POLITECHNIKA POZNAŃSKA

Iwo Błądek

Machine Learning and Formal Verification for Acquisition of Knowledge in Heuristic Program Synthesis

Doctoral dissertation submitted in partial fulfilment

of the requirements for the degree of Doctor of Philosophy

Supervisor: Krzysztof Krawiec, Ph. D., Dr. Habil., Professor

Poznań, Poland
2022

Acknowledgments

First, I want to thank my supervisor, **Krzysztof Krawiec**, for all the discussions and work that we have done together and his support during my time as a PhD student – I will be forever impressed by his work ethic. Next, I would like to thank people with whom I had a privilege to work scientifically as an undergraduate student, and if not for them it is not certain that I would end up in this place: **Piotr Formanowicz** (deep discussions about computational complexity and quantum computation), **Maciej Drozdowski** (work on scheduling problems, which resulted in a paper [29]), **Jacek Błażewicz** and **Natalia Szóstak** (our work together on multi-agent simulations for the RNA world hypothesis). Last but not least, there are also people that I had the pleasure to work with and come to know more closely after I became a PhD student: **Maciej Komosiński** and **Konrad Miazga** (for long discussions about science, philosophy, and consciousness, which also resulted in a paper [27]), **Robert Susmaga** (for sharing his deep passion for linear algebra and interesting discussions on many other topics), **Tomasz Pawlak** and **Paweł Liskowski** (for interesting discussions during our joint conference voyages). I could also thank **my family**, but it is so natural that I could probably append it to any acknowledgments I ever write – and despite of that I still do it!

I also received support from Polish National Science Centre, which awarded me grant no. 2018/29/N/ST6/01646 for pursuing several research directions presented in this thesis.

Abstract

Automatic program synthesis is increasingly used to support programmers in software development, enabling automatization for users based on simple cues or examples, and in scientific discovery. However, because the search space grows exponentially with the length of solutions, the exact approaches to program synthesis fall short as their runtime becomes too long. One of the alternatives to limiting the search space is applying heuristic algorithms, which do not provide guarantees of eventually finding the solution, but are often able to solve this task relatively quickly.

In this thesis, we propose and examine four approaches to heuristic program synthesis based on genetic programming (GP). The common feature of these approaches is acquiring some additional information (knowledge) about the particular program synthesis task they are solving, and then extending GP so that it is able to use that information. The first approach, Evolutionary Program Sketching (EPS), uses a framework characteristic for memetic algorithms to complete a provided partial program (sketch) using an optimizing Satisfiability Modulo Theories (SMT) solver, and either replace the original program by the completed sketch, or assign the fitness of the completed sketch. The second approach, Counterexample-Driven Genetic Programming (CDGP), uses formal verification to check if the programs are correct, and if they are not, then an input representing a fault is returned and added to the set of test cases, thus guiding the subsequent search. The third approach, Counterexample-Driven Symbolic Regression (CDSR), adapts CDGP to the problem of symbolic regression with formal constrains, and optimizes for both numerical error on a test set and satisfaction of the constraints. Finally, Neuro-Guided Genetic Programming uses a neural network to predict the conditional probability of instructions that should occur in programs given input-output examples, and then those predictions are used to bias search. Importantly, two of these approaches (CDGP and CDSR) allow synthesizing programs that are provably correct with respect to specification, which makes them close to unique in the domain of metaheuristics, where search algorithms do not offer such guarantees and typically produce approximate solutions only.

Computational experiments demonstrate that all these approaches are feasible and in certain cases can compete with the state-of-the-art exact counterparts. The addition of knowledge in almost all cases improves the efficiency of search compared to the "vanilla" GP.

Uczenie maszynowe i formalna weryfikacja dla pozyskiwania wiedzy w heurystycznej syntezie programów

Streszczenie rozprawy w języku polskim

1 Wprowadzenie

Automatyczna synteza programów komputerowych to problem polegający na znalezieniu programu komputerowego spełniającego pewien zestaw wymagań/ograniczeń, który nazywamy *specyfikacją*. Specyfikacja może przyjmować różne formy, przykładowo zbiór przykładów wejście-wyjście, formalne ograniczenia wyrażone w języku logiki, demonstracja etapów pośrednich wykonania programu, czy też opis zadania w języku naturalnym. Zbiór dopuszczalnych programów określony jest *językiem programowania*, który także jest, obok specyfikacji, częścią opisu danego problemu syntezy.

W literaturze można znaleźć wiele algorytmów rozwiązujących różne warianty problemu syntezy. Podejścia te możemy zaklasyfikować do czterech głównych paradygmatów [77]:

- *Metody pełnego przeglądu*, w których przestrzeń możliwych programów jest systematycznie przeszukiwana.
- *Podejścia oparte o rozwiązywanie ograniczeń*, w których problem syntezy zostaje przetransformowany do innego znanego problemu, na przykład spełnialności formuł logicznych (SAT), i rozwiązany za pomocą dedykowanych dla tego problemu metod.
- *Podejścia dedukcyjne*, w których specyfikacja jest stopniowo przekształcana do oczekiwanego programu, najczęściej za pomocą szeregu predefiniowanych reguł.
- *Techniki statystyczne i stochastyczne*, które oparte są na uczeniu maszynowym lub algorytmach metaheurystycznych, takich jak programowanie genetyczne.

