we only have a single loop head in our program. As a result, the set of loop head locations H is singleton, $H = \{l\}$. In Section 6.4.1, we defined neural ranking functions such that they are bounded from below. In order to be considered a ranking function, we have to enforce the second condition – a *strict decrease* along consecutive program states. In this section, we will describe how we train the neural ranking function to decrease along the sampled execution traces. As laid out in Section 6.3, these traces are supplied as training data $D = \{D_l\}$ (note that D is singleton since H is singleton) such that each $(o_t, o_{t+1}) \in D_l$ is a pair of observations at iteration t and $t + 1$ at a loop head. By construction, o_{t+1} occurs immediately after o_t in the loop location. Therefore, our goal is to train the NRF in such a way that its output decreases by a discrete amount $\delta > 0$ along each pair $(o, o') \in D_l$. Formally, a NRF $f(\theta; \cdot)$ is a ranking function *for the training data* D_l if

$$\forall (o, o') \in D_l.\ f(\theta; o') \leq f(\theta; o) - \delta. \tag{6.7}$$

Note that Equation (6.7) does not imply that it is a valid ranking for all inputs, but only for the inputs defined in D_l. In Section 6.5 we will discuss how we use

formal verification to check if a candidate $f(\theta;\cdot)$ also generalises to *all* possible inputs. However, before it comes to that, we have to find parameters θ for the neural ranking function $f(\theta;\cdot)$ such that (6.7) is satisfied. To this end, we solve the following optimisation problem:

$$\arg\min_{\theta} \frac{1}{|D_l|} \sum_{(o,o')\in D_l} \underbrace{\max\{f(\theta;o') - f(\theta;o) + \delta, 0\}}_{\mathcal{L}(o,o',\theta)} \tag{6.8}$$

We call the function $\mathcal{L}(o,o',\theta)$ the loss of the neural network over a given pair (o,o'). The higher the value of $\mathcal{L}$ the more the network increases over the pair, whereas for pairs that decrease by at least δ the value is always 0 (i.e., negative values do not affect the sum). This relationship is made explicit in Theorem 6.4.2.

Theorem 6.4.2. For all $(o,o') \in D_l$ we have that, if $\mathcal{L}(o,o',\theta) = 0$ then

$$f(\theta;o) \geq f(\theta;o') + \delta.$$

Proof. We will prove this implication by contraposition. Hence, we assume that $f(\theta;o) < f(\theta;o') + \delta$. By rearranging, and the definition of the loss, we get

$$0 < f(\theta;o') - f(\theta;o) + \delta = \mathcal{L}(o,o',\theta).$$

This means that $\mathcal{L}(o,o',\theta) \neq 0$ which concludes the proof by contraposition. □

Therefore, the result of the optimisation problem in (6.8) are parameters θ that ensure that the network decreases along all sampled traces. In other words, if we have weights θ such that the loss is 0 for all pairs in D_l then by Theorem 6.4.2, $f(\theta;\cdot)$ is decreasing by at least δ for each pair in D_l. This describes the second condition of ranking functions – a strict decrease along consecutive program states. The process of finding these parameters by minimising the average loss for each training pair constitutes the "training" of our neural networks using the training data D_l. In the context of synthesis, this can also be considered the synthesis procedure. Example 6.4.1 shows instances of neural ranking function parameters.

Example 6.4.1 (Parameters of Ranking Functions)**.** Consider Example 6.3.1 with the loop from Figure 6.1 with variables x, y, and z. Recall the example traces

$$[(5, -3, 10), (6, -3, 10), (7, -3, 10), (8, -3, 10), (9, -3, 10)],$$

and,

$$[(-2, 3, -5), (-1, 3, -5), (0, 3, -5), (1, 3, -5), (2, 3, -5)]$$

that we generated during the tracing phase. We use the monolithic architecture in Figure 6.3, which has exactly one hidden layer made of two neurons without a bias and an output layer of constant 1. We set $\delta = 1$ and train the network using the monolithic ranking loss. Once training has concluded, we obtain the following learned weights θ as the matrix

$$W = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Note that there may be several ranking functions. Hence, this matrix is not a unique solution, but just one potential solution. Assuming we arrived at the aforementioned weight matrix we get the neural ranking function

$$f(\theta; x, y, z) = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \mathrm{ReLU}(\begin{bmatrix} -1 & 0 & 1 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}).$$

This expression is equivalent to $\mathrm{ReLU}(z - x) + \mathrm{ReLU}(y - x)$ which when considering the definition of ReLU corresponds to the ranking function

$$x, y, z \mapsto \max\{y - x, 0\} + \max\{z - x, 0\}$$

that we derived in Section 6.1. Equivalently, the weights correspond to the neural network depicted in Figure 6.3. Importantly, the resulting NRF $f(\theta; x, y, z)$ defines a ranking to the training data. In particular, for each of the two traces above, the NRF maps the sequence of states to the decreasing sequence

$$5, 4, 3, 2, 1.$$

Hence, if the sliding window of size two constructed from the two traces constitute the entire training data set D_l, $f(\theta; x, y, z)$ defines a correct ranking function for D_l as it is decreasing for pairs of consecutive states and bounded from below.

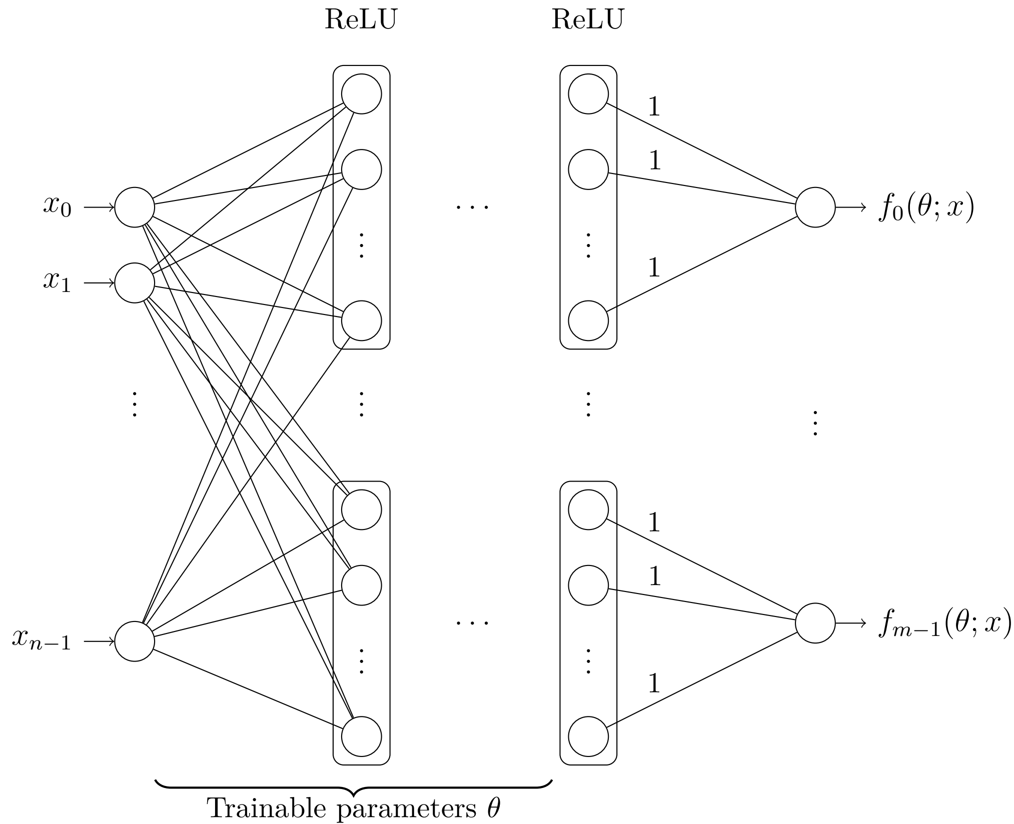


Figure 6.7: Architecture for lexicographic NRFs.

6.4.3 Lexicographic Ranking Loss

Neural ranking functions with more than one output neuron, that is $m > 1$, are considered lexicographic ranking functions. Hence, a lexicographic NRF can be described as a function, $\boldsymbol{f}_{0,\ldots,m-1}(\theta; x) = [f_0(\theta; \cdot), \ldots, f_{m-1}(\theta; \cdot)]$ where each $f_i(\theta; \cdot)$ is defined as in Equation (6.6). This is illustrated in Figure 6.7. Lexicographic arguments are suitable for programs with nested loops. We use m-dimensional lexicographic ranking functions for m nested loops. Hence, the set of loop head locations H has size m. Unlike the monolithic case where we simply require a decrease of at least δ on the output, we now need to ensure a *lexicographic* decrease of at least δ. To this end, we associate with each loop header an index $i \in \{0, \ldots, m-1\}$, such that the inner nested loop has a larger index than the outer loop. A lexicographic decrease occurs if there is a decrease by δ in the output neuron at some index $i \in \{0, \ldots, m-1\}$, while the outputs of all indices

$j \in \{0, \ldots, i-1\}$ do not increase. When considering the set of training pairs $D = \{D_0, \ldots D_{m-1}\}$ (cf. Section 6.3), we want to train the lexicographic neural ranking function $\boldsymbol{f}_{0,\ldots,m-1}(\theta; x)$ such that for all $i \in \{0, \ldots, m-1\}$ and for all $(o, o') \in D_i$ the corresponding output neurons decrease lexicographically as follows:

$$f_i(\theta; o') \leq f_i(\theta; o) - \delta \qquad \text{and} \tag{6.9}$$

$$f_j(\theta; o') \leq f_j(\theta; o) \qquad \text{for all } j < i. \tag{6.10}$$

Again, training is done using a loss function that is minimised during the training phase. In particular, we solve the following optimisation problem to obtain the best parameters θ:

$$\arg\min_{\theta} \frac{1}{|D_0| + \cdots + |D_{m-1}|} \sum_{i=0}^{m-1} \sum_{(o,o') \in D_i} \mathcal{L}_i(o, o', \theta) \tag{6.11}$$

In other words, the optimisation problem tries to minimise the average loss per training pair. The loss for each pair is determined by the dataset it belongs to. Hence, for index i, the loss is defined as

$$\mathcal{L}_i(o, o', \theta) = \max\{f_i(\theta; o') - f_i(\theta; o) + \delta, 0\} + \sum_{j=0}^{i-1} \max\{f_j(\theta; o') - f_j(\theta; o), 0\}. \tag{6.12}$$

The first term corresponds to (6.9) and is equivalent to the monolithic loss described in Equation (6.8). The second term ensures that no increase occurs in all output neurons with index 0 to $i-1$ (cf. (6.10)). The loss $\mathcal{L}_i(o, o', \theta)$ attains its minimal value 0 when both conditions are satisfied. Hence, if the optimisation problem in (6.11) is solved all samples satisfy these conditions and the neural network constitutes a lexicographic neural ranking function over the sampled traces. Example 6.4.2 shows training using the lexicographic running example from Section 6.3.

Example 6.4.2. In Example 6.3.2, we showed how the following training pairs are generated for the nested loops in Figure 6.5:

$$D_0 = \{((0, 0, 3), (1, 0, 3)), ((1, 0, 3), (2, 1, 3))\}$$

$$D_1 = \{((2, 0, 4), (2, 1, 4)), ((3, 0, 4), (3, 1, 4)), ((3, 1, 4), (3, 2, 4))\}$$

We use a neural network as in Figure 6.7 with one hidden layer and one hidden neuron in each of the two blocks. During training the first output neuron converges to the following function:

$$f_0(i, j, k) = \max\{k - i, 0\}. \tag{6.13}$$

However, in the second output, the optimisation procedure may converge to either of the following functions:

$$f_1(i, j, k) = \max\{i - j, 0\} \tag{6.14}$$

$$f_1(i, j, k) = \max\{k - j, 0\}. \tag{6.15}$$

While both functions are valid rankings, proving that they are in fact valid requires different auxiliary invariants. In Section 2.3, we describe methods for extracting invariants. In particular, proving that (6.15) decreases along the inner loop requires the auxiliary invariant $i < k$ while checking that (6.13) decreases along the outer loop requires an argument that the inner loop leaves k and i unchanged. This ambiguity with its dependence on auxiliary invariants is a drawback of our method that we will discuss in Section 6.6.3.

6.5 Verification

The verification phase is the third and last phase of the neural termination analysis framework. The input to the verification procedure is a neural ranking function $f(\theta; \cdot)$ and the original program P that we want to show terminating. The NRF $f(\theta; \cdot)$ has weights θ trained on the sampled execution traces in the data set D which is generated using P. The purpose of this phase is to *formally* verify that $f(\theta; \cdot)$ not only is a valid ranking for D but also generalises to all possible states at the respective loop heads H of P. Therefore, we construct the same symbolic encoding as described in Section 2.3 and Section 5.1 for the auxiliary invariants A_H and loop head transition T_H in P. Since $f(\theta; \cdot)$ is bounded from below by construction, all that remains to be shown is that it decreases along consecutive states in T_H,

$$\forall s, s' \colon (s, s') \in T_H \land s \in A_H \implies f(\theta; \omega(s)) \geq f(\theta; \omega(s')) + \delta. \tag{6.16}$$

Note that θ, which has been trained in the learning phase, is constant now. If this formula is valid, we have that $f(\theta; \cdot)$ is a correct ranking function. Once again, to perform this verification task, we use SMT solvers. As in Section 2.3, we use the theory of integer arithmetic to encode the program semantics and states. To this end, we need to define a discretisation procedure that turns the function $f\colon \Theta \times \mathbb{R}^n \to \mathbb{R}$ into a symbolic function $\hat{f}\colon \mathbb{Z}^n \to \mathbb{Z}$ where $\mathbb{Z}^n$ is the encoding of the program state. One advantage of using the theory of integers is the fact that we can replace the greater equals with a strict inequality and discard δ in (6.16). In the end, we use an SMT solver to prove the validity of

$$\forall s, s'\colon (s, s') \in T_H \wedge s \in A_H \implies \hat{f}(s) > \hat{f}(s'). \tag{6.17}$$

If this is valid, we have found a ranking function $\hat{f}$ proving termination of P.

Related Areas of Research: The methods presented in this section are related to two active areas of research: *Quantisation of Neural Networks* and *Formal Verification of Neural Networks*. The main motivation for the former is a reduced memory footprint, as well as more efficient inferences for neural networks [215–217]. In the latter, formal guarantees against adversarial attacks as well as other safety and explainability concerns have driven research [218–231]. While we do not provide any novel contributions to either of these areas, we believe that our setting adds a new and so far unexplored application to both areas. Further, applying more sophisticated techniques developed in those fields may benefit the efficacy of NTA. We leave it as future work to connect the work presented in this thesis to either of the aforementioned areas of research.

6.5.1 Discretising Neural Ranking Functions

Our neural networks are compositions of linear layers and ReLU activation functions. As defined in Section 6.4.1, a NRF is just a series of weights and biases that are multiplied by the inputs. As usual in machine learning, these weights and biases are floating point matrices and vectors. In order to turn NRFs into symbolic expressions

that can be fed to SMT solvers in the form of first-order logic formulas we first apply the multiplications defined by the matrix-vector multiplication to symbolic variables. A layer $\sigma(W, b; x)$ of a neural ranking function is defined as

$$\sigma(W, b; x) = \text{ReLU}\left(\begin{bmatrix} w_{0,0} & \dots & w_{0,n-1} \\ \vdots & \ddots & \vdots \\ w_{m-1,0} & \dots & w_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ \vdots \\ x_{m-1} \end{bmatrix} + \begin{bmatrix} b_0 \\ \vdots \\ b_{m-1} \end{bmatrix}\right) \tag{6.18}$$

in (6.4). Since standard SMT solvers in integer arithmetic do not support matrices and vectors, we construct the following vector of symbolic expression with symbolic variables $x_0, \dots x_{m-1}$ from $\sigma(W, b; x)$,

$$\text{ReLU}\left(\begin{bmatrix} (w_{0,0} * x_0 + \dots w_{0,n-1} * x_{m-1}) + b_0 \\ \vdots \\ (w_{m-1,0} * x_0 + \dots + w_{m-1,n-1} * x_{m-1}) + b_{m-1} \end{bmatrix}\right).$$

We apply this operation recursively to all layers where the input vector of the first layer is the vector of symbolic variables $[x_0, \dots x_{m-1}]$ while the input of layer $i > 0$ is the vector of symbolic expressions obtained from performing this operation to layer $i - 1$. As the last layer W^k of a neural ranking function as in (6.6) is a constant vector, we finally end up with the expression

$$\begin{bmatrix} W_0^k & \dots & W_{m-1}^k \end{bmatrix} \begin{bmatrix} \text{ReLU}(e_0) \\ \vdots \\ \text{ReLU}(e_{m-1}) \end{bmatrix} = W_0^k * \text{ReLU}(e_0) + \dots + W_{m-1}^k * \text{ReLU}(e_{m-1}),$$

where $e_1, \dots e_{m-1}$ are expressions from applying this operation recursively to the previous layers. Now all that is left to do is turn the floating point weights and biases into integers. We do so by simply rounding the values to the nearest integer, or alternatively, multiplying the values by a constant factor and then rounding them. In either case, we end up with an expression applying multiplication, addition, and ReLU to integer scalar values and symbolic variables $x_0, \dots, x_{m-1}$. This discretisation procedure translates a neural ranking function $f(\theta, b; \cdot)$ into a symbolic function $\hat{f}$ that a standard SMT solver can work with, allowing us to use SMT solvers to check for the validity of Formula (6.17). Example 6.5.1 shows this procedure.

6.5.2 Verifying Lexicographic Ranking Functions

For lexicographic neural ranking functions, we verify the validity of the conditions in Eq. (6.9) and (6.10) over each loop head and the respective output component of the neural ranking function. Let $H = \{l_0, \ldots, l_{m-1}\}$ be the set of loop headers and $\boldsymbol{f}_{0,\ldots,m-1}(\theta; x) = [f_0(\theta; \cdot), \ldots, f_{m-1}(\theta; \cdot)]$ be the neural ranking function obtained after training. We apply the same procedure of performing symbolic operations to variables and expressions, and rounding to the nearest integers as in the monolithic case. In the end, we obtain the vector of symbolic functions $[\hat{f}_0(\theta; \cdot), \ldots, \hat{f}_{m-1}(\theta; \cdot)]$. To check that this is indeed a correct lexicographic ranking function, we use an SMT solver to verify for every $i \in \{0, \ldots, m-1\}$ the validity of the following conditions:

$$\begin{aligned}
\forall s, s' \colon (s, s') \in T_{l_i} \wedge s \in A_{l_i} &\implies \hat{f}_i(s) > \hat{f}_i s') \\
\forall s, s' \colon (s, s') \in T_{l_i} \wedge s \in A_{l_i} &\implies \hat{f}_{i-1}(s) \geq \hat{f}_{i-1}(s') \\
&\vdots \\
\forall s, s' \colon (s, s') \in T_{l_i} \wedge s \in A_{l_i} &\implies \hat{f}_0(s) \geq \hat{f}_0(s').
\end{aligned}$$

Note that each condition can be checked independently. For a ranking function to be valid, each of the conditions needs to be valid. We remark that, unlike the monolithic case, T_{l_i} may represent runs of arbitrary length when the loop with loop head l_i has nested loops inside its body. To encode T_{l_i}, we substitute every inner loop with a summary of that loop. Several methods have been developed for this purpose [155, 158, 232]. As a heuristic, we construct a loop summary that encodes the invariants of all variables that are never assigned within the loop, together with a transition invariant that encodes the respective (and previously verified) lexicographic component in the neural ranking function and the respective auxiliary invariant. Using or developing more sophisticated summarisation techniques is a matter of future research.

Example 6.5.1 (Checking the Validity of a Neural Ranking Function)**.** Consider once again the program from Figure 6.1. In Example 6.4.1 we have derived the

ranking function

$$\begin{aligned} f(\theta; x, y, z) &= \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \text{ReLU}(\begin{bmatrix} -1 & 0 & 1 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}) \\ &= \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \text{ReLU}(-1 * x + 0 * y + 1 * z) \\ \text{ReLU}(-1 * x + 1 * y + 0 * z) \\ \text{ReLU}(0 * x + 0 * y + 0 * z) \end{bmatrix} \\ &= 1 * \text{ReLU}(z - x) + 1 * \text{ReLU}(y - x) + \text{ReLU}(0) \\ &= \max\{z - x, 0\} + \max\{y - x, 0\}. \end{aligned}$$

Since all the weights consist of 0 and 1, rounding to the closest integer does not change any values. Hence, we arrive at the SMT LIA function $\hat{f}(x, y, z) = \max\{z - x, 0\} + \max\{y - x, 0\}$. With the same static analysis techniques as in Chapter 5 to derive the loop head transition relation

$$x' = x + 1$$

and the auxiliary invariants

$$(x < y \lor x < z) \land y' = y \land z' = z,$$

where x, y, z and x', y', z' are integer variables describing the states before and after the iteration, respectively. Using an SMT solver we can now prove,

$$(x < y \lor x < z) \land x' = x + 1 \land y' = y \land z' = z \implies \hat{f}(x, y, z) > \hat{f}(x', y', z').$$

The validity of this formula confirms that $\hat{f}$ decreases by at least 1 for every possible assignment of the variables (remember, $\hat{f}$ maps to the set of integers). In addition, f and therefore $\hat{f}$ is bounded from below by construction owing to the use of ReLUs.

6.6 Experimental Evaluation

We have presented Neural Termination Analysis (NTA) as a framework termination analysis consisting of tracing, learning, and verification. This framework utilises neural networks to synthesise neural ranking functions (NRFs). In the previous section, we showed multiple architectures as well as training set-ups. To test

the efficacy of NTA, we developed a prototype implementation of the methods discussed in the previous sections for proving termination of Java methods, which we name ν`Term`. Our implementation strictly separates tracing, learning, and verification as discussed in Section 6.2. We use this implementation to answer the following three research questions:

RQ1 Can NRFs be used to formally prove the termination of programs?

RQ2 Do NRFs advance the state of the art in termination analysis?

RQ3 How do NRFs scale in terms of the complexity of the program?

To obtain program traces, we first sample input data for the program in question using a multivariate normal distribution. Subsequently, we execute the program using this test data while maintaining control of the execution and collecting memory snapshots using the Java Virtual Machine Tool Interface (JVMTI). For more details on sampling and tracing, we refer to Appendix C.

Once tracing is completed, this data is used to train the neural ranking function, where PyTorch [233] is used as the machine learning framework. We use PyTorch's implementation of the Adam optimisation algorithm [234] for training the neural networks. As architecture for neural ranking functions, we use a single hidden layer with bias followed by a constant layer consisting of a vector of ones. Finally, we encode the problem of certifying the neural ranking function into an SMT formula and use Z3 [202] to solve it. More details about the implementation of this step can be found in Section 5.1 and Section 6.5.

6.6.1 Benchmarks and Setup

To answer *RQ1* and *RQ2* we use the set of termination problems presented in Section 5.3 and also work with the same restrictions and assumptions discussed there. Furthermore, in the case of *RQ2*, we compare ν`Term` to `Ultimate` [164], `AProVE` [159], and `DynamiTe` [153], which, collectively, represent the state of the art in termination analysis.

Setup: The experiments were conducted on Linux Kernel 5.15 running on an Intel Core i7 5820K at 3.3 GHz with 16 GB RAM and an NVIDIA GTX 980 graphics card. For learning, we use the Adam optimiser provided by PyTorch and a learning rate of 0.05. We ran the benchmarks 5 times with random seeds that were fixed a priori for reproducibility. Full instructions on how to reproduce the results (including the seeds) are part of the supplementary material. We ran all tools with a timeout of 60 seconds for each problem.

6.6.2 Results

The results of running νTerm on the problem different problem sets are presented in Table 6.1. In the following, we use the results to answer the three research questions.

	TermComp		SV-COMP		NEW		Total	
# tot.	38		72		14		124	
νTerm	34.6	91%	49.6	68.9%	13	92.9%	97.2	78.4%
AProVE	34	89.5%	64	88.9%	3	21.4%	102	81.5%
Ultimate	34	87.1%	61	84.5%	1	7.8%	96	77.4%
DynamiTe	31	81.6%	46	63.9%	7	50%	84	67.7%

Table 6.1: Results of running νTerm (with 7 neurons), AProVE, Ultimate, and DynamiTe on SV-COMP, TermComp, and NEW. In the case of νTerm we report the average results rounded to the first decimal. The last two columns show the union of the first two problem sets and all three problem sets, respectively.

Can neural ranking functions be used to formally prove the termination of software programs? To answer this question, we ran νTerm on the three benchmark sets mentioned above. The results are given in Table 6.1. We observe the best performance of νTerm when using a neural network consisting of 7 neurons with 1000 sample traces with a maximum length of 1000. νTerm proves termination for 97.2 out of 124 problems on average (100 in the best out of the five runs), which accounts for 78.4% of the problems in the problem set. When considering the different problem sets separately, we solve 91.0% (TermComp), 68.9% (SV-COMP), and 92.9% (NEW) of the problems. Even when disregarding the NEW set, νTerm solves 76.6% of the problems.

In Table 6.2 we present the same results, but where the benchmarks are split into problems with and without nested loops. Here, we can see that νTerm solves 79.43% of all single loop problems and 71.11% of all nested problems. Given that very simple neural networks suffice to prove termination of a substantial subset of the standard benchmarks, we answer *RQ1* in the affirmative.

	no nesting		nested	
# tot.	106		18	
νTerm	84.2	79.43%	12.8	71.11%
RFSynth/cvc5	73	68.9%	–	–
AProVE	81	76.4%	17	94.4%
Ultimate	79	74.5%	16	88.9%
DynamiTe	67	63.2%	10	55.6%

Table 6.2: Comparing results when split up into problems with nested loops and without nested loops. We also add the results from Chapter 5 of running RFSynth with cvc5 to compare.

Do neural ranking functions advance the state of the art in termination analysis? To compare with the state of the art, we also ran Ultimate [164], AProVE [159], and DynamiTe [153] on the same benchmarks. Since Ultimate [164] and DynamiTe [153] do not support Java code as input, we ran the experiments on the C versions of the problems. The results are presented in Table 6.1. Overall, the strongest tool is AProVE, which solves 81.4% of all problems, followed by νTerm with 78.4% and Ultimate with 77.4% and finally DynamiTe with 67.7%. By considering the pre-existing data sets separately, we see that νTerm comes in first on the TermComp set and third on SV-COMP. On these two sets combined, νTerm solves 76.6% of the problems with AProVE and Ultimate solving 89.1% and 86.4% respectively and DynamiTe trailing with 70% of problems solved. Table 6.2 shows a split of the results into problems with and without nested loops. We observe that νTerm outperforms all other tools on the problems without nesting while lagging behind AProVE and Ultimate on the problems with nested loops. We also see that neural termination analysis improves the performance of SyGuS based

termination analysis by over 10%. We conclude that, on the existing benchmarks, ν`Term` performs comparably to the state of the art and even beats it on some subsets of the problems.

We hypothesise that ν`Term` significantly advances the state of the art when applied to programs that have either disjunctive loop conditions or programs that are non-linear. As discussed in Section 5.3, the existing benchmark sets suffer from confirmation bias and focus on programs that avoid these features. We, therefore, introduced the set `NEW` with programs that feature

1. non-linear conditions, and
2. disjunctions in conditions.

For example, consider the following code snippet contained in the `NEW` problem set:

```
int n = *;
int a = 0;
while ( a*a*4 <= n ) {
  a = a + 1;
}
```

This loop only uses two variables, a and n, where n remains constant throughout the execution while a is incremented by 1 in every iteration. Despite the fact that the loop guard $a^2 \cdot 4 \leq n$ is non-linear, there is a linear ranking function. Furthermore, the execution traces of the loop only show the increment of a, which is also linear. ν`Term` can solve this problem with a tiny neural network, consisting of a single neuron, and reports the ranking function $\mathrm{ReLU}(n - a + 1)$. Neither `AProVE` nor `Ultimate` can prove termination of this problem, but `DynamiTe`, a tool which also utilises execution traces, can solve it.

In addition to non-linear behaviour, disjunctions also increase the complexity of formal reasoning significantly. The combination of features is illustrated by the code shown in Figure 6.8. None of the tools we compare with can show termination of this loop, while ν`Term` proves termination by learning the ranking function $\mathrm{ReLU}(m-a+2)+\mathrm{ReLU}(n-b+2)$. Note that `DynamiTe` is able to show termination

```
int m = ⋆;
int n = ⋆;
int a = 0;
int b = 0;
while ( a*a <= m or b*b <= n ) {
  a = a + 1;
  b = b + 1;
}
```

Figure 6.8: Loop with non-linear and disjunctive loop guard from the NEW set.

if the loop head was either of the disjuncts without the other. However, once both conditions are connected with a disjunction, DynamiTe fails to prove termination.

The NEW problem set comprises problems that exhibit either non-linear conditions, disjunctions, or a combination of both. For this set, νTerm solves on average 92.9% of the problems while DynamiTe comes second, solving 50%, followed by AProVE (21.4%) and Ultimate (7.8%).

In conclusion, the experiments conducted and presented in Table 6.1 show that on existing benchmarks, νTerm either performs comparable to or stronger than (e.g., on TermComp) the state of the art. Furthermore, we identified weaknesses in the existing tools when considering a broader range of programs and showed that neural termination analysis can solve these problems by providing a set of programs on which νTerm outperforms all existing tools by a large margin. Thus, we can answer *RQ2* in the affirmative.