Jako że przestrzeń przeszukiwania możliwych rozwiązań problemu syntezy jest bardzo duża i rośnie wykładniczo wraz z rosnącą długością programów, podejścia dokładne potrzebują dużo czasu by zwrócić rozwiązanie dla bardziej złożonych problemów. Alternatywnym podejściem są algorytmy heurystyczne, które co prawda nie gwarantują rozwiązania problemu syntezy ani znalezienia globalnie najlepszego programu w przypadku optymalizacyjnego wariantu tego problemu (w którym szukamy programu najlepszego według

pewnej funkcji celu), ale w zamian są w stanie zwrócić relatywnie szybko aproksymowane rozwiązanie. W rozprawie skupiamy się na *programowaniu genetycznym* (GP) [103, 159], które jest heurystycznym podejściem do syntezy w którym utrzymywana jest populacja programów wraz z presją selekcyjną na ich jakość, i poprzez mechanizm selekcji oraz operatory wariacji (mutacja, krzyżowanie) znajdowane są rozwiązania o coraz wyższej jakości.

Wspólną cechą wszystkich prac opisanych w niniejszej rozprawie jest wzbogacenie programowania genetycznego o dodatkową informację (wiedzę), która jest następnie wykorzystywana przez różne elementy algorytmu (selekcja, operatory wariacji) do zwiększenia efektywności przeszukiwania, albo wręcz umożliwienia rozwiązywania przez GP problemu syntezy dla danego rodzaju specyfikacji. W rozprawie przedstawione są następujące podejścia:

- *ewolucyjne szkicowanie programów* [30],
- *programowanie genetyczne kierowane kontrprzykładami* [32, 106, 107],
- *regresja symboliczna kierowana kontrprzykładami* [31],
- *programowanie genetyczne wspierane sztuczną siecią neuronową* [122].

2 Ewolucyjne szkicowanie programów

Ewolucyjne szkicowanie programów (ang. Evolutionary Program Sketching, EPS) [30] bazuje na paradygmacie *syntezy programów przez szkicowanie* [174, 176], w którym algorytm syntezy oprócz specyfikacji działania dostaje również na wejście częściowo napisany program („szkic") wraz ze wskazanymi miejscami ("luki"), które powinny zostać wypełnione brakującymi fragmentami programu. Szkicowanie pozwala uniknąć pełnego przeszukiwania ogromnej przestrzeni rozwiązań, a jednocześnie naturalnie wpisuje się w praktyczne zastosowania syntezy programów w sytuacji, kiedy programista/użytkownik ma dość dobre pojęcie jak w ogólności powinna wyglądać struktura programu, ale chce skorzystać z algorytmu syntezy żeby uzupełnić, często wymagające dłuższej analizy, detale.

EPS zwalnia użytkownika z obowiązku przygotowania szkicu, ponieważ są one generowane automatycznie na drodze ewolucji, a następnie uzupełniane przez metodę syntezy opartą o rozwiązywanie ograniczeń. Problem syntezy sprowadzany jest do problemu spełnialności formuł logicznych modulo teorie (ang. Satisfiability Modulo Theories, SMT) [20, 54]. Specyfikacją działania programu są przykłady wejście-wyjście, a zadaniem jest znalezienie programu spełniającego jak najwięcej z nich. Do rozwiązania tego zadania optymalizacji wykorzystany został solwer Z3 [53], który udostępnia narzędzie służące do optymalizacji bazujące na SMT [26].

EPS działa na podobnej zasadzie jak algorytmy memetyczne [140], to znaczy rozwiązania znalezione przez algorytm ewolucyjny są następnie oceniane i potencjalnie modyfikowane na podstawie tego, jak dobrze solwer SMT dał radę je uzupełnić. Zaproponowano dwa warianty podejścia: w wariancie Baldwinowskim (EPS-B) ma miejsce jedynie przypisywanie do danego rozwiązania (szkicu) wartości przystosowania jego uzupełnionej przez solwer SMT wersji. Z kolei w wariancie Lamarckowskim (EPS-L) rozwiązanie jest także zastępowane w populacji przez uzupełniony szkic. Nowe luki dodawane są do programów przez operatory mutacji i inicjalizacji, które traktują luki jako element terminalny gra-

matyki języka programowania o przypisanym typie wskazującym na rodzaj wartości zwracanej przez brakujący fragment kodu.

Przeprowadzono wstępne eksperymenty obliczeniowe, w których zostały ze sobą porównane EPS-B i EPS-L w różnych konfiguracjach, a także sprawdzono czy radzą sobie one lepiej niż samo GP bez mechanizmu szkicowania. Wyniki pokazały, że szkicowanie rzeczywiście daje lepsze wyniki niż standardowe GP, o ile luki są wypełniane przez stałe a nie tylko zmienne wejściowe programu. Oba warianty EPS okazały się znacząco dominować nad GP przy tej samej liczbie pokoleń, przy czym wariant Baldwinowski okazał się zdecydowanie lepszy od Lamarckowskiego. Żeby skompensować długi rzeczywisty czas działania EPS, przetestowane zostały również dwie dodatkowe konfiguracje w których GP miało do dyspozycji taki sam budżet czasu obliczeniowego oraz znacznie większą populację – po tych zmianach EPS-B pozostał zdecydowanie dominującym wariantem jeżeli chodzi o skuteczność, jednak EPS-L okazał się być w tych warunkach gorszym algorytmem niż GP.

3 Programowanie genetyczne kierowane kontrprzykładami

Programowanie genetyczne kierowane kontrprzykładami (ang. Counterexample-Driven Genetic Programming, CDGP) [32, 106, 107] jest próbą zastosowania GP do problemów syntezy programów, w których zadanie jest określone wyłącznie przez formalną specyfikację złożoną z logicznych ograniczeń wyrażonych w logice pierwszego rzędu rozszerzonej teoriami. Teorie pozwalają wzbogacić semantykę wyrażeń logicznych i znacząco ułatwić przygotowywanie formalnych specyfikacji; w tej pracy skupiliśmy się na dwóch z nich: liniowej arytmetyce liczb całkowitych (Linear Integer Arithmetic, LIA), oraz operacjach na ciągach znaków i liczbach (Strings with Linear Integer Arithmetic, SLIA).

Problem z bezpośrednim użyciem GP do zadań syntezy na podstawie formalnej specyfikacji polega na trudności w skonstruowaniu odpowiedniej funkcji celu, gdyż GP realizuje zadanie optymalizacji, podczas gdy synteza na podstawie formalnej specyfikacji to problem przeszukiwania. Problem ten próbowano rozwiązać poprzez zliczanie spełnionych indywidualnych ograniczeń [82, 88], wyróżnianie poziomu spełnienia danego ograniczenia [94, 95], czy też wykorzystanie kontrprzykładów pochodzących z nieudanych weryfikacji [96, 97].

CDGP również jest oparte o zbieranie kontrprzykładów, które po przekształceniu do regularnych testów wykorzystywane są do obliczania miary przystosowania programów w populacji. W odróżnieniu od wcześniej opisanych prac, wykorzystujemy to podejście do rozwiązywania bardziej złożonych problemów i wprowadzamy dodatkowy warunek który musi spełnić program zanim zostanie poddany weryfikacji. Początkowo zbiór testów w CDGP jest pusty, i może ulec powiększeniu gdy program w populacji spełni procent α (parametr algorytmu) już zebranych testów i zostanie poddane formalnej weryfikacji przy użyciu solwera SMT. Celem weryfikacji jest formalne udowodnienie poprawności programu dla każdego wejścia; proces ten przeprowadzany jest z pomocą solwera SMT, który pozwala uniknąć testowania programu na wszystkich możliwych wejściach. Jeżeli weryfikacja zakończy się sukcesem, to CDGP kończy działanie, ponieważ program spełniający formalną specyfikację został znaleziony. W przeciwnym wypadku, tworzony jest nowy test na bazie kontrprzykładu zwróconego przez solwer SMT i dodawany jest do zbioru testów.

Zastosowany tutaj mechanizm pozwala ograniczyć liczbę kosztownych weryfikacji kiedy posiadamy informację, że program jest niepoprawny (nie spełnia wszystkich testów). Z drugiej strony niższe wartości parametru α pozwalają uzyskać więcej testów i dostarczać tym samym więcej informacji ewolucyjnemu algorytmowi przeszukiwania.

Eksperymenty obliczeniowe wykazały, że w dziedzinie LIA CDGP działa gorzej niż formalne metody syntezy z którymi go porównywaliśmy (pełnoprzeglądowy EUSolver [11], oraz oparty o rozwiązywanie ograniczeń CVC4 [169]) – osiąga gorsze rezultaty w znacznie gorszym czasie. Jednak LIA to relatywnie prosta teoria, dla znacznie trudniejszej SLIA to CDGP okazało się uzyskiwać lepsze rezultaty. Programy znalezione przez CDGP były też często, dla obu problemów, wielokrotnie krótsze niż te znalezione przez metody dokładne. Inne wnioski z eksperymentów obliczeniowych to bardzo wysoka skuteczność algorytmu selekcji lexicase [178] w porównaniu do selekcji turniejowej, lekka przewaga wartości $\alpha = 0.75$ nad innymi testowanymi wartościami, a także obserwacja, że kontrprzykłady generowane przez solwer SMT pozwalają uzyskać lepsze rezultaty niż te generowane losowo.

4 Regresja symboliczna kierowana kontrprzykładami

Regresja symboliczna kierowana kontrprzykładami (ang. Counterexample-Driven Symbolic Regression, CDSR) [31] to efekt adaptacji CDGP do rozwiązywania problemów regresji symbolicznej. Regresja symboliczna polega na znalezieniu wyrażenia matematycznego, które możliwie dobrze wyjaśni zbiór przykładów wejście-wyjście (zbiór uczący). Jest to problem z dziedziny uczenia maszynowego, ponieważ oczekujemy że zaproponowana formuła będzie również trafnie przewidywać wartości dla przykładów spoza, potencjalnie obarczonego szumem i błędami (np. pomiarowymi), zbioru uczącego. W rozważanym przez nas scenariuszu, poza zbiorem testów użytkownik dostarcza także zbiór ograniczeń logicznych (przykładowo, wymagając od formuły symetrii albo monotoniczności), i tak zdefiniowane ogólniejsze zadanie nazywamy *regresją symboliczną z formalnymi ograniczeniami* (Symbolic Regression with Formal Constraints, SRFC) [31].

W porównaniu do CDGP, w CDSR zmienione zostały głównie dwa elementy. Po pierwsze, ze zbioru uczącego wydzielony został zbiór walidacyjny, który zapobiega przeuczeniu i kończy działanie algorytmu kiedy błąd na zbiorze uczącym nie ulega poprawie przez pewną liczbę pokoleń. Po drugie, musieliśmy przyjąć próg błędu (domyślnie 5% odchyłu od oczekiwanego wyjścia) przy którym uznajemy testy za spełnione dla kryterium weryfikacji opartym na parametrze α. Opracowaliśmy również CDSR_p, wariant CDSR w którym spełnienie indywidualnych ograniczeń jest uwzględnione w wektorze przystosowania rozwiązań, dzięki czemu spełnianie ograniczeń ma bezpośredni wpływ na presję selekcyjną.