How do neural ranking functions scale in terms of the complexity of the program? Our experiments have only required tiny neural networks, consisting of no more than 10 neurons. Running the experiments with a variable number of neurons does not yield significant performance gains on the existing problem sets. This is shown in Figure 6.9 where we report the percentage of solved problems for varying numbers of neurons. There is a significant jump in the beginning where we go from one neuron to two neurons. However, any gains after that are small. We hypothesise that programmers avoid writing loops that require termination arguments that depend on a very large number of variables. To evaluate how

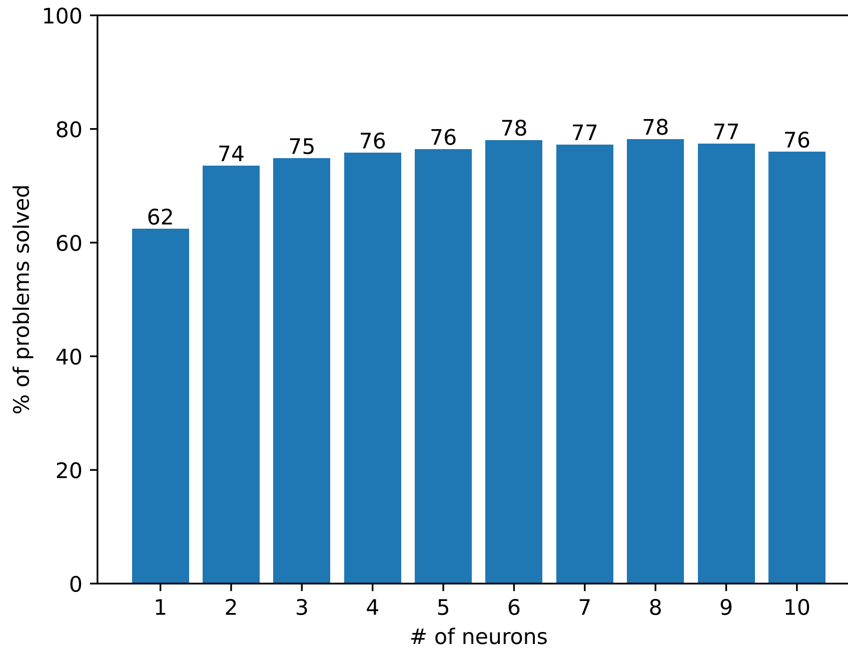


Figure 6.9: Percentage of problems solved for neural networks with 1 to 10 hidden neurons.

```
int n_1 = *;
...
int n_k = *;
int a = 0;
while ( a*a*4 <= n_1 or ... or a*a*4 <= n_k ) {
  a++;
}
```

Figure 6.10: Code snippet template with variables a and $n_1, \ldots n_k$ a single loop with a single increment in the loop body. The loop guard consists of k disjuncts.

our technique scales in the number of variables that are required for the ranking argument, we use the program template presented in Figure 6.10, designed to require at least k neurons. We include the instances of this template for values of k up to 4 in the `NEW` problem set. The instances for $k = 3, k = 4$ are also shown in Appendix B. Note that neither `AProVE` nor `Ultimate` can solve this problem for any value of k; `DynamiTe` is able to solve these problems for k up to 2 (with a noticeable increase in runtime from 8 s to 30 s), but times out for any larger k. Note that ν`Term` solves these problems in the problem set within 2 seconds. Trying programs with k up to 10 it becomes clear that the learning procedure

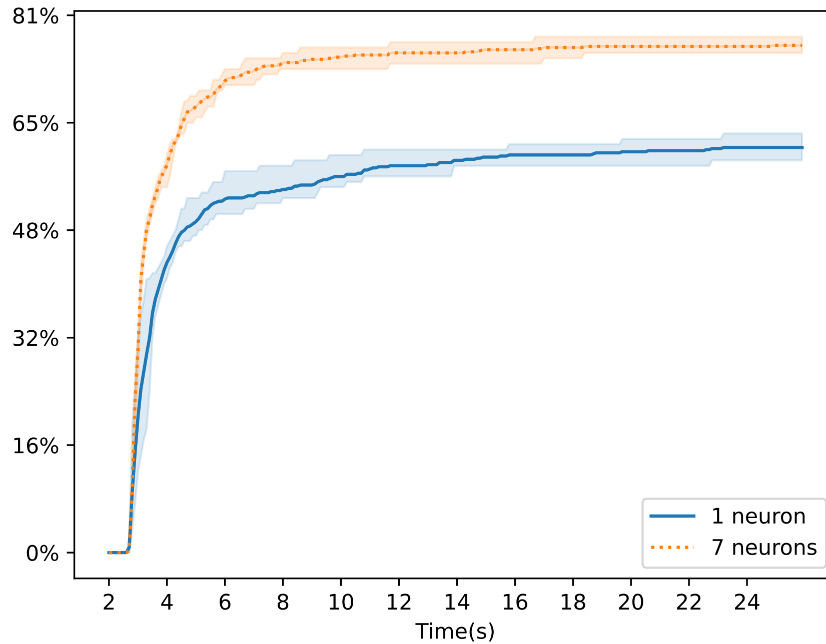


Figure 6.11: We plot what percentage (as part of the whole problem set) of problems is solved within a certain time budget.

continues to scale well. This can further be seen in Figure 6.11, where we plot how many problems two neural ranking function architectures solve within a certain time budget. We compare an architecture with 1 neuron to an architecture with 7 neurons. In both cases, the vast majority of problems are solved within 6 seconds, regardless of the number of neurons. We see that the larger network is able to solve more problems overall without having to compromise on speed.

It is worth emphasising that neural networks with 7 neurons, for example, which would be able to prove such a loop for $k = 7$ terminating, are laughably small compared to state-of-the-art neural networks used in other areas such as natural language processing. Hence, it is likely that by increasing the size of the neural network one would sooner run into issues verifying the neural networks and generating meaningful traces than training the neural networks.

Furthermore, we hypothesise that the vast majority of loops in real-world programs do not have termination conditions that involve hundreds of disjuncts. In conclusion, we can tentatively answer *RQ3* in the affirmative by suggesting that

neural termination analysis scales well in the complexity of the program, noting that other parts such as verification and tracing might not scale as well.

6.6.3 Discussion

Owing to the inherent difficulty of the problem at hand (termination), methods for solving it are necessarily incomplete. Neural termination analysis is no exception. Under this constraint, we presented an experimental evaluation to answer three research questions regarding the efficacy of neural ranking functions (*RQ1*), their advantages over other approaches (*RQ2*), and their scalability (*RQ3*). Despite the favourable answers for each of the questions, it is important to point out weaknesses of our approach. For example, it is easy to construct programs where NTA fails but other methods are successful. Usually, these fall into one of the following three cases.

Insufficient Data: Neural termination analysis learns termination arguments from execution traces. Hence, any program feature that limits the data that can be collected is a problem for our approach. For example, several benchmarks in the dataset behave similarly to the loop in Figure 6.12a. This loop has more than one iteration if and only if x is one of $1, 2, 3, 4$ and even then the trace is extremely short. Similar issues can occur when offsets or certain program branches only occur in rare cases. Such instances may lead to overfitting of the networks to the sampled traces. As a result, a learned neural ranking function may satisfy all required properties over the sampled traces but not when verifying it for all possible inputs in the verification procedure. Solver-driven test input generation may be a means to ensure that traces for these behaviours are included in the training data set. Another issue related to insufficient data is the existence of large constants in the problems. For instance, consider the program in Figure 6.12b. This would require complete traces (letting x go all the way to 10000000), which takes a long time to gather. Furthermore, learning a bias of this size can also pose problems.

```
int x = ⋆;
while (x > 0) {
  x = −2⋆x + 10;
}
```

(a) This loop always produces short traces.

```
int x = ⋆;
while (x < 10000000) {
  x++;
}
```

(b) To see the constant 10000000 in the trace we would have to execute the loop for as many iterations.

Figure 6.12: Loops where insufficient data is a problem.

Model Expressivity: As ranking functions become more complex, we need neural architectures that can express them. One instance from the dataset where this problem manifests is a benchmark where the ranking function depends on whether an input variable is even or odd. None of the neural architectures discussed in Sec. 6.4 is expressive enough to capture the concept of "even" and "odd". One way of solving this problem is by considering further neural architectures, which would require a more sophisticated data collection. The key limiting factor when deploying such architectures will likely be the increased complexity of the verification process, rather than the learning.

Verification: When there are multiple correct ranking functions, the verification procedure may not be able to prove all of them correct. This behaviour can be observed in the loop shown in Figure 6.13. When purely looking at the execution

```
int j = i;
while (i < 100) {
  i++;
  j = i;
}
```

Figure 6.13: Trace at loop head suggests two ranking functions but one needs an additional auxiliary invariant.

traces, which is what the learning procedure does, i and j have the exact same values at the loop head. Hence, if the learning process comes up with the ranking function $100 - j$ the verifier would not be able to prove it correct unless it is supplied with the auxiliary invariant that $i = j$ at the loop head. One way to solve

this problem could be to integrate existing methods that discover such invariants. Generally, applying more sophisticated loop invariant generation methods can only improve the effectiveness of our work and is the subject of future investigation. Methods for generating loop invariants include procedures for programs over arrays using theorem provers [235], or constraint based invariants [236–238]. Invariant generation techniques have also been applied to unbounded system verification [237], and path programs [238]. Invariants for Java Programs have been constructed using symbolic execution [239]. Tools for the discovery of invariants from trace data include Daikon [240] and DIG [241]. In Chapter 3 we also discuss invariant generation methods that use machine learning. Note that all of these techniques, much like ours, are incomplete as the problem itself is undecidable.

6.7 Threats to Validity

We used an experimental evaluation to show the effectiveness of Neural Termination Analysis. This makes us vulnerable to multiple threats to validity. It should be noted that we use some of the work discussed in Chapter 5. This mostly includes the benchmark selection and the background on program analysis. Hence, the threats discussed in Section 5.5 also apply here. Furthermore, the validity of an experimental evaluation using software is always threatened by the potential of a faulty implementation. To address this, NTA always provides a ranking function when termination is proven. This in turn can also be verified by hand in case of doubt. We discuss more subtle threats to validity below.

Test input generation: We require that we can generate test input data so that the resulting execution traces can be used to train the neural network. While our simple sampling method is successful on our benchmarks, it may not be possible in general to obtain sufficiently diverse test inputs. We discussed more sophisticated means of test input generation that can be used to mitigate this issue. It should be noted that due to the separation of tracing and verification, "bad" traces do

not impede on the *validity* of existing results. Hence, methods for better trace generation may only *improve* the results.

Neural architecture complexity: While our experiments suggest that tiny neural networks can prove termination of most program loops, there may exist programs that require a large number of neurons. This further amplifies the benchmark bias discussed previously. In particular, by constructing the neural network architectures with the pre-existing benchmark problems in mind, we may have introduced a bias towards networks that work well on the benchmark set but not elsewhere. Hence, different problem sets may require different network architectures. Notably, these might increase training and verification complexity significantly. This potential increase in complexity may also threaten the use of SMT solvers. We use off-the-shelf SMT solvers to check neural ranking functions. Using larger networks may pose limits to the scalability of off-the-shelf SMT solvers.

Benchmark Bias: We used the same benchmarks as in Chapter 5. Hence, the benchmark threats and biases that we discussed in Section 5.5 carry over. While we use standard benchmarks from literature introduced by others to enable a comparison of different termination tools, these benchmarks may not be representative of software written by developers. For a more complete evaluation, ν`Term` would need to be lifted from prototype status and into a mature tool applicable to a wide range of real-world software.

6.8 Related Work

Much like before, we are looking at ranking function synthesis as an instance of function synthesis. Hence, we characterise Neural Termination Analysis in the standard dimensions of synthesis techniques as follows:

Intent: Logical specifications describing states, transitions, and ranking constraints (decreasing and boundedness) for programs that we want to show terminating.

Search Space: Weights and biases for different neural network architectures with ReLU activation functions.

Search Technique: Standard neural network training techniques: Training data gathered from execution traces and gradient descent to find weights of neural networks according to a specific loss function during training.

Neural Termination Analysis draws from several areas at the intersection of machine learning and verification. In addition, we pointed out some related areas of research in Sections 6.5 and 6.3. In contrast, Section 3.2 discusses related work in termination analysis. In the following, we will reiterate some of the already mentioned works in more detail and discuss notable differences to the contributions of this chapter.

As mentioned, execution traces have also been used by `DynamiTe` [153] where the collected data is used to synthesise Ramsey based ranking functions. Instead of machine learning models, `DynamiTe` constructs SMT problems from the traces such that the solutions represent ranking functions. In contrast, [167] uses linear regression and quadratic programming to automatically infer loop bounds from given execution traces. Loop bounds, which are essentially ranking functions, are also synthesised using traces in [172]. In the base case, the authors propose convex optimisation to train an affine loop bound (i.e. ranking function). In case a simple affine function is insufficient, the authors present a method based on clustering to partition the training data, and to learn separate affine ranking functions for each partition. These affine functions are then combined by reducing it to a set covering problem resulting in piecewise affine functions. Finally, the authors also present a method for lexicographic ranking functions where the aforementioned techniques are combined into synthesising a vector of loop bounds. By contrast, the work presented in this chapter employs neural networks whose expressive power subsumes a wide variety of ranking function templates, including piecewise affine functions. Our learning phase only relies on optimising a loss function, and can thus be implemented using generic optimisation algorithms, such as gradient descent, that are readily available in machine learning frameworks.

Piecewise affine lexicographic functions are also the synthesis target in [170]. Instead of traces, the authors use examples obtained from a CEGIS loop to learn ranking functions using decision trees and SVMs. In this method, CEGIS is used to obtain training examples while the decision tree and SVM are used to build the partitions of the piecewise functions. An SMT solver is used to generate a ranking for the examples falling under the specified partitions.

Recently, Support Vector Machines (SVM) have been used to synthesise ranking functions [168]. The same method has been extended to multiphase ranking functions [169]. In both cases, SVMs are used to learn the coefficients of ranking function templates that need to be supplied as parameters to the procedure. These templates can be linear functions, polynomials, etc. The methods are applied to loop programs where the loop guards are restricted to a conjunction of inequalities and the loop body is a set of potentially nondeterministic assignment statements. Most real-world software does not fit into these constraints, including many of our benchmarks. In particular, in Section 6.6 we show how neural termination analysis is able to show programs with *disjunctive* loop guards terminating.

In [171], a deep learning based method for termination analysis is introduced. This method uses neural networks with sigmoidal activation functions, which are shown to be an appropriate ranking function representation for programs defined using continuous functions, without disjunctions and conditional choices. While this is suitable to describe deterministic dynamical systems in discrete time, this language restriction makes the method inapplicable to a lot of software, including the majority of our simple termination analysis benchmarks. We estimate that only 46 out of 110 programs in the `TermComp` and `SV-COMP` problem sets are in the scope of (but not necessarily solved by) their method and remark that our method solves 39 out of these 46 problems. Besides, 5 out of the 14 benchmarks in `NEW` are in their scope, all of which are solved by ν`Term`. Unfortunately, we cannot directly evaluate the effectiveness of their method on either of the three sets because an implementation is unavailable. Moreover, their method cannot be easily implemented in our infrastructure. In fact, neural ranking functions

with sigmoidal activations lack efficient—and complete—decision procedures for checking their validity. Notably, their approach required the development of bespoke decision procedures for this purpose. Conversely, our method uses ReLU activation functions, which can be encoded into expressions in decidable theories, for which efficient SMT solvers are available. Our work goes a step further by showing that neural networks with ReLU activation functions are sufficient to obtain results that are comparable to state-of-the-art tools and even enable the effective termination analysis of programs that are beyond their reach.