Spośród testowanych wariantów CDSR, w eksperymentach CDSR_p okazał się zdecydowanie najlepszy jeżeli chodzi o spełnianie ograniczeń, podczas gdy pod względem błędu średniokwadratowego na zbiorze testowym nieco lepszy okazał się standardowy wariant CDSR. Przeprowadziliśmy również kompleksowe porównanie CDSR z szeregiem klasycznych algorytmów uczenia maszynowego dla problemu regresji. Algorytmy te nie przyjmują formalnych ograniczeń jako danych uczących, i celem eksperymentu było sprawdzenie, czy pomimo to potrafią te ograniczenia spełnić poprzez naukę na przykładach. Nieco

wbrew naszym oczekiwaniom okazało się, że najlepsze algorytmy regresji spełniają średnio więcej ograniczeń niż CDSR_p. CDSR_p jednak wyraźnie częściej daje radę spełnić wszystkie ograniczenia na raz oraz jest znacznie skuteczniejszy w spełnianiu pewnych rodzajów ograniczeń. Z kolei standardowy wariant CDSR zdołał uzyskać mniejszy błąd na zbiorze testowym.

5 Programowanie genetyczne wspierane sztuczną siecią neuronową

Prace nad programowaniem genetycznym wspieranym sztuczną siecią neuronową (ang. Neuro-Guided Genetic Programing) [122] były zainspirowane podejściem DeepCoder [17], w którym problem syntezy programów rozwiązywany jest w dwóch fazach. W pierwszej fazie, model uczenia maszynowego (sztuczna sieć neuronowa) uczony jest rozkładu prawdopodobieństwa wystąpienia instrukcji w programie pod warunkiem przykładów wejście-wyjście. Co istotne, proces uczenia przeprowadzany jest tylko raz dla wybranej domeny problemów. W drugiej fazie, nauczony model jest wykorzystywany jako wsparcie dla zewnętrznego algorytmu przeszukiwania potrafiącego wykorzystać predykcje sieci neuronowej przy rozwiązywaniu konkretnych problemów syntezy (potencjalnie innych, niż na których sieć była uczona w pierwszej fazie). Przykładowo, takim algorytmem może być zmodyfikowany wariant DFS (ang. depth-first search), który konstruuje drzewa reprezentujące programy metodą priorytetyzowanego przeglądu, dając pierwszeństwo przeszukiwaniu gałęzi odpowiadających bardziej prawdopodobnym instrukcjom według wskazań sieci neuronowej.

Nasz przyczynek w zakresie tego podejścia do syntezy programów miała charakter eksperymentalny, i polegała na jego zastosowaniu razem z GP oraz przetestowaniu, jak parametryzacja GP wpływać będzie na skuteczność tego podejścia. Eksperymenty przeprowadzone zostały przy pomocy naszej własnej implementacji sieci neuronowej oraz modułu generującego dane uczące, bazujących jednak mocno na tych zastosowanych w DeepCoder [17]. Wykazały one, że jest istotna poprawa efektywności dla GP wspieranego przez sieć neuronową w porównaniu do wariantów bez takiego wsparcia lub z prostym obciążeniem faworyzującym instrukcje najczęściej występujące w zbiorze uczącym. Zaobserowano także, że zastosowanie predykcji sieci także podczas inicjalizacji GP daje bardzo wyraźną poprawę w porównaniu do używania predykcji sieci wyłącznie podczas mutacji.

6 Podsumowanie

Niniejsza rozprawa proponuje cztery heurystyczne podejścia do syntezy programów oparte na GP i zdobywające dodatkową wiedzę o zadaniu syntezy za pomocą formalnej weryfikacji/optymalizacji przy użyciu solwera SMT lub uczenia maszynowego. Najważniejsze rezultaty rozprawy to:

- Ewolucyjne podejście do syntezy programów przez szkicowanie, w którym szkice są generowane automatycznie przez GP.

- Podejście do syntezy programów na podstawie formalnej specyfikacji przy użyciu GP, oparte na kontrprzykładach uzyskanych z formalnej weryfikacji niepoprawnych programów.
- Definicja zadania regresji symbolicznej z formalnymi ograniczeniami.
- Opracowanie algorytmu wykorzystującego GP do rozwiązywania tego zadania, i dogłębne porównanie go z klasycznymi algorytmami uczenia maszynowego dla problemu regresji.
- Implementacja i przetestowanie podejścia do rozwiązywania problemu syntezy użytego w DeepCoder [17] w reżimie obliczeń ewolucyjnych.

Preface

Some ideas, figures, and portions of text presented in this dissertation have appeared previously in the following publications:

- Iwo Błądek and Krzysztof Krawiec. Evolutionary Program Sketching. In Mauro Castelli, James McDermott, and Lukas Sekanina, editors, *EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming*, volume 10196 of *LNCS*, pages 3–18, Amsterdam, 19-21 April 2017. Springer Verlag
- Krzysztof Krawiec, Iwo Błądek, and Jerry Swan. Counterexample-Driven Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 953–960, New York, NY, USA, 2017. ACM
- Krzysztof Krawiec, Iwo Błądek, Jerry Swan, and John H. Drake. Counterexample-Driven Genetic Programming: Stochastic Synthesis of Provably Correct Programs. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 5304–5308. International Joint Conferences on Artificial Intelligence Organization, 7 2018
- Iwo Błądek, Krzysztof Krawiec, and Jerry Swan. Counterexample-Driven Genetic Programming: Heuristic Program Synthesis from Formal Specifications. *Evolutionary Computation*, 26(3):441–469, Fall 2018
- Paweł Liskowski, Iwo Błądek, and Krzysztof Krawiec. Neuro-Guided Genetic Programming: Prioritizing Evolutionary Search with Neural Networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, page 1143–1150, New York, NY, USA, 2018. Association for Computing Machinery
- Iwo Błądek and Krzysztof Krawiec. Solving Symbolic Regression Problems with Formal Constraints. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pages 977–984, New York, NY, USA, 2019. ACM

Contents

Chapter 1

Introduction

1.1 Motivation

It is often said that information is more valuable than gold. In the current so-called "information society" this seems particularly evident – information often makes the difference between success and defeat in business, war, or social interactions. Information in itself, however, is static, like books in a library. It is the flow and processing of information, and ultimately applying it to the real world problems, that makes it valuable.