The idea of using the weights of a trained neural network to synthesise objects has also been used in [73] to synthesise formulas, and in [74] to synthesise recursive list predicates in a functional language. In both cases, positive, negative, and implication examples are used as training data. Note that in the absence of implication constraints, the procedure essentially falls under the programming-by-example (PBE) paradigm. Hence, the inclusion of implication examples can be thought of as an extension of PBE. In [73], the authors describe a method to synthesise atoms which subsequently can be used to synthesise invariants. Their experiments are set in the domain of Constrained Horn Clauses (CHC). In contrast, the setting of [74] is recursive list predicates in a functional programming language. A recursive functional program is laid out as a template and a special neural network is trained to fill in parts of the template. The authors use special-purpose RNNs that are tailored for this type of recursive predicate synthesis problem. In the experimental evaluation, 11 predicates such as `sorted` are synthesised.

The method described in [124] also utilises the weights of a trained network but applies it to synthesis of SMT formulas from traces. Since they synthesise formulas instead of functions, measures need to be taken so that neural networks can be interpreted as boolean values. To this end, the authors use gated continuous logic networks (G-CLNs) in conjunction with Basic Fuzzy Logic. Hence, their method requires finding continuous equivalents for logical connectives $(\wedge, \vee, \neg, \dots)$ and other predicates such as $<$, $>$, $\leq$, etc. Once trained, the formulas are constructed

by recursing through the G-CLN and extracting clauses with parameters above 0.5. The authors apply this method of constructing SMT formulas to invariant synthesis.

Finally, neural networks are utilised in a similar manner in [177] to prove stability of dynamical systems. A neural network is trained with data from a CEGIS loop to satisfy a form of decreasingness and boundedness that is appropriate for the setting of dynamical systems. Once trained, an SMT solver in the CEGIS loop is invoked to verify the resulting neural network. This work has served as an inspiration for neural termination analysis and was extended to almost-sure termination for probabilistic programs [176].

7 Conclusion

In this thesis, we investigated the application of machine learning techniques to function synthesis. In the following, we summarise our methods and findings. Finally, we will discuss potential future work in Section 7.1.

We first looked into the domain of syntax-guided synthesis and presented an enumerative search strategy for SyGuS as a single player game. To solve this “game of synthesis”, we describe an algorithm based on AlphaZero’s Monte-Carlo tree search. The search agent is guided by machine learned policy and value functions. To balance exploration and exploitation, we use the upper confidence bound for trees. The machine learning models are trained in a reinforcement learning loop from data gathered in previous iterations. We conduct an experimental evaluation on the LIA SyGuS problem set using gradient boosted trees as machine learning models. In order to increase the size of the pre-existing problem set, we also present a method for automatically generating synthesis problems from pre-existing SMT problems using unification and anti-unification. The experiments show that the learned heuristics improve the performance of the search agent. On average, we go from solving 34.3% without learned heuristics to solving 60.7% on the test set. We find that the newly generated SyGuS problems increase the diversity of the training set and lead to improved results on the overall combined testing set. However, when exclusively training on the new problems, we do not see any

significant improvements on the old problems. Compared to cvc5, the experiments show that we can compete with and even outperform cvc5 on some setups. This is most significant on the newly generated problems where we solve 93.2% and cvc5 only solves 77%. In conclusion, we find that machine learned search guidance helps pruning the search space significantly. This is even more impressive when considering the fact that we only used the most basic machine learning models with very simple syntactic features. Furthermore, we do not perform any other pruning of the search space other than the machine learned guidance.

Next, we investigated the application of function synthesis to termination analysis. We implemented a method for generating SyGuS-IF problems that encapsulate the constraints of ranking functions for Java programs. We collect a set of termination problems from pre-existing problem sets. In addition, we add 14 new programs that state-of-the-art tools struggle with. These termination problems contain disjunctive and non-linear loop guards. In the experimental evaluation, we apply cvc5 as well as our machine learning based solver to the resulting synthesis problems. This shows that while SyGuS based ranking function synthesis works, it does not significantly advance the state of the art. Using cvc5 as a backend solver, we solve 68.9% of the problems in all problem sets combined. This places us last in comparison to AProVE (solving 79.2%), Ultimate (solving 75.5%), and DynamiTe (solving 69.8%). A more fine-grained analysis shows that this approach seems to perform poorly on pre-existing problem sets. Interestingly, on our newly generated set, we solve as many problems as DynamiTe clearly beating the other competitors. We believe that this is due to the fact that the background solvers we use are based on term enumeration. Hence, even in complex programs involving non-linear and disjunctive loop guards, cvc5 is able to quickly produce expressions of small depths. Further, we are even able to produce ranking functions that require branching with if-then-else constructs. Despite these advantages, however, we believe that the drawbacks such as lack of domain specific knowledge cannot be compensated for.

Finally, in Chapter 6, we seek to improve on the termination analysis results. To this end, we present a framework for synthesising ranking functions called Neural Termination Analysis (NTA). In this framework, we use the input programs to generate execution traces. These traces constitute the training data for neural networks that we train to act as ranking functions. The properties of a ranking function (i.e. decreasingness and boundedness) are enforced through the design of the neural network and the loss function. After training, the resulting neural ranking function is translated to a symbolic expression representing the ranking function. Subsequently, a standard SMT solver is used to verify that the resulting function indeed is a correct ranking function for the input program. Neural Termination Analysis has the advantage, that the complexity of the synthesis procedure is entirely delegated to the machine learning algorithm (e.g. gradient descent). Meanwhile, the SMT solver only has the task of checking the validity of a given ranking function, instead of synthesising a ranking function ground up. As such, the technique can adequately be described as "using gradient descent as a ranking function synthesis algorithm". To evaluate the efficacy of NTA, we provide a prototype implementation called ν`Term` and test it on the aforementioned collection of benchmarks. When using tiny neural networks with one hidden layer and a straightforward training script, we solve 76.6% of problems, performing comparably to state-of-the-art tools. Furthermore, we show that ν`Term` can solve 94.3% of the problems in our newly created benchmark set, which have proven to be challenging for competing tools. All of these results were obtained using tiny neural networks, indicating the scalability of this method.

7.1 Future Work

We have already hinted at some possibilities for future work in the previous sections. In the following, we point out additional future work in machine learning for function synthesis.

7.1.1 Reinforcement Learning for Synthesis

We emphasised that the search procedure we implemented exclusively worked with machine learned search guidance. State-of-the-art tools use optimised algorithms and data structures to efficiently enumerate and construct terms. Further, enumerative synthesis tools use many different techniques to prune the search space by, for example, discarding branches that cannot lead to solutions. We believe that combining the machine learned guidance we provide with a mature tool that implements the aforementioned optimisations would improve the performance significantly.

Models and Features: We use machine learned guidance with gradient boosted tree models and simple syntactic features describing the search state. We believe that investigating more diverse features that are not based on simple term walks could improve the results. This can be further improved by considering other machine learning models. Notably, when considering neural networks, the process of feature selection may become obsolete. In fact, in AlphaZero which is the basis for the Monte-Carlo based search algorithm, deep neural networks are used and hence, no handcrafted features were provided. In this context, one might have to consider the use of tree neural networks or other designs that are applicable to logical formulas and expressions.

Function Generation Procedure: In our setup, we generate functions by searching a tree generated by a given grammar. This has the advantage that each function is in the language defined by the syntactic constraint and that every expression we generate is well-formed. It is worth investigating and comparing MCTS to other search procedures. For example, it would be interesting to use the learned policy and value to guide an A^* style search. A different approach would be to disregard the grammar tree entirely and use a machine learning model to generate the functions directly. For example, one could investigate the use of encoder-decode networks to encode specifications and generate solutions. Multiple networks that

take the tree structure of terms and formulas into account have been developed. Some have even been applied to program synthesis in the context of code translation [242].

Problem Generation: We are certain that generating hard but solvable problems will not only improve the performance of our methods but that of other tools in active development as well. In our setup, we used very simple heuristics based on the type and length of the LGG. Leveraging other properties or calling solvers during the generation to produce adversarial examples might lead to new and challenging problem sets. In addition, we did not change the syntactic specifications in our generated problems. The problem sets could be diversified by mutating the syntactic specifications and only allowing for syntactically restricted solutions. A fundamentally different approach to generating new problems would be the use of generative AI models. For example, one could use generative adversarial networks to iteratively solve and generate new problems. This would be particularly interesting as it could be incorporated into the reinforcement learning setup that we presented.

7.1.2 Neural Termination Analysis

As stated in Section 6.7, an implementation of neural termination analysis within a mature termination prover would make the techniques applicable to a wider range of programs and real world software.

Tracing Data Generation: In our experimental evaluation, we used a simple sampling method to generate execution traces. As mentioned in Section 6.3 automated testing and fuzzing are active areas of research. Incorporating the techniques used in either of these areas would improve the quality of the training data. We suspect that as the programs to analyse become more complex, the need for better sampling strategies becomes more apparent. For example, if the control flow graph of the program to analyse contains many branches, a certain coverage by the traces might be required. We believe that it is very likely that applying more sophisticated techniques to tracing data generation can only improve our results. However, we leave it to future work to investigate this.

Verification: In νTerm, we take the trained neural network f_{nn} and generate a symbolic function f with integer coefficients by recursing into the network and rounding float values to the nearest integer. This could lead to a situation where f_{nn} is a correct ranking for all pairs in the training set D (i.e. has a loss of zero) while the expression f is not a correct ranking for D. As mentioned in Section 6.5, discretisation of neural networks is an active area of research. We believe that methods developed in this area can be applied here to decrease the error that simple rounding could incur.

Other Areas of Application: One feature of Neural Termination Analysis is that we enforced the formal specifications of ranking functions using the loss function as well as the neural network design. In particular, we used ReLU activation functions with a margin ranking loss function. In future work, applications of similar ideas to other problems of program verification should be investigated. For example, one could investigate if it is possible to design a network f and corresponding loss function such that $f(\vec{x}) = \vec{x'}$ describes the transition of a state x before a loop and x' after the same loop. Such a network f could be described as "loop summary network". Similar methods could also be used to discover invariants. It remains open if other synthesis specifications can be translated to corresponding machine learning primitives, such as loss functions and network designs.

Learning Certificates: Neural Termination Analysis falls into a wider range of "guess-and-check" methods that use inexact methods to guess a solution or certificate followed by exact methods to verify the guess. Usually, the inexact methods are probabilistic in nature and, usually involve some form of machine learning. In contrast, verification is usually done using formal methods. In our case, the neural network plays the role of a certificate or proof candidate for termination, while an SMT solver is used to verify the candidate. This methodology is conducive to *existential nth-order problems*, (i.e. synthesis problems), since the guessing mechanism can search for witnesses while the checker "only" has to find proofs in a logic of order $n-1$. We believe that guess-and-check is a very powerful methodology

that combines the power of machine learning with the rigour and guarantees of formal methods. Future work should explore the interplay of this dichotomy.

Appendices

A
Example: Generated SyGuS Problem from SMT

Figure A.1 shows an SMT-LIB problem from the `UltimateAutomizer` directory of the SMT-LIB benchmark database. For readability, we removed comments detailing the origin of the problem. Using the algorithm presented in Section 4.8.1 we obtain the `SyGuS-IF` problem shown in Figure A.2.

```
(set-logic LIA)
(declare-fun |c_mult_#in~n| () Int)
(declare-fun |c_mult_#res| () Int)
(declare-fun c_mult_~n () Int)
(assert (exists ((v_nnf_14 Int))
  (and (<= (+ (* 2 v_nnf_14) (* 2 c_mult_~n) 1)
       |c_mult_#res|)
    (<= |c_mult_#in~n| c_mult_~n)
    (<= c_mult_~n v_nnf_14))))
(assert (not (and
    (<= |c_mult_#in~n| c_mult_~n)
    (exists ((mult_~n Int) (v_nnf_15 Int))
    (and (<= (+ (* 2 mult_~n) (* 2 v_nnf_15) 1)
         |c_mult_#res|)
      (<= |c_mult_#in~n| mult_~n)
      (<= mult_~n v_nnf_15))))))
(check-sat)
```

Figure A.1: Problem `Primes_true-unreach-call.c_276.smt2` from the LIA problem set.

```
(set-logic LIA)

(declare-var c_mult_hashSymbolin~n Int)
(declare-var c_mult_hashSymbolres Int)
(declare-var c_mult_~n Int)

(synth-fun synthTarget((x Int)(y Int)) Int
  ((NTInt Int) (NTbool Bool))
  ((NTInt Int
    (x y 0 1 (- NTInt) (+ NTInt NTInt)
    (- NTInt NTInt) (ite NTbool NTInt NTInt)))
  (NTbool Bool
    ((not NTbool) (and NTbool NTbool) (or NTbool NTbool)
    (ite NTbool NTbool NTbool) (= NTInt NTInt)
    (< NTInt NTInt) (> NTInt NTInt) )))
)

(constraint (not (and
  (exists ((v_nnf_14 Int))
    (and
      (<= (synthTarget v_nnf_14 c_mult_~n)
         c_mult_hashSymbolres)
      (<= c_mult_hashSymbolin~n c_mult_~n)
      (<= c_mult_~n v_nnf_14)))
  (not
    (and
      (<= c_mult_hashSymbolin~n c_mult_~n)
      (exists ((mult_~n Int) (v_nnf_15 Int))
        (and
          (<= (synthTarget mult_~n v_nnf_15 )
             c_mult_hashSymbolres)
          (<= c_mult_hashSymbolin~n mult_~n)
          (<= mult_~n v_nnf_15))))))))
(check-synth)
```

Figure A.2: Resulting SyGuS problem generated from SMT problem in Figure A.1.