Before the 1940s, the word "computer" more often than not denoted a person whose job was to perform repetitive computations. It was not until the advent of programmable electronic computing devices, today known as *computers*, that humans became able to relegate the task of information processing to machines. Instead of constructing a separate device for every kind of computation one may want to perform, we can build a single machine with behavior directed by a replaceable piece of code in its memory – a *program*. This is how the division between *hardware* and *software* was born. The formal models of computation were also created around that time, such as Turing machines [189] and λ-calculus [43, 44], which formalized the intuitive notion of an *algorithm* and allowed for many deep insights into what can theoretically be computed and what are the bounds on the resources (memory, time) needed to solve particular computational problems.

Due to the enormous usefulness of computers as general information processors, they are prevalent in almost all aspects of contemporary society. Our lives were transformed by the computational power, ease of communication, and flexibility that these devices offer. As a side effect, the complexity of software also has been increasing to satisfy the ever-growing expectations, and producing reliable software became challenging and time-consuming.

One of the potential ways to facilitate the production of software and to make it more reliable is *automatic program synthesis*, which poses the task of generating a program consistent with the user intent as a search or optimization problem. There are several existing approaches to program synthesis, and we review them in Chapter 2. Unfortunately, the exact approaches to program synthesis are hindered by the exponential size of the search space of programs that needs to be searched. To address this challenge, in this thesis we resort to the means offered by heuristic algorithms and machine learning, namely genetic programming (Section 4.2) and neural networks (Chapter 8). While not providing

guarantees that an optimal solution will be found, heuristic algorithms have high chance of finding relatively good solutions faster than the exact approaches, which often return either an optimal solution or no solution at all.

1.2 Aims and scope

The particular focus of this thesis is on approaches which, despite relying on heuristics for search and optimization, are still guaranteed to synthesize, upon successful termination, programs that are provably correct or provably optimal. The common feature of all the approaches that we propose is acquiring some additional information (knowledge) about the particular program synthesis task they are solving, and then using it to make the search with genetic programming more effective, or even allow it to solve a particular type of program synthesis problems that are otherwise beyond its reach. This ties closely to the notion of *prioritization* of search, i.e., introducing a bias to the algorithm so that it visits the promising regions of the search space first, and thus on average its effectiveness increases for the computational problem of interest.

To perform effective prioritization, a search algorithm has to be provided with knowledge about the problem as a whole and/or the particular problem's instance. We consider three sources of such knowledge:

- Using an optimizing SMT solver [26] to fill the 'holes' in the evolved programs so that the maximum possible number of tests is passed.
- Finding counterexamples that reveal the faults in the currently considered candidate programs (working solutions), i.e., inputs for which programs exhibit incorrect behavior.
- Training a machine learning model on a set of instances of a certain program synthesis problem in order to predict the features of the correct program given its specification.

While the first element on the above list fits into the established research area of *memetic algorithms* [140], the other elements are rarely seen in conjunction with genetic programming (nor in conjunction with many other heuristic search algorithms that could be conceivably used for program synthesis). Our goal is thus to investigate the ways in which genetic programming can be augmented by the aforementioned techniques so that its effectiveness for solving program synthesis problems increases.

In summary, the general goals of this thesis are as follows:

- designing heuristic-driven algorithms for the synthesis of *provably correct* programs,
- defining and exploring new types of tasks that can be approached with this 'apparatus',
- devising new notions of generalization for such tasks,
- assessing the usefulness of these ideas in realistic settings, including working with benchmarks that are rooted in well-known programming/physics/engineering problems, and examining the robustness of methods/solutions to noise.

1.3 Organization of the thesis

This dissertation is organized as follows.

Chapter 2 defines the problem of program synthesis and its most important characteristics. We describe the main paradigms of solving that problem, and provide several examples of practical applications.

Chapter 3 explains the importance and basic principles of formal verification of software. We focus on Satisfiability Modulo Theories (SMT), which is the formalism we employed in several approaches described in this thesis. The chapter concludes with the examples of successful industrial use of these techniques.

Chapter 4 is an introduction to evolutionary computation and genetic programming, and describes all crucial components of these algorithms, as well as some of their practical applications.

Chapter 5 presents Evolutionary Program Sketching (EPS), our approach to program synthesis by sketching. After description of the algorithm and its connection to memetic algorithms, we examine it on several benchmarks and compare it with "vanilla" genetic programming.

Chapter 6 presents Counterexample-Driven Genetic Programming (CDGP), our approach for using genetic programming to synthesize programs from formal specifications. After presenting the verification framework involved with this process and its integration with genetic programming, we investigate the effectiveness of CDGP on a suite of integer and text processing benchmarks.

Chapter 7 describes our extension of CDGP for symbolic regression problems, which we dubbed Counterexample-Driven Symbolic Regression (CDSR). In the experimental part, we compare various variants of CDSR with the state of the art machine learning regression algorithms.