B
New Termination Problems

In Figures B.1 to B.6 we present 6 of the 14 problems in the `NEW` problem set. We created these problems to showcase some weaknesses of state-of-the-art tools. These usually feature disjunctive loop guards and linear loop bodies with non-linear loop guards. The remaining 8 problems are variations of the same theme.

```
int n = *;
int a = 0;
while (a * a * 4 <= n ) {
    a = a + 1;
}
```

Figure B.1: Problem named DynamiteExampleX4. This is one of the simplest examples which is a slight variation from the running example presented in [153].

```
int m = *;
int n = *;
int o = *;
int a = 0;
while (a*a* 4 <= n or a*a* 4 <= m or a*a* 4 <= o) {
    a = a + 1;
}
```

Figure B.2: Problem named DynExUpTo3VarsDisj with 3 disjuncts and non-linear guard. Non-linearity comes from squaring a single variable, `a`.

```
int m = *;
int n = *;
int o = *;
int p = *;
int a = 0;
while (a*a*4 <= n or a*a* 4 <= m or  a*a* 4 <= o or  a*a*
   4 <= p) {
    a = a + 1;
}
```

Figure B.3: Problem named DynExUpTo4VarsDisj. Same as Figure B.2 but with 4 disjuncts.

```
int m = *;
int a = 0;
while (a * a <= m) {
    a = a + 1;
    m = m - 1;
}
```

Figure B.4: Problem named PolyClosingIn. The loop guard has a non-linear term. Furthermore, the variable `m` decreases while `a` increases.

```
int n = *;
int a = 0;
int b = 0;
while (a * a <= n or b <= n) {
    a = a + 1;
    b = b + 1;
}
```

Figure B.5: Problem named SquareDisj2Vars. We have two disjuncts, one of them is non-linear as it contains a squared term. The other disjunct is linear.

```
int m = *;
int n = *;
int a = 0;
int b = 0;
while ((a + 1) * (a + 1) <= n or (b + 1) * (b + 1) <= m) {
    a = a + 1;
    b = b + 1;
}
```

Figure B.6: Problem named TwoPol4VarsDisj. The loop guard is a disjunction of two conditions containing polynomial expressions.

C
Generating Execution Traces

C.1 Input Sampling

Empirically, we found that loop guards in many cases perform binary comparison operations. Therefore, instead of simply sampling random inputs from a uniform distribution, we sample from a multivariate normal distribution where one pair of inputs is covariant. We call this pairwise anticorrelated sampling (PAS).

Definition C.1.1 (Pairwise Anticorrelated Sampling)**.** Let K_T and K_L, be two constants and let a and b be random integers with $0 \leq a, b \leq n-1$ from a uniform distribution. A pairwise anticorrelated sample is a random vector $[x_0, x_1, \ldots, x_{n-1}] \in \mathbb{R}^n$ sampled from a multivariate normal distribution with mean 0 and covariance matrix $A \in \mathbb{R}^n \times \mathbb{R}^n$ such that

$$A(i,j) = \begin{cases} K_T & \text{if } i = j \\ K_L & \text{if } i, j \in \{a, b\} \text{ with } i \neq j \\ 0 & \text{otherwise.} \end{cases}$$

Intuitively, PAS samples a vector with a covariance of K_L for a randomly chosen pair and a variance K_T for each remaining element.

C.2 Tracing

We generate execution traces dynamically, by running the program. The process of tracing takes two inputs: the program that is to be traced and a list of program locations in the program where a snapshot of the state is to be taken. In our case, these locations are the loop heads in the program. Our approach is conceptually simple and independent of the platform and programming language. As we consider Java, we give implementation details specific to the Java environment and the Java Virtual Machine (JVM), but our approach could also be applied to more abstract models of computation. The following steps describe how we generate execution traces. We repeat the steps 1000 times to generate as many execution traces.

Input sampling Termination analysis is commonly applied to program fragments that contain some initialisation and a (possibly nested) loop. Therefore, we work with programs that are not closed but require inputs. Furthermore, since we only consider deterministic programs, two traces that are generated with the same sequence of inputs are identical. We use Java's reflection mechanism to obtain a list of arguments that the function we want to analyse accepts. Given this list, we generated random argument tuples using the sampling described in Section C.1.

Execution We start executing the program with the sampled arguments. We maintain control over the JVM during the execution using the Java Virtual Machine Tool Interface (JVMTI). Once we hit a loop head location, we halt the execution and take a snapshot of the memory.

Snapshot Using the JVMTI, we have access to the Local Variable Table (LVT). The LVT contains all local variables of the given function. We create a memory snapshot by iterating through the LVT and reading the values of every variable that is in scope at the given location. For variables that are out of scope, we record a placeholder default value (which depends on the type of the variable). Since the number of local variables does not change, the size of the snapshots is always the

same. Hence, we do not have to worry about padding etc. Once the snapshot is collected, we append it to the trace of the current program. If the maximum length of a trace is reached we force a termination of the virtual machine, otherwise we resume the execution. In our experiments, we use a maximum trace length of 1000. The resulting list of snapshots constitutes an execution trace.

References

[1] Julian Parsert and Elizabeth Polgreen. "Reinforcement Learning and Data-Generation for Syntax-Guided Synthesis". In: *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*. Ed. by Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan. AAAI Press, 2024, pp. 10670–10678. URL: https://doi.org/10.1609/aaai.v38i9.28938.

[2] Mirco Giacobbe, Daniel Kroening, and Julian Parsert. "Neural termination analysis". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. Ed. by Abhik Roychoudhury, Cristian Cadar, and Miryung Kim. ACM, 2022, pp. 633–645. URL: https://doi.org/10.1145/3540250.3549120.

[3] Alonzo Church. "Logic, Arithmetic, and Automata". In: *Journal of Symbolic Logic* 29.4 (1964), pp. 210–210.

[4] Joyce Friedman. "Alonzo Church. Application of recursive arithmetic to the problem of circuit synthesis. Summaries of talks presented at the Summer Institute for Symbolic Logic Cornell University, 1957, 2nd edn., Communications Research Division, Institute for Defense Analyses, Princeton, N. J., 1960, pp. 3–50. 3a-45a." In: *The Journal of Symbolic Logic* 28.4 (1963), pp. 289–290.

[5] José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. "FlashFill++: Scaling Programming by Example by Cutting to the Chase". In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 952–981. URL: https://doi.org/10.1145/3571226.

[6] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. "Program Synthesis with Large Language Models". In: *CoRR* abs/2108.07732 (2021). arXiv: 2108.07732. URL: https://arxiv.org/abs/2108.07732.

[7] *GitHub Copilot*. Accessed: 15 Nov 2023. URL: https://github.com/features/copilot/.

[8] Cristina David, Pascal Kesseli, Daniel Kroening, and Matt Lewis. "Program Synthesis for Program Analysis". In: *ACM Trans. Program. Lang. Syst.* 40.2 (2018), 5:1–5:45. URL: https://doi.org/10.1145/3174802.

[9] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. "Synthesizing software verifiers from proof rules". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. Ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, 2012, pp. 405–416. URL: https://doi.org/10.1145/2254064.2254112.

[10] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. "Program analysis as constraint solving". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. Ed. by Rajiv Gupta and Saman P. Amarasinghe. ACM, 2008, pp. 281–292. URL: https://doi.org/10.1145/1375581.1375616.

[11] E.M. Clarke and R.P. Kurshan. "Computer-aided verification". In: *IEEE Spectrum* 33.6 (1996), pp. 61–67.

[12] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. URL: https://doi.org/10.1145/363235.363259.

[13] Stephen A. Cook. "Soundness and Completeness of an Axiom System for Program Verification". In: *SIAM J. Comput.* 7.1 (1978), pp. 70–90. URL: https://doi.org/10.1137/0207005.

[14] Alan Mathison Turing et al. "On computable numbers, with an application to the Entscheidungsproblem". In: *J. of Math* 58.345-363 (1936), p. 5.

[15] Xiuhan Shi, Xiaofei Xie, Yi Li, Yao Zhang, Sen Chen, and Xiaohong Li. "Large-scale analysis of non-termination bugs in real-world OSS projects". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. Ed. by Abhik Roychoudhury, Cristian Cadar, and Miryung Kim. ACM, 2022, pp. 256–268. URL: https://doi.org/10.1145/3540250.3549129.

[16] *OpenSSL Security Advisory [15 March 2022]*. 2022. URL: https://www.openssl.org/news/secadv/20220315.txt (visited on 03/15/2022).

[17] Robert W. Floyd. "Assigning Meanings to Programs". In: *Proceedings of Symposium in Applied llathematics*. Vol. 19. 1967, pp. 19–32.

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

[19] Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli, and Clark W. Barrett. "Extending SMT Solvers to Higher-Order Logic". In: *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*. Ed. by Pascal Fontaine. Vol. 11716. Lecture Notes in Computer Science. Springer, 2019, pp. 35–54. URL: https://doi.org/10.1007/978-3-030-29436-6%5C_3.

[20] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. "Counterexample Guided Inductive Synthesis Modulo Theories". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 270–288. URL: https://doi.org/10.1007/978-3-319-96145-3%5C_15.

[21] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. "cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis". In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11562. Lecture Notes in Computer Science. Springer, 2019, pp. 74–83. URL: https://doi.org/10.1007/978-3-030-25543-5%5C_5.

[22] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144. eprint: https://www.science.org/doi/pdf/10.1126/science.aar6404. URL: https://www.science.org/doi/abs/10.1126/science.aar6404.

[23] Rajeev Alur, Dana Fisman, Rishabh Singh, and Abhishek Udupa. *Syntax Guided Synthesis Competition*. https://sygus.org/. 2017.

[24] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

[25] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016. URL: https://doi.org/10.1007/978-3-662-50497-0.

[26] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2017.

[27] Dexter C. Kozen. *Automata and Computability*. 1st. Berlin, Heidelberg: Springer-Verlag, 1997.

[28] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. "Syntax-Guided Synthesis". In: *Dependable Software Systems Engineering*. Vol. 40. NATO Science for Peace and Security Series, D: Information and Communication Security. IOS Press, 2015, pp. 1–25.

[29] Gordon D. Plotkin. "A structural approach to operational semantics". In: *J. Log. Algebraic Methods Program.* 60-61 (2004), pp. 17–139.

[30] Frances E. Allen. “Control Flow Analysis”. In: *SIGPLAN Not.* 5.7 (July 1970). Ed. by Robert S. Northcote, pp. 1–19. URL: https://doi.org/10.1145/390013.808479.

[31] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490. URL: https://doi.org/10.1145/115372.115320.

[32] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999. URL: https://doi.org/10.1007/978-3-662-03811-6.

[33] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification, 2nd ed.* Wiley-Teubner, 1987.

[34] John C. Reynolds. *Theories of programming languages*. Cambridge University Press, 1998.

[35] E. Harzheim. *Ordered Sets*. Advances in Mathematics - Kluwer Academic Publishers. Springer, 2005. URL: https://books.google.at/books?id=z_aoL8CiPfgC.

[36] John Harrison. “Inductive Definitions: Automation and Application”. In: *Higher Order Logic Theorem Proving and Its Applications, 8th International Workshop, Aspen Grove, UT, USA, September 11-14, 1995, Proceedings*. Ed. by E. Thomas Schubert, Phillip J. Windley, and Jim Alves-Foss. Vol. 971. Lecture Notes in Computer Science. Springer, 1995, pp. 200–213. URL: https://doi.org/10.1007/3-540-60275-5%5C_66.

[37] Piotr Rudnicki and Andrzej Trybulec. “On Equivalents of Well-Foundedness”. In: *J. Autom. Reason.* 23.3-4 (1999), pp. 197–234. URL: https://doi.org/10.1023/A:1006218513245.

[38] Alonzo Church. “Application of Recursive Arithmetic to the Problem of Circuit Synthesis”. In: *Journal of Symbolic Logic* 28.4 (1963), pp. 289–290.

[39] Richard J. Waldinger and Richard C. T. Lee. “PROW: A Step Toward Automatic Program Writing”. In: *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, USA, May 7-9, 1969*. Ed. by Donald E. Walker and Lewis M. Norton. William Kaufmann, 1969, pp. 241–252. URL: http://ijcai.org/Proceedings/69/Papers/024.pdf.

[40] C. Cordell Green. “Application of Theorem Proving to Problem Solving”. In: *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, USA, May 7-9, 1969*. Ed. by Donald E. Walker and Lewis M. Norton. William Kaufmann, 1969, pp. 219–240. URL: http://ijcai.org/Proceedings/69/Papers/023.pdf.

[41] Zohar Manna and Richard J. Waldinger. “A Deductive Approach to Program Synthesis”. In: *ACM Trans. Program. Lang. Syst.* 2.1 (1980), pp. 90–121. URL: https://doi.org/10.1145/357084.357090.

[42] Zohar Manna and Richard J. Waldinger. “Towards automatic program synthesis”. In: *Symposium on Semantics of Algorithmic Languages*. Ed. by Erwin Engeler. Vol. 188. Lecture Notes in Mathematics. Springer, 1971, pp. 270–310. URL: https://doi.org/10.1007/BFb0059702.

[43] Petra Hozzová, Laura Kovács, Chase Norman, and Andrei Voronkov. “Program Synthesis in Saturation”. In: *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*. Ed. by Brigitte Pientka and Cesare Tinelli. Vol. 14132. Lecture Notes in Computer Science. Springer, 2023, pp. 307–324. URL: https://doi.org/10.1007/978-3-031-38499-8%5C_18.

[44] David A. Basin, Yves Deville, Pierre Flener, Andreas Hamfelt, and Jørgen Fischer Nilsson. “Synthesis of Programs in Computational Logic”. In: *Program Development in Computational Logic: A Decade of Research Advances in Logic-Based Program Development*. Ed. by Maurice Bruynooghe and Kung-Kiu Lau. Vol. 3049. Lecture Notes in Computer Science. Springer, 2004, pp. 30–65. URL: https://doi.org/10.1007/978-3-540-25951-0%5C_2.

[45] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. “Program Synthesis”. In: *Found. Trends Program. Lang.* 4.1-2 (2017), pp. 1–119.

[46] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. “Component-based synthesis of table consolidation and transformation tasks from examples”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 422–436. URL: https://doi.org/10.1145/3062341.3062351.

[47] Sumit Gulwani. “Automating string processing in spreadsheets using input-output examples”. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 317–330. URL: https://doi.org/10.1145/1926385.1926423.

[48] Rishabh Singh and Sumit Gulwani. “Predicting a Correct Program in Programming by Example”. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 398–414. URL: https://doi.org/10.1007/978-3-319-21690-4%5C_23.

[49] Oleksandr Polozov and Sumit Gulwani. “FlashMeta: a framework for inductive program synthesis”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. Ed. by Jonathan Aldrich and Patrick Eugster. ACM, 2015, pp. 107–126. URL: https://doi.org/10.1145/2814270.2814310.

[50] John K. Feser, Swarat Chaudhuri, and Isil Dillig. “Synthesizing data structure transformations from input-output examples”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Stephen M. Blackburn. ACM, 2015, pp. 229–239. URL: https://doi.org/10.1145/2737924.2737977.

[51] Xinyu Wang, Isil Dillig, and Rishabh Singh. “Program synthesis using abstraction refinement”. In: *Proc. ACM Program. Lang.* 2.POPL (2018), 63:1–63:30. URL: https://doi.org/10.1145/3158151.

[52] Illia Polosukhin and Alexander Skidanov. "Neural Program Search: Solving Programming Tasks from Description and Examples". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=BJTtWDyPM.

[53] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. "RobustFill: Neural Program Learning under Noisy I/O". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 990–998. URL: http://proceedings.mlr.press/v70/devlin17a.html.