Chapter 8 presents Neuro-Guided Genetic Programming, which employs an artificial neural network to prioritize genetic programming search. We describe the network's architecture, training, and how it assists the search. We also present the results of the computational experiments, in which we compare different variants of our approach with the baselines.

Chapter 9 summarizes this dissertation, and suggests future research directions.

Chapter 2

Program Synthesis

In this chapter, we define a problem of program synthesis and describe the main aspects that can vary between synthesis tasks: user intent (specification), and search space (programming language). We also present the main paradigms of solving synthesis problems, as well as the practical applications of this technology.

2.1 Introduction

In the 1950s, ACM editors described assemblers as "automatic programming systems" [171], and, according to Parnas [152], in the 1940s the term was even used for a rather unsophisticated automation of punching holes in paper tapes fed into computing machines. Parnas further writes: "In short, automatic programming always has been a euphemism for programming with a higher-level language than was then available to the programmer" [152, p. 1333]. In accordance with this logic, nowadays automatic programming, under the name of *program synthesis* that is used by most authors (e.g., [10, 11, 77, 105]), does not include assemblers and higher level programming languages. The goal has shifted to the generation of whole programs in these languages based on a specification of their expected behavior, and everything indicates that this understanding will remain in place for a foreseeable future.

2.2 What are programs?

The word "program" (from Greek πρόγραμμα, *programma*) originally meant a written public notice, but two additional meanings evolved over time: that of "written or printed list of pieces at a concert, playbill", and "a definite plan or scheme, method of operation or line of procedure prepared or announced beforehand" [7]. This last meaning of "program" is also the one being used in "linear programming"[1]. It is not hard to see how the modern meaning of the word related to computers evolved from it – at its core, a computer program is simply a prepared beforehand sequence of actions to be performed by a machine executing it.

[1] "To do this entails a look at the structure and state of the system, and at the objective to be fulfilled, in order to construct a statement of the actions to be performed, their timing, and their quantity (called a 'program' or 'schedule') [..]" [50, p. 1].

From the perspective of software engineer, a program is a sequence of instructions in a certain programming language which, when executed, produces the expected result aligned with requirements (if the program is correct[2]). Requirements are divided into *functional* (i.e., related to the outcomes that program produces) and *non-functional* (other characteristics, such as processing time, memory usage, stability, ease of maintenance). Generally, most of the work in program synthesis, and in this thesis too, focuses on the functional requirements, although there is also some research into satisfying non-functional requirements such as optimization of program's runtime [114, 131], or deobfuscation of software [87].

There are several approaches to modeling programs in order to investigate the limitations of computation, and the most widely known are *Turing machines* [189] and *lambda calculus* [43, 44]. These formalisms have enormous theoretical importance due to their simplicity, which still allows them to capture, as it is widely believed (Church-Turing thesis), all possible computable functions. Program synthesis research has, however, rather an "engineering" approach, i.e., it focuses on efficient solving of practical problems. As a result, if any reasoning about programs is performed (see, e.g., deductive approaches to synthesis in Section 2.5.3.3), it happens on the level of semantics of a particular programming language.

As shown by Turing [189], it is fundamentally impossible to create an algorithm that always determines in a finite time if a program halts or not – in other words, this problem is *undecidable*[3]. In program synthesis systems, non-terminating programs are often avoided by design (e.g., by proving that a given program halts, or constraining a programming language so that it comprises only terminating programs), and if not, then programs that exceed a given runtime limit are discarded (which may ultimately cause the synthesis algorithm to fail the synthesis task).

In this thesis, we limit ourselves to programs that:

- always halt,
- are functional in the sense that the produced output defines the entirety of the outcomes of computation (i.e., there are no side effects),
- have no loops or recursive calls,
- are deterministic.

Even under these assumptions, an efficient synthesis of such programs is far from being trivial. The main reasons for this are a large space of possible programs to search and complex interactions between program instructions (a small change in code can drastically impact the result).

[2] And a big part of software engineering literature is dedicated to avoiding such situations.

[3] Turing's proof of this fact is based on trying to verify (whether a program halts) a program that uses exactly that verifier to do the same but loops forever if the answer is positive. What if we required our halt verifier to only work for programs that do not try to verify that property themselves? Would such limited halting problem still be undecidable? An interesting question that is rarely being posed.

2.3 Definition of program synthesis

According to Gulwani [77, p. 3]:

> Program Synthesis (PS) is the task of automatically finding programs from the underlying programming language that satisfy user intent expressed in some form of constraints.

Krawiec [105, p. 3] expressed this definition more formally, which we present below together with a clear distinction between synthesis tasks and synthesis problems:

Definition 2.3.1. A **program synthesis task** is an ordered pair $(\mathcal{P}, Correct)$, where $\mathcal{P}$ is a programming language, and $Correct$ is a function $\mathcal{P} \to \mathbb{B}$ called *correctness predicate*. Solving the program synthesis task $(\mathcal{P}, Correct)$ consists in finding a program $p \in \mathcal{P}$ that satisfies $Correct$.

Definition 2.3.2. A **program synthesis problem** is a set of all possible program synthesis tasks for a fixed programming language $\mathcal{P}$ and a specific form of correctness predicate (see Section 2.5.1).

In other words, a *program synthesis task* is an instance of a *program synthesis problem*. When we design program synthesis algorithms (also called *program synthesizers*, or simply *synthesizers*), we want them to solve program synthesis problems, i.e., be able to handle every individual synthesis task that belongs to that problem. Synthesizers are also often specialized for a certain application domain, such as for example synthesis of text processing formulas, or symbolic regression. Since in a program synthesis problem the programming language is fixed, discovering and exploiting its characteristics is usually crucial for efficient program synthesis. In certain situations, one can even design languages in a way that facilitates the synthesis process (e.g., FlashFill [74], Rosette [188]).

Depending on the expressive power of a programming language $\mathcal{P}$, a program synthesis task can be either *solvable* or *unsolvable*. The latter means that no program $p \in \mathcal{P}$ satisfies the $Correct$ predicate, while the former means the opposite, i.e., that such programs exists. Efficient handling of unsolvable tasks may require detection of such situations in order to avoid unnecessary computations. In this thesis, we consider only solvable synthesis tasks.

Compilers, used routinely in contemporary software development, translate a program written in one language into some other, usually lower level, language, and could be on this ground considered to be program synthesis algorithms, with the $Correct$ predicate expressing the semantic correspondence between the synthesized and the original code. Despite this, they are not treated as such, because there is hardly any search involved in this process. Arguably, there are exceptions from this rule, for instance the optimization of code during compilation, and particularly superoptimization [131], which searches for an optimal semantically-equivalent code fragment and is not limited to being merely refactorization of the original code. Superoptimization is, however, universally agreed to be a program synthesis problem in its own right [77].

We can distinguish several variants of program synthesis problems based on the objective of computation [150]:

- **Search PS problem:** find a program $p \in \mathcal{P}$ such that $Correct(p)$ (Definition 2.3.2).

- **Decision PS problem:** determine, if there exists a program $p \in \mathcal{P}$ such that $Correct(p)$.
- **Optimization PS problem:** find a program $p \in \mathcal{P}$ such that $Correct(p)$ and the program p is optimal for a certain objective function (e.g., MSE on test cases).
- **Counting PS problem:** count the number of programs $p \in \mathcal{P}$ such that $Correct(p)$.

In this thesis, we present our contributions for solving the search PS problem (Sections 6, 8), and the optimization PS problem (Sections 5, 7).

2.4 Syntax-guided synthesis problem

Alur et al. [10] introduced the *syntax-guided synthesis* (SyGuS) problem, which formalizes, and to some extent constrains, the programming language $\mathcal{P}$ and the $Correct$ predicate from Definition 2.3.1.

Definition 2.4.1. The *syntax-guided synthesis* (SyGuS) task is a triplet (τ, ϕ, G), where τ is a background theory, ϕ is a logical formula expressing the $Correct$ predicate of a function f to be synthesized, and G is a context-free grammar specifying a set of allowed expressions. The solution of a SyGuS problem is an expression $e \in L$, where L is a set of all expressions over the grammar G, such that $\phi[f/e]$ (formula ϕ with all occurrences of f replaced with e, with appropriately substituted arguments) is true in the theory τ.

Definition 2.4.2. The *syntax-guided synthesis* (SyGuS) problem is a set of all possible SyGuS tasks.

A background theory τ is what provides meaning (semantics) to symbols. Background theories were introduced as an extension to Boolean satisfiability problem (SAT), leading to the concept of Satisfiability Modulo Theories (SMT) [20, 54] (for a more detailed discussion, see Section 3.2). For example, the theory of integers allows to use integer variables and constants in logical formulas, as well as arithmetical functions and relations (e.g., $+$, $-$, $\times$, $\div$, $>$, $<$, $=$). Often, the limited forms of theories, known as *logics*, are considered, which impose additional constraints on the allowed expressions; for example, Linear Integer Arithmetic (LIA) requires, among others, that multiplication can happen only between constants or a constant and a variable.

The logical formula ϕ describes the synthesis task as a relation between inputs of the function f being synthesized and its outputs (it is an example of *formal specification*, presented in Section 2.5.1). ϕ uses symbols from the background theory τ.

Finally, the grammar G is a syntactic template. It gives flexibility in specifying how the final solution is supposed to look like, potentially even including domain knowledge that a user may posses. Notice that since a programming language, defined by G, is part of the input, a SyGuS problem is not a program synthesis problem per Definition 2.3.2, which assumes that a programming language is fixed – instead, we could be tempted to call it a *meta-synthesis problem*.

Example 2.4.3. The `max2` program synthesis task (computing a maximum of two numbers) can be defined as a SyGuS task in the following way:

- τ: theory of integers, providing the standard mathematical interpretation of integer constants and operators ($+$, $-$, $>$, $<$, $=$, $\leq$, $\geq$, etc.). It is also commonly assumed that this theory includes the `ite` (if-then-else) operator.
- ϕ: $\forall_{x,y \in \mathbb{Z}}\ f(x,y) \geq x\ \wedge\ f(x,y) \geq y\ \wedge\ (f(x,y) = x\ \vee f(x,y) = y)$
- G (with the starting symbol `I`):

```
I ::= 1 | x | y | (+ I I) | (− I I) | (ite B I I)
B ::= (> I I) | (< I I) | (≥ I I) | (≤ I I) | (= I I) |
      (∧ B B) | (∨ B B) | (¬ B)
```

□

2.5 Dimensions of program synthesis

Gulwani [73] identified three "dimensions" of program synthesis, by which he meant different aspects influencing program synthesis problems and the techniques of solving them. Those are:

1. expression of user intent (specification),
2. space of programs over which to search (programming language),
3. search technique (synthesis algorithm).

2.5.1 Expression of user intent

The first dimension is user intent, which is a *specification* of the target program's expected behavior (the *Correct* predicate in Definition 2.3.1). It describes *what* a program is supposed to do, but not *how*. Discovering the latter is the main goal of program synthesis.