[54] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. "AutoPandas: neural-backed generators for program synthesis". In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 168:1–168:27. URL: https://doi.org/10.1145/3360594.

[55] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. "Neuro-Symbolic Program Synthesis". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=rJ0JwFcex.

[56] Kairo Morton, William T. Hallahan, Elven Shum, Ruzica Piskac, and Mark Santolucito. "Grammar Filtering for Syntax-Guided Synthesis". In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 1611–1618. URL: https://doi.org/10.1609/aaai.v34i02.5522.

[57] Armando Solar-Lezama. "Program sketching". In: *Int. J. Softw. Tools Technol. Transf.* 15.5-6 (2013), pp. 475–495. URL: https://doi.org/10.1007/s10009-012-0249-7.

[58] Armando Solar-Lezama. "The Sketching Approach to Program Synthesis". In: *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*. Ed. by Zhenjiang Hu. Vol. 5904. Lecture Notes in Computer Science. Springer, 2009, pp. 4–13. URL: https://doi.org/10.1007/978-3-642-10672-9%5C_3.

[59] Maxwell I. Nye, Luke B. Hewitt, Joshua B. Tenenbaum, and Armando Solar-Lezama. "Learning to Infer Program Sketches". In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 4861–4870. URL: http://proceedings.mlr.press/v97/nye19a.html.

[60] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. “A Machine Learning Framework for Programming by Example”. In: *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*. Vol. 28. JMLR Workshop and Conference Proceedings. JMLR.org, 2013, pp. 187–195. URL: http://proceedings.mlr.press/v28/menon13.html.

[61] Percy Liang, Michael I. Jordan, and Dan Klein. “Learning Programs: A Hierarchical Bayesian Approach”. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*. Ed. by Johannes Fürnkranz and Thorsten Joachims. Omnipress, 2010, pp. 639–646. URL: https://icml.cc/Conferences/2010/papers/568.pdf.

[62] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. “DeepCoder: Learning to Write Programs”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=ByldLrqlx.

[63] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. “Program Synthesis Using Deduction-Guided Reinforcement Learning”. In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 587–610. URL: https://doi.org/10.1007/978-3-030-53291-8%5C_30.

[64] Kevin Ellis, Maxwell I. Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. “Write, Execute, Assess: Program Synthesis with a REPL”. In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d’Alché-Buc, Emily B. Fox, and Roman Garnett. 2019, pp. 9165–9174. URL: https://proceedings.neurips.cc/paper/2019/hash/50d2d2262762648589b1943078712aa6-Abstract.html.

[65] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. “Learning Libraries of Subroutines for Neurally–Guided Bayesian Program Induction”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper_files/paper/2018/file/7aa685b3b1dc1d6780bf36f7340078c9-Paper.pdf.

[66] Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Lucas Morales, Luke B. Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. “DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning”. In: *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and

Eran Yahav. ACM, 2021, pp. 835–850. URL: https://doi.org/10.1145/3453483.3454080.

[67] Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. "Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=H1Xw62kRZ.

[68] Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. "BUSTLE: Bottom-Up Program Synthesis Through Learning-Guided Exploration". In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL: https://openreview.net/forum?id=yHeg4PbFHh.

[69] Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. "CrossBeam: Learning to Search in Bottom-Up Program Synthesis". In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL: https://openreview.net/forum?id=qhC8mr2LEKq.

[70] Xinyun Chen, Chang Liu, and Dawn Song. "Execution-Guided Neural Program Synthesis". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=H1gfOiAqYm.

[71] Kensen Shi, Hanjun Dai, Wen-Ding Li, Kevin Ellis, and Charles Sutton. "LambdaBeam: Neural Program Search with Higher-Order Functions and Lambdas". In: *CoRR* abs/2306.02049 (2023). arXiv: 2306.02049. URL: https://doi.org/10.48550/arXiv.2306.02049.

[72] Giovanni De Toni, Luca Erculiani, and Andrea Passerini. "Learning compositional programs with arguments and sampling". In: *CoRR* abs/2109.00619 (2021). arXiv: 2109.00619. URL: https://arxiv.org/abs/2109.00619.

[73] Naoki Kobayashi, Taro Sekiyama, Issei Sato, and Hiroshi Unno. "Toward Neural-Network-Guided Program Synthesis and Verification". In: *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings*. Ed. by Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi. Vol. 12913. Lecture Notes in Computer Science. Springer, 2021, pp. 236–260. URL: https://doi.org/10.1007/978-3-030-88806-0%5C_12.

[74] Naoki Kobayashi and Minchao Wu. "Neural Network-Guided Synthesis of Recursive List Functions". In: *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Vol. 13993. Lecture Notes in Computer Science. Springer, 2023, pp. 227–245. URL: https://doi.org/10.1007/978-3-031-30823-9%5C_12.

[75] Sorav Bansal and Alex Aiken. "Binary Translation Using Peephole Superoptimizers". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 177–192. URL: http://www.usenix.org/events/osdi08/tech/full%5C_papers/bansal/bansal.pdf.

[76] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodík, and Dinakar Dhurjati. "Scaling up Superoptimization". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*. Ed. by Tom Conte and Yuanyuan Zhou. ACM, 2016, pp. 297–310. URL: https://doi.org/10.1145/2872362.2872387.

[77] Ameer Haj-Ali. "Machine Learning in Compiler Optimization". PhD thesis. University of California, Berkeley, USA, 2020. URL: https://www.escholarship.org/uc/item/9431v2tg.

[78] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. "MLGO: a Machine Learning Guided Compiler Optimizations Framework". In: *CoRR* abs/2101.04808 (2021). arXiv: 2101.04808. URL: https://arxiv.org/abs/2101.04808.

[79] Mark Stephenson, Saman P. Amarasinghe, Martin C. Martin, and Una-May O'Reilly. "Meta optimization: improving compiler heuristics with machine learning". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*. Ed. by Ron Cytron and Rajiv Gupta. ACM, 2003, pp. 77–90. URL: https://doi.org/10.1145/781131.781141.

[80] Zheng Wang and Michael F. P. O'Boyle. "Machine Learning in Compiler Optimization". In: *Proc. IEEE* 106.11 (2018), pp. 1879–1901. URL: https://doi.org/10.1109/JPROC.2018.2817118.

[81] Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Köppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, Taylan Cemgil, Mohammadamin Barekatain, Yujia Li, Amol Mandhane, Thomas Hubert, Julian Schrittwieser, Demis Hassabis, Pushmeet Kohli, Martin Riedmiller, Oriol Vinyals, and David Silver. "Faster sorting algorithms discovered using deep reinforcement learning". In: *Nature* 618.7964 (June 2023), pp. 257–263. URL: https://doi.org/10.1038/s41586-023-06004-9.

[82] Ameer Haj-Ali, Nesreen K. Ahmed, Theodore L. Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. "NeuroVectorizer: end-to-end vectorization with deep reinforcement learning". In: *CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020*. ACM, 2020, pp. 242–255. URL: https://doi.org/10.1145/3368826.3377928.

[83] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. "Verified lifting of stencil computations". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by Chandra Krintz and Emery D. Berger. ACM, 2016, pp. 711–726. URL: https://doi.org/10.1145/2908080.2908117.

[84] José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael F. P. O'Boyle. "C2TACO: Lifting Tensor Code to TACO". In: *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2023, Cascais, Portugal, October 22-23, 2023*. Ed. by Coen De Roover, Bernhard Rumpe, and Amir Shaikhha. ACM, 2023, pp. 42–56. URL: https://doi.org/10.1145/3624007.3624053.

[85] Susmit Jha and Sanjit A. Seshia. "A theory of formal synthesis via inductive learning". In: *Acta Informatica* 54.7 (2017), pp. 693–726. URL: https://doi.org/10.1007/s00236-017-0294-5.

[86] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. "Program synthesis using conflict-driven learning". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by Jeffrey S. Foster and Dan Grossman. ACM, 2018, pp. 420–435. URL: https://doi.org/10.1145/3192366.3192382.

[87] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. "Scaling Enumerative Program Synthesis via Divide and Conquer". In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 319–336. URL: https://doi.org/10.1007/978-3-662-54577-5%5C_18.

[88] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. "Accelerating search-based program synthesis using learned probabilistic models". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by Jeffrey S. Foster and Dan Grossman. ACM, 2018, pp. 436–449. URL: https://doi.org/10.1145/3192366.3192410.

[89] Xujie Si, Yuan Yang, Hanjun Dai, Mayur Naik, and Le Song. "Learning a Meta-Solver for Syntax-Guided Program Synthesis". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=Syl8Sn0cK7.

[90] Tristan Cazenave. "Monte-Carlo Expression Discovery". In: *Int. J. Artif. Intell. Tools* 22.1 (2013). URL: https://doi.org/10.1142/S0218213012500352.

[91] Thibault Gauthier. "Deep Reinforcement Learning for Synthesizing Functions in Higher-Order Logic". In: *LPAR*. Vol. 73. EPiC Series in Computing. EasyChair, 2020, pp. 230–248.

[92] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. “Program synthesis using natural language”. In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. Ed. by Laura K. Dillon, Willem Visser, and Laurie A. Williams. ACM, 2016, pp. 345–356. URL: https://doi.org/10.1145/2884781.2884786.

[93] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. “Multi-modal program inference: a marriage of pre-trained language models and component-based synthesis”. In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–29. URL: https://doi.org/10.1145/3485535.

[94] Anirudh Khatry, Joyce Cahoon, Jordan Henkel, Shaleen Deep, K. Venkatesh Emani, Avrilia Floratou, Sumit Gulwani, Vu Le, Mohammad Raza, Sherry Shi, Mukul Singh, and Ashish Tiwari. “From Words to Code: Harnessing Data for Program Synthesis from Natural Language”. In: *CoRR* abs/2305.01598 (2023).

[95] Kensen Shi, David Bieber, and Rishabh Singh. “TF-Coder: Program Synthesis for Tensor Manipulations”. In: *ACM Trans. Program. Lang. Syst.* 44.2 (2022), 10:1–10:36. URL: https://doi.org/10.1145/3517034.

[96] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages”. In: *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*. Ed. by Trevor Cohn, Yulan He, and Yang Liu. Vol. EMNLP 2020. Findings of ACL. Association for Computational Linguistics, 2020, pp. 1536–1547. URL: https://doi.org/10.18653/v1/2020.findings-emnlp.139.

[97] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation”. In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Ed. by Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih. Association for Computational Linguistics, 2021, pp. 8696–8708. URL: https://doi.org/10.18653/v1/2021.emnlp-main.685.

[98] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. “CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis”. In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL: https://openreview.net/pdf?id=iaYcJKpY2B%5C_.

[99] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such,

Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. "Evaluating Large Language Models Trained on Code". In: *CoRR* abs/2107.03374 (2021). arXiv: 2107.03374. URL: https://arxiv.org/abs/2107.03374.

[100] Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Oleksandr Polozov, and Charles Sutton. "Natural Language to Code Generation in Interactive Data Science Notebooks". In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*. Ed. by Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki. Association for Computational Linguistics, 2023, pp. 126–173. URL: https://doi.org/10.18653/v1/2023.acl-long.9.

[101] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. "Synchromesh: Reliable Code Generation from Pre-trained Language Models". In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL: https://openreview.net/forum?id=KmtVD97J43e.

[102] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. "Learning to Represent Programs with Graphs". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=BJOFETxR-.

[103] Rohan Bavishi, Harshit Joshi, José Cambronero, Anna Fariha, Sumit Gulwani, Vu Le, Ivan Radicek, and Ashish Tiwari. "Neurosymbolic repair for low-code formula languages". In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (2022), pp. 1093–1122. URL: https://doi.org/10.1145/3563327.

[104] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. "Competition-level code generation with AlphaCode". In: *Science* 378.6624 (2022), pp. 1092–1097. eprint: https://www.science.org/doi/pdf/10.1126/science.abq1158. URL: https://www.science.org/doi/abs/10.1126/science.abq1158.

[105] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong,

Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation". In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual.* Ed. by Joaquin Vanschoren and Sai-Kit Yeung. 2021. URL: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html.

[106] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. "A systematic evaluation of large language models of code". In: *MAPS@PLDI 2022: 6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA, 13 June 2022.* Ed. by Swarat Chaudhuri and Charles Sutton. ACM, 2022, pp. 1–10. URL: https://doi.org/10.1145/3520312.3534862.

[107] Zishun Yu, Yunzhe Tao, Liyu Chen, Tao Sun, and Hongxia Yang. "B-Coder: Value-Based Deep Reinforcement Learning for Program Synthesis". In: *CoRR* abs/2310.03173 (2023). arXiv: 2310.03173. URL: https://doi.org/10.48550/arXiv.2310.03173.

[108] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. "CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning". In: *NeurIPS*. 2022. URL: http://papers.nips.cc/paper%5C_files/paper/2022/hash/8636419dea1aa9fbd25fc4248e702da4-Abstract-Conference.html.

[109] Naman Jain, Skanda Vaidyanath, Arun Shankar Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram K. Rajamani, and Rahul Sharma. "Jigsaw: Large Language Models meet Program Synthesis". In: *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 2022, pp. 1219–1231. URL: https://doi.org/10.1145/3510003.3510203.

[110] Gust Verbruggen, Vu Le, and Sumit Gulwani. "Semantic programming by example with pre-trained models". In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–25. URL: https://doi.org/10.1145/3485477.

[111] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Kim M. Hazelwood, Gabriel Synnaeve, and Hugh Leather. "Large Language Models for Compiler Optimization". In: *CoRR* abs/2309.07062 (2023). arXiv: 2309.07062. URL: https://doi.org/10.48550/arXiv.2309.07062.

[112] Harshit Joshi, Abishai Ebenezer, José Cambronero, Sumit Gulwani, Aditya Kanade, Vu Le, Ivan Radicek, and Gust Verbruggen. "FLAME: A small language model for spreadsheet formulas". In: *CoRR* abs/2301.13779 (2023).

[113] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. "PHOG: Probabilistic Model for Code". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016.* Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 2933–2942. URL: http://proceedings.mlr.press/v48/bielik16.html.

[114] Grigory Fedyukovich, Yueling Zhang, and Aarti Gupta. "Syntax-Guided Termination Analysis". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 124–143. URL: https://doi.org/10.1007/978-3-319-96145-3%5C_7.

[115] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivancic, and Aarti Gupta. "Dynamic inference of likely data preconditions over predicates by tree learning". In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*. Ed. by Barbara G. Ryder and Andreas Zeller. ACM, 2008, pp. 295–306. URL: https://doi.org/10.1145/1390630.1390666.

[116] Siddharth Krishna, Christian Puhrsch, and Thomas Wies. "Learning Invariants using Decision Trees". In: *CoRR* abs/1501.04725 (2015). arXiv: 1501.04725. URL: http://arxiv.org/abs/1501.04725.

[117] Rongchen Xu, Fei He, and Bow-Yaw Wang. "Interval counterexamples for loop invariant learning". In: *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. Ed. by Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann. ACM, 2020, pp. 111–122. URL: https://doi.org/10.1145/3368089.3409752.

[118] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. "Learning invariants using decision trees and implication counterexamples". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 499–512. URL: https://doi.org/10.1145/2837614.2837664.

[119] He Zhu, Stephen Magill, and Suresh Jagannathan. "A data-driven CHC solver". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by Jeffrey S. Foster and Dan Grossman. ACM, 2018, pp. 707–721. URL: https://doi.org/10.1145/3192366.3192416.

[120] Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. "Automatic loop-invariant generation and refinement through selective sampling". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. Ed. by Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen. IEEE Computer Society, 2017, pp. 782–792. URL: https://doi.org/10.1109/ASE.2017.8115689.

[121] Rahul Sharma, Aditya V. Nori, and Alex Aiken. "Interpolants as Classifiers". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 71–87. URL: https://doi.org/10.1007/978-3-642-31424-7%5C_11.

[122] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. "Learning Loop Invariants for Program Verification". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada.* Ed. by Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett. 2018, pp. 7762–7773. URL: https://proceedings.neurips.cc/paper/2018/hash/65b1e92c585fd4c2159d5f33b5030ff2-Abstract.html.

[123] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. "Code2Inv: A Deep Learning Framework for Program Verification". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II.* Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 151–164. URL: https://doi.org/10.1007/978-3-030-53291-8%5C_9.

[124] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. "Learning nonlinear loop invariants with gated continuous logic networks". In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020.* Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 106–120. URL: https://doi.org/10.1145/3385412.3385986.

[125] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. *Finding Inductive Loop Invariants using Large Language Models.* 2023. arXiv: 2311.07948 [cs.PL].

[126] Jan Haltermann and Heike Wehrheim. "Machine Learning Based Invariant Generation: A Framework and Reproducibility Study". In: *15th IEEE Conference on Software Testing, Verification and Validation, ICST 2022, Valencia, Spain, April 4-14, 2022.* IEEE, 2022, pp. 12–23. URL: https://doi.org/10.1109/ICST53961.2022.00012.

[127] Jonathan Laurent and André Platzer. "Learning to Find Proofs and Theorems by Learning to Refine Search Strategies: The Case of Loop Invariant Synthesis". In: *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022.* Ed. by Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh. 2022. URL: http://papers.nips.cc/paper%5C_files/paper/2022/hash/1f14ac136d55c34a18a04ce3db083599-Abstract-Conference.html.

[128] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[129] Ashish Tiwari. "Termination of Linear Programs". In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings.* Ed. by Rajeev Alur and Doron A. Peled. Vol. 3114. Lecture Notes in Computer Science. Springer, 2004, pp. 70–82. URL: https://doi.org/10.1007/978-3-540-27813-9%5C_6.

[130] Mark Braverman. "Termination of Integer Linear Programs". In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 372–385. URL: https://doi.org/10.1007/11817963%5C_34.

[131] Eike Neumann, Joël Ouaknine, and James Worrell. "On Ranking Function Synthesis and Termination for Polynomial Programs". In: *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*. Ed. by Igor Konnov and Laura Kovács. Vol. 171. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 15:1–15:15. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2020.15.

[132] Mehran Hosseini, Joël Ouaknine, and James Worrell. "Termination of Linear Loops over the Integers". In: *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece.* Ed. by Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi. Vol. 132. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 118:1–118:13. URL: https://doi.org/10.4230/LIPIcs.ICALP.2019.118.

[133] Florian Frohn and Jürgen Giesl. "Termination of Triangular Integer Loops is Decidable". In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11562. Lecture Notes in Computer Science. Springer, 2019, pp. 426–444. URL: https://doi.org/10.1007/978-3-030-25543-5%5C_24.

[134] Andreas Podelski and Adrey Rybalchenko. "A Complete Method for the Synthesis of Linear Ranking Functions". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Bernhard Steffen and Giorgio Levi. Springer, 2004, pp. 239–251.

[135] Max Dauchet and Sophie Tison. "The Theory of Ground Rewrite Systems is Decidable". In: *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*. IEEE Computer Society, 1990, pp. 242–248. URL: https://doi.org/10.1109/LICS.1990.113750.

[136] Andreas Podelski and Andrey Rybalchenko. "A Complete Method for the Synthesis of Linear Ranking Functions". In: *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*. Ed. by Bernhard Steffen and Giorgio Levi. Vol. 2937. Lecture Notes in Computer Science. Springer, 2004, pp. 239–251. URL: https://doi.org/10.1007/978-3-540-24622-0%5C_20.

[137] Amir M. Ben-Amram and Samir Genaim. "Ranking Functions for Linear-Constraint Loops". In: *J. ACM* 61.4 (2014), 26:1–26:55. URL: https://doi.org/10.1145/2629488.

[138] Laure Gonnord, David Monniaux, and Gabriel Radanne. “Synthesis of ranking functions using extremal counterexamples”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Stephen M. Blackburn. ACM, 2015, pp. 608–618. URL: https://doi.org/10.1145/2737924.2737976.

[139] Patrick Cousot. “Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming”. In: *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*. Ed. by Radhia Cousot. Vol. 3385. Lecture Notes in Computer Science. Springer, 2005, pp. 1–24. URL: https://doi.org/10.1007/978-3-540-30579-8%5C_1.

[140] Yinghua Chen, Bican Xia, Lu Yang, Naijun Zhan, and Chaochen Zhou. “Discovering Non-linear Ranking Functions by Solving Semi-algebraic Systems”. In: *ICTAC*. Vol. 4711. Lecture Notes in Computer Science. Springer, 2007, pp. 34–49.

[141] Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. “Ranking function synthesis for bit-vector relations”. In: *Formal Methods Syst. Des.* 43.1 (2013), pp. 93–120. URL: https://doi.org/10.1007/s10703-013-0186-4.

[142] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. “Linear Ranking with Reachability”. In: *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. Ed. by Kousha Etessami and Sriram K. Rajamani. Vol. 3576. Lecture Notes in Computer Science. Springer, 2005, pp. 491–504. URL: https://doi.org/10.1007/11513988%5C_48.

[143] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. “The Polyranking Principle”. In: *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*. Ed. by Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung. Vol. 3580. Lecture Notes in Computer Science. Springer, 2005, pp. 1349–1361. URL: https://doi.org/10.1007/11523468%5C_109.

[144] Jan Leike and Matthias Heizmann. “Ranking Templates for Linear Loops”. In: *Log. Methods Comput. Sci.* 11.1 (2015). URL: https://doi.org/10.2168/LMCS-11(1:16)2015.

[145] Caterina Urban. “The Abstract Domain of Segmented Ranking Functions”. In: *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. Ed. by Francesco Logozzo and Manuel Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer, 2013, pp. 43–62. URL: https://doi.org/10.1007/978-3-642-38856-9%5C_5.

[146] Caterina Urban. “FuncTion: An Abstract Domain Functor for Termination - (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Christel Baier and

Cesare Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 464–466. URL: https://doi.org/10.1007/978-3-662-46681-0%5C_46.

[147] Andreas Podelski and Andrey Rybalchenko. “Transition Invariants”. In: *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*. IEEE Computer Society, 2004, pp. 32–41. URL: https://doi.org/10.1109/LICS.2004.1319598.

[148] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. “Termination proofs for systems code”. In: *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*. Ed. by Michael I. Schwartzbach and Thomas Ball. ACM, 2006, pp. 415–426. URL: https://doi.org/10.1145/1133981.1134029.

[149] Byron Cook, Abigail See, and Florian Zuleger. “Ramsey vs. Lexicographic Termination Proving”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Nir Piterman and Scott A. Smolka. Vol. 7795. Lecture Notes in Computer Science. Springer, 2013, pp. 47–61. URL: https://doi.org/10.1007/978-3-642-36742-7%5C_4.

[150] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. “Termination Analysis with Compositional Transition Invariants”. In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. Ed. by Tayssir Touili, Byron Cook, and Paul B. Jackson. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 89–103. URL: https://doi.org/10.1007/978-3-642-14295-6%5C_9.

[151] Jianhui Chen and Fei He. “Proving Termination by k-Induction”. In: *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 1239–1243. URL: https://doi.org/10.1145/3324884.3418929.

[152] Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. “Synthesizing Ranking Functions from Bits and Pieces”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Marsha Chechik and Jean-François Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 54–70. URL: https://doi.org/10.1007/978-3-662-49674-9%5C_4.

[153] Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. “DynamiTe: dynamic termination and non-termination proofs”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 189:1–189:30. URL: https://doi.org/10.1145/3428257.

[154] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. "Proving Conditional Termination". In: *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*. Ed. by Aarti Gupta and Sharad Malik. Vol. 5123. Lecture Notes in Computer Science. Springer, 2008, pp. 328–340. URL: https://doi.org/10.1007/978-3-540-70545-1%5C_32.

[155] Aliaksei Tsitovich, Natasha Sharygina, Christoph M. Wintersteiger, and Daniel Kroening. "Loop Summarization and Termination Analysis". In: *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Ed. by Parosh Aziz Abdulla and K. Rustan M. Leino. Vol. 6605. Lecture Notes in Computer Science. Springer, 2011, pp. 81–95. URL: https://doi.org/10.1007/978-3-642-19835-9%5C_9.

[156] Patrick Cousot and Radhia Cousot. "An abstract interpretation framework for termination". In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. Ed. by John Field and Michael Hicks. ACM, 2012, pp. 245–258. URL: https://doi.org/10.1145/2103656.2103687.

[157] Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. "Bit-Precise Procedure-Modular Termination Analysis". In: *ACM Trans. Program. Lang. Syst.* 40.1 (2018), 1:1–1:38. URL: https://doi.org/10.1145/3121136.

[158] Shaowei Zhu and Zachary Kincaid. "Termination analysis without the tears". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 1296–1311. URL: https://doi.org/10.1145/3453483.3454110.

[159] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. "Analyzing Program Termination and Complexity Automatically with AProVE". In: *J. Autom. Reason.* 58.1 (2017), pp. 3–31. URL: https://doi.org/10.1007/s10817-016-9388-y.

[160] Marc Brockschmidt, Carsten Otto, Christian von Essen, and Jürgen Giesl. "Termination Graphs for Java Bytecode". In: *Verification, Induction, Termination Analysis - Festschrift for Christoph Walther on the Occasion of His 60th Birthday*. Ed. by Simon Siegler and Nathan Wasser. Vol. 6463. Lecture Notes in Computer Science. Springer, 2010, pp. 17–37. URL: https://doi.org/10.1007/978-3-642-17172-7%5C_2.

[161] Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. "Automated Termination Analysis of Java Bytecode by Term Rewriting". In: *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK*. Ed. by Christopher Lynch. Vol. 6. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für

Informatik, 2010, pp. 259–276. URL: https://doi.org/10.4230/LIPIcs.RTA.2010.259.

[162] Fausto Spoto. "The Julia Static Analyzer for Java". In: *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*. Ed. by Xavier Rival. Vol. 9837. Lecture Notes in Computer Science. Springer, 2016, pp. 39–57. URL: https://doi.org/10.1007/978-3-662-53413-7%5C_3.

[163] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. "COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode". In: *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever. Vol. 5382. Lecture Notes in Computer Science. Springer, 2007, pp. 113–132. URL: https://doi.org/10.1007/978-3-540-92188-2%5C_5.

[164] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. "Termination Analysis by Learning Terminating Programs". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 797–813. URL: https://doi.org/10.1007/978-3-319-08867-9%5C_53.

[165] Yu-Fang Chen, Matthias Heizmann, Ondrej Lengál, Yong Li, Ming-Hsien Tsai, Andrea Turrini, and Lijun Zhang. "Advanced automata-based algorithms for program termination checking". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by Jeffrey S. Foster and Dan Grossman. ACM, 2018, pp. 135–150. URL: https://doi.org/10.1145/3192366.3192405.

[166] Marc Brockschmidt, Richard Musiol, Carsten Otto, and Jürgen Giesl. "Automated Termination Proofs for Java Programs with Cyclic Data". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 105–122. URL: https://doi.org/10.1007/978-3-642-31424-7%5C_13.

[167] Aditya V. Nori and Rahul Sharma. "Termination proofs from tests". In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. Ed. by Bertrand Meyer, Luciano Baresi, and Mira Mezini. ACM, 2013, pp. 246–256. URL: https://doi.org/10.1145/2491411.2491413.

[168] Yue Yuan and Yi Li. "Ranking Function Detection via SVM: A More General Method". In: *IEEE Access* 7 (2019), pp. 9971–9979. URL: https://doi.org/10.1109/ACCESS.2018.2890692.

[169] Yi Li, Xuechao Sun, Yong Li, Andrea Turrini, and Lijun Zhang. "Synthesizing Nested Ranking Functions for Loop Programs via SVM". In: *Formal Methods and Software Engineering - 21st International Conference on Formal Engineering Methods, ICFEM 2019, Shenzhen, China, November 5-9, 2019, Proceedings.* Ed. by Yamine Aït Ameur and Shengchao Qin. Vol. 11852. Lecture Notes in Computer Science. Springer, 2019, pp. 438–454. URL: https://doi.org/10.1007/978-3-030-32409-4%5C_27.

[170] Satoshi Kura, Hiroshi Unno, and Ichiro Hasuo. "Decision Tree Learning in CEGIS-Based Termination Analysis". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II.* Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 75–98. URL: https://doi.org/10.1007/978-3-030-81688-9%5C_4.

[171] Wang Tan and Yi Li. "Synthesis of ranking functions via DNN". In: *Neural Comput. Appl.* 33.16 (2021), pp. 9939–9959. URL: https://doi.org/10.1007/s00521-021-05763-8.

[172] Rongchen Xu, Jianhui Chen, and Fei He. "Data-Driven Loop Bound Learning for Termination Analysis". In: *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 2022, pp. 499–510. URL: https://doi.org/10.1145/3510003.3510220.

[173] Yoav Alon and Cristina David. "Using graph neural networks for program termination". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022.* Ed. by Abhik Roychoudhury, Cristian Cadar, and Miryung Kim. ACM, 2022, pp. 910–921. URL: https://doi.org/10.1145/3540250.3549095.

[174] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. "Proving non-termination". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008.* Ed. by George C. Necula and Philip Wadler. ACM, 2008, pp. 147–158. URL: https://doi.org/10.1145/1328438.1328459.

[175] Zhilei Han and Fei He. "Data-driven Recurrent Set Learning For Non-termination Analysis". In: *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023.* IEEE, 2023, pp. 1303–1315. URL: https://doi.org/10.1109/ICSE48619.2023.00115.

[176] Alessandro Abate, Daniele Ahmed, Mirco Giacobbe, and Andrea Peruffo. "Formal Synthesis of Lyapunov Neural Networks". In: *IEEE Control. Syst. Lett.* 5.3 (2021), pp. 773–778. URL: https://doi.org/10.1109/LCSYS.2020.3005328.

[177] Alessandro Abate, Mirco Giacobbe, and Diptarko Roy. "Learning Probabilistic Termination Proofs". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II.* Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 3–26. URL: https://doi.org/10.1007/978-3-030-81688-9%5C_1.

[178] T.M. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997. URL: https://books.google.at/books?id=EoYBngEACAAJ.

[179] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press, 2014.

[180] Tianqi Chen and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. San Francisco, California, USA: ACM, 2016, pp. 785–794. URL: http://doi.acm.org/10.1145/2939672.2939785.

[181] Temur Kutsia. “Anti-Unification: Algorithms and Applications”. In: *27th International Workshop on Unification, UNIF 2013, Eindhoven, Netherlands, June 26, 2013*. Ed. by Konstantin Korovin and Barbara Morawska. Vol. 19. EPiC Series in Computing. EasyChair, 2013, p. 2. URL: https://doi.org/10.29007/jbx2.

[182] Terese. *Term rewriting systems*. Vol. 55. Cambridge tracts in theoretical computer science. Cambridge University Press, 2003.

[183] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. “Programming by Example Using Least General Generalizations”. In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*. Ed. by Carla E. Brodley and Peter Stone. AAAI Press, 2014, pp. 283–290. URL: https://doi.org/10.1609/aaai.v28i1.8744.

[184] Franz Baader and Wayne Snyder. “Unification Theory”. In: *Handbook of Automated Reasoning (in 2 volumes)*. Ed. by John Alan Robinson and Andrei Voronkov. Elsevier and MIT Press, 2001, pp. 445–532. URL: https://doi.org/10.1016/b978-044450813-3/50010-2.

[185] Jörg H. Siekmann. “Unification Theory”. In: *J. Symb. Comput.* 7.3/4 (1989), pp. 207–274. URL: https://doi.org/10.1016/S0747-7171(89)80012-4.

[186] David M. Cerna. “Anti-unification and the theory of semirings”. In: *Theor. Comput. Sci.* 848 (2020), pp. 133–139. URL: https://doi.org/10.1016/j.tcs.2020.10.020.

[187] David M. Cerna and Temur Kutsia. “Idempotent Anti-unification”. In: *ACM Trans. Comput. Log.* 21.2 (2020), 10:1–10:32. URL: https://doi.org/10.1145/3359060.

[188] María Alpuente, Santiago Escobar, Javier Espert, and José Meseguer. “A modular order-sorted equational generalization algorithm”. In: *Inf. Comput.* 235 (2014), pp. 98–136. URL: https://doi.org/10.1016/j.ic.2014.01.006.

[189] Alberto Martelli and Ugo Montanari. “An Efficient Unification Algorithm”. In: *ACM Trans. Program. Lang. Syst.* 4.2 (1982), pp. 258–282. URL: https://doi.org/10.1145/357162.357169.

[190] Gordon D. Plotkin. “A Note on Inductive Generalization”. In: *Machine Intelligence* 5 (1970), pp. 153–163.

[191] Noam Chomsky and Marcel P Schützenberger. "The algebraic theory of context-free languages". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 26. Elsevier, 1959, pp. 118–161.

[192] Guillaume Lample and François Charton. "Deep Learning For Symbolic Mathematics". In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=S1eZYeHFDS.

[193] François Bergeron, Gilbert Labelle, and Pierre Leroux. *Combinatorial Species and Tree-like Structures*. Ed. by MargaretTranslator Readdy. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1997.

[194] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olsák. "Reinforcement Learning of Theorem Proving". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett. 2018, pp. 8836–8847. URL: https://proceedings.neurips.cc/paper/2018/hash/55acf8539596d25624059980986aaa78-Abstract.html.

[195] Levente Kocsis and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning". In: *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Vol. 4212. Lecture Notes in Computer Science. Springer, 2006, pp. 282–293. URL: https://doi.org/10.1007/11871842%5C_29.

[196] Jan Jakubuv and Josef Urban. "ENIGMA: Efficient Learning-Based Inference Guiding Machine". In: *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*. Ed. by Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke. Vol. 10383. Lecture Notes in Computer Science. Springer, 2017, pp. 292–302. URL: https://doi.org/10.1007/978-3-319-62075-6%5C_20.

[197] Karel Chvalovský, Jan Jakubuv, Martin Suda, and Josef Urban. "ENIGMA-NG: Efficient Neural and Gradient-Boosted Inference Guidance for E". In: *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*. Ed. by Pascal Fontaine. Vol. 11716. Lecture Notes in Computer Science. Springer, 2019, pp. 197–215. URL: https://doi.org/10.1007/978-3-030-29436-6%5C_12.

[198] Michael Färber, Cezary Kaliszyk, and Josef Urban. "Machine Learning Guidance for Connection Tableaux". In: *J. Autom. Reason.* 65.2 (2021), pp. 287–320. URL: https://doi.org/10.1007/s10817-020-09576-7.

[199] Geoffrey Irving, Christian Szegedy, Alexander A. Alemi, Niklas Eén, François Chollet, and Josef Urban. "DeepMath - Deep Sequence Models for Premise Selection". In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett. 2016, pp. 2235–2243.

URL: https://proceedings.neurips.cc/paper/2016/hash/f197002b9a0853eca5e046d9ca4663d5-Abstract.html.

[200] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. "The SMT Competition 2015-2018". In: *J. Satisf. Boolean Model. Comput.* 11.1 (2019), pp. 221–259.

[201] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I.* Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442. URL: https://doi.org/10.1007/978-3-030-99524-9%5C_24.

[202] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. URL: https://doi.org/10.1007/978-3-540-78800-3%5C_24.

[203] Hassan Eldib, Meng Wu, and Chao Wang. "Synthesis of Fault-Attack Countermeasures for Cryptographic Circuits". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II.* Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 343–363. URL: https://doi.org/10.1007/978-3-319-41540-6%5C_19.

[204] Dirk Beyer. "Competition on Software Verification and Witness Validation: SV-COMP 2023". In: *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II.* Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Vol. 13994. Lecture Notes in Computer Science. Springer, 2023, pp. 495–522. URL: https://doi.org/10.1007/978-3-031-30820-8%5C_29.

[205] Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. "The Termination and Complexity Competition". In: *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III.* Ed. by Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen. Vol. 11429. Lecture Notes in Computer Science. Springer, 2019, pp. 156–166. URL: https://doi.org/10.1007/978-3-030-17502-3%5C_10.

[206] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. "Counterexample-Guided Quantifier Instantiation for Synthesis in SMT". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9207. LNCS. Springer, 2015, pp. 198–216. URL: https://doi.org/10.1007/978-3-319-21668-3%5C%5F12.

[207] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. "Test input generation with java PathFinder". In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*. Ed. by George S. Avrunin and Gregg Rothermel. ACM, 2004, pp. 97–107. URL: https://doi.org/10.1145/1007512.1007526.

[208] Cristian Cadar and Koushik Sen. "Symbolic execution for software testing: three decades later". In: *Commun. ACM* 56.2 (2013), pp. 82–90. URL: https://doi.org/10.1145/2408776.2408795.

[209] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 209–224. URL: http://www.usenix.org/events/osdi08/tech/full%5C_papers/cadar/cadar.pdf.

[210] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. Retrieved 2021-10-26 15:30:20+02:00. CISPA Helmholtz Center for Information Security, 2021. URL: https://www.fuzzingbook.org/.

[211] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey". In: *Cybersecur.* 1.1 (2018), p. 6. URL: https://doi.org/10.1186/s42400-018-0002-y.

[212] Mauro Baluda, Giovanni Denaro, and Mauro Pezzè. "Bidirectional Symbolic Analysis for Effective Branch Testing". In: *IEEE Trans. Software Eng.* 42.5 (2016), pp. 403–426. URL: https://doi.org/10.1109/TSE.2015.2490067.

[213] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. "Adaptive Random Testing: The ART of test case diversity". In: *J. Syst. Softw.* 83.1 (2010), pp. 60–66. URL: https://doi.org/10.1016/j.jss.2009.02.022.

[214] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. "Feedback-Directed Random Test Generation". In: *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 2007, pp. 75–84. URL: https://doi.org/10.1109/ICSE.2007.37.

[215] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations". In: *J. Mach. Learn. Res.* 18 (2017), 187:1–187:30. URL: http://jmlr.org/papers/v18/16-456.html.

[216] Thomas A. Henzinger, Mathias Lechner, and Dorde Zikelic. "Scalable Verification of Quantized Neural Networks". In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021.* AAAI Press, 2021, pp. 3787–3795. URL: https://doi.org/10.1609/aaai.v35i5.16496.

[217] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. "A Survey of Quantization Methods for Efficient Neural Network Inference". In: *CoRR* abs/2103.13630 (2021). arXiv: 2103.13630. URL: https://arxiv.org/abs/2103.13630.

[218] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. "Intriguing properties of neural networks". In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings.* Ed. by Yoshua Bengio and Yann LeCun. 2014. URL: http://arxiv.org/abs/1312.6199.

[219] Luca Pulina and Armando Tacchella. "Challenging SMT solvers to verify neural networks". In: *AI Commun.* 25.2 (2012), pp. 117–135. URL: https://doi.org/10.3233/AIC-2012-0525.

[220] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher A. Strong, Clark W. Barrett, and Mykel J. Kochenderfer. "Algorithms for Verifying Deep Neural Networks". In: *Found. Trends Optim.* 4.3-4 (2021), pp. 244–404. URL: https://doi.org/10.1561/2400000035.

[221] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. "Safety Verification of Deep Neural Networks". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I.* Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10426. Lecture Notes in Computer Science. Springer, 2017, pp. 3–29. URL: https://doi.org/10.1007/978-3-319-63387-9%5C_1.

[222] Rüdiger Ehlers. "Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks". In: *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings.* Ed. by Deepak D'Souza and K. Narayan Kumar. Vol. 10482. Lecture Notes in Computer Science. Springer, 2017, pp. 269–286. URL: https://doi.org/10.1007/978-3-319-68167-2%5C_19.

[223] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin T. Vechev. "Fast and Effective Robustness Certification". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada.* Ed. by Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett. 2018, pp. 10825–10836. URL: https://proceedings.neurips.cc/paper/2018/hash/f2f446980d8e971ef3da97af089481c3-Abstract.html.

[224] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. “Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks”. In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10426. Lecture Notes in Computer Science. Springer, 2017, pp. 97–117. URL: https://doi.org/10.1007/978-3-319-63387-9%5C_5.

[225] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. “Efficient Neural Network Robustness Certification with General Activation Functions”. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett. 2018, pp. 4944–4953. URL: https://proceedings.neurips.cc/paper/2018/hash/d04863f100d59b3eb688a11f95b0ae60-Abstract.html.

[226] 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: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 443–452. URL: https://doi.org/10.1007/978-3-030-25540-4%5C_26.

[227] Patrick Henriksen and Alessio R. Lomuscio. “Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search”. In: *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*. Ed. by Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang. Vol. 325. Frontiers in Artificial Intelligence and Applications. IOS Press, 2020, pp. 2513–2520. URL: https://doi.org/10.3233/FAIA200385.

[228] David Shriver, Sebastian G. Elbaum, and Matthew B. Dwyer. “Reducing DNN Properties to Enable Falsification with Adversarial Attacks”. In: *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 275–287. URL: https://doi.org/10.1109/ICSE43902.2021.00036.

[229] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and Pawan Kumar Mudigonda. “A Unified View of Piecewise Linear Neural Network Verification”. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett. 2018, pp. 4795–4804. URL: https://proceedings.neurips.cc/paper/2018/hash/be53d253d6bc3258a8160556dda3e9b2-Abstract.html.

[230] Panagiotis Kouvaros and Alessio Lomuscio. “Towards Scalable Complete Verification of Relu Neural Networks via Dependency-based Branching”. In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*. Ed. by Zhi-Hua Zhou. ijcai.org, 2021, pp. 2643–2650. URL: https://doi.org/10.24963/ijcai.2021/364.

[231] Hoang-Dung Tran, Stanley Bak, Weiming Xiang, and Taylor T. Johnson. “Verification of Deep Convolutional Neural Networks Using ImageStars”. In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12224. Lecture Notes in Computer Science. Springer, 2020, pp. 18–42. URL: https://doi.org/10.1007/978-3-030-53288-8%5C_2.

[232] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M. Wintersteiger. “Loop Summarization Using Abstract Transformers”. In: *Automated Technology for Verification and Analysis, 6th International Symposium, ATVA 2008, Seoul, Korea, October 20-23, 2008. Proceedings*. Ed. by Sung Deok Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan. Vol. 5311. Lecture Notes in Computer Science. Springer, 2008, pp. 111–125. URL: https://doi.org/10.1007/978-3-540-88387-6%5C_10.

[233] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d’Alché-Buc, Emily B. Fox, and Roman Garnett. 2019, pp. 8024–8035. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[234] 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*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[235] Laura Kovács and Andrei Voronkov. “Finding Loop Invariants for Programs over Arrays Using a Theorem Prover”. In: *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Marsha Chechik and Martin Wirsing. Vol. 5503. Lecture Notes in Computer Science. Springer, 2009, pp. 470–485. URL: https://doi.org/10.1007/978-3-642-00593-0%5C_33.

[236] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. "Invariant Synthesis for Combined Theories". In: *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*. Ed. by Byron Cook and Andreas Podelski. Vol. 4349. Lecture Notes in Computer Science. Springer, 2007, pp. 378–394. URL: https://doi.org/10.1007/978-3-540-69738-1%5C_27.

[237] Shuvendu K. Lahiri and Randal E. Bryant. "Indexed Predicate Discovery for Unbounded System Verification". In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*. Ed. by Rajeev Alur and Doron A. Peled. Vol. 3114. Lecture Notes in Computer Science. Springer, 2004, pp. 135–147. URL: https://doi.org/10.1007/978-3-540-27813-9%5C_11.

[238] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. "Path invariants". In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. Ed. by Jeanne Ferrante and Kathryn S. McKinley. ACM, 2007, pp. 300–309. URL: https://doi.org/10.1145/1250734.1250769.

[239] ThanhVu Nguyen, Matthew B. Dwyer, and Willem Visser. "SymInfer: Inferring Program Invariants using Symbolic States". In: *ASE*. IEEE Computer Society, 2017, pp. 804–814.

[240] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. "The Daikon system for dynamic detection of likely invariants". In: *Sci. Comput. Program.* 69.1-3 (2007), pp. 35–45. URL: https://doi.org/10.1016/j.scico.2007.01.015.

[241] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. "DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants". In: *ACM Trans. Softw. Eng. Methodol.* 23.4 (2014), 30:1–30:30. URL: https://doi.org/10.1145/2556782.

[242] Xinyun Chen, Chang Liu, and Dawn Song. "Tree-to-tree Neural Networks for Program Translation". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett. 2018, pp. 2552–2562. URL: https://proceedings.neurips.cc/paper/2018/hash/d759175de8ea5b1d9a2660e45554894f-Abstract.html.