User intent may assume many different forms, most common of which are:

- **Input-output examples** – a set of pairs (x, y), in which x represents an input to a program, and y the target output that the program is supposed to return for x. This is arguably the most common form of user intent, and in a vast majority of cases it forces a synthesis algorithm to "guess" the expected program's behavior on the inputs not provided in the original set of input-output examples. In other words, this is where program synthesis meets machine learning in the task of generalizing beyond a provided training sample (inductive learning).
- **Formal specification** – a logical formula ϕ describing the relation between program's inputs and the expected output. The formula ϕ is usually universally quantified over variables representing inputs of a program. If a *precondition* is present, i.e., a condition specifying the inputs for which the behavior of a program is defined, then ϕ is an implication with the precondition as an antecedent, and the correctness condition, called *postcondition*, as a consequent. For example, a formal specification of a program (function) $f : \mathbb{R} \to \mathbb{R}$ that returns a value $f(x) > 0$ when $x > 100$, would be:

$$\forall_{x \in \mathbb{R}}\ x > 100 \implies f(x) > 0.$$

In general, formal specification enforces programs to satisfy a set of constraints, and as a special case the constraints can unambiguously define output for each input (*complete formal specification*). Contrary to the input-output examples, in program synthesis from formal specification it is rare to consider generalization of programs beyond the provided constraints.

- **Natural language** – programmers usually receive requirements (specification) of a project in the form of a description in natural language. Their knowledge of the language, together with the general intelligence and knowledge of computer science, makes it possible for them to understand the task and solve it. Currently, we are unable to create an artificial intelligence system with such capabilities. However, many patterns can be already discovered using natural language processing techniques [130]. A sufficient training set of language descriptions and their programmatic solutions can be used as a basis for a program synthesis system to learn an appropriate mapping, as was attempted for example in [25]. Another notable example of this form of user intent are natural language query interfaces to databases [12, 120, 153].
- **Demonstration** (traces) – *traces* are intermediate environment states resulting either from an execution of a (reference) program on some input, or a demonstration of computational steps. They are readily available, for example, as an effect of reverse engineering (or debugging) [116]. Traces are also crucial for *programming by demonstration* paradigm [49]. A recent example of such a system is the Rousillon Chrome extension [40], which records user's activity in the browser, and on that basis constructs a script in Helena [4], a high-level programming language for web automation, which then scrapes hierarchical and distributed data from the websites of interest.
- **Programs** – programs can also themselves constitute a specification of program synthesis problems. In fact, this scenario occurs naturally in several applications, such as: program optimization (e.g., superoptimization [131, 158]), deobfuscation [87], synthesis of program inverses [182]. Programs are also sometimes provided in conjunction with other forms of intent, for example in the *programming by sketching* paradigm [176, 174], in which the synthesizer generates code to fill the holes indicated in an incomplete program (sketch) by a programmer so that the unit tests are passed.

2.5.2 Space of programs

The second dimension is the search space of valid programs (a programming language $\mathcal{P}$ in Definition 2.3.1) that may constitute a solution of a given program synthesis task. This choice is made by the developer of the synthesis system, and in some synthesis systems it can be additionally restricted by a user to fit their needs.

The search space does not necessarily contain a program consistent with the user intent, i.e., the language may be too limited. Efficient handling of program synthesis tasks for which there is no solution in the given programming language is an additional complication, and it is a common practice that synthesizers are explicitly stated to not

be optimized for such situations – usually, they will run until exhaustion of the assigned computational budget. In this thesis, we also assume that a solution to the program synthesis task is always present in the search space.

When selecting a programming language for a synthesis system, it is necessary to ensure a good balance between expressiveness of the language and efficiency of search. On one hand, a synthesis system needs to be useful for a user, and on the other it needs to realize its task in a reasonable time. How programs are represented in the system can also play a big role in the overall efficiency of the synthesizer, and sometimes imposing additional constraints on the code's structure can be beneficial, while not reducing the expressive power of the language (e.g., not using `goto` statements in C++ programs).

An important subclass of search spaces in program synthesis problems are those of *loop-free programs*. Reasoning about such programs is much simpler, and it is often easy to guarantee that every program terminates. The synthesis of loop-free programs is, however, still challenging because the number of possible programs is exponential in the number of their components and program's length.

There are many features of programming languages that can influence the range of methods that can be used for the synthesis as well as the hardness of a synthesis problem, and we present below the most significant of them:

- restricted model of computation (e.g., regular expressions) / domain specific language (DSL) / a subset of a conventional programming language / conventional programming language,
- functional / imperative,
- single function / multiple functions / modules / entire applications,
- loop-free programs / programs with loops,
- programs that always terminate / programs that are not guaranteed to terminate.

Finally, we can briefly consider two interesting extreme cases of search space:

- *Every candidate program is also an instruction* – upon closer inspection, the distinction between a program and an instruction is fluid: usually both are functions, and often a program may consist of just a single instruction. We can thus imagine a theoretical program synthesis task, in which the set of instructions contains every valid program as a component to be used. While in such a scenario the shortest solution is always composed of just a single instruction, finding it may be harder than discovering a composition of many other program-instructions.
- *There is only a single instruction* – in this case, the programs are constructed from only a single instruction. An example may be the synthesis of logical functions using only NOR logical gate, which is sufficient for the construction of all possible logical functions.

2.5.3 Search technique

According to Gulwani [77], program synthesis techniques can be divided into enumerative, constraint solving, deductive, and stochastic/statistical. We will briefly present each

paradigm in the following sections.

2.5.3.1 Enumerative search

Enumerative approaches, as the name suggests, enumerate programs in a certain order, and for each individual program check if it satisfies the specification. These methods are typically exhaustive, meaning that eventually they are guaranteed to find a correct program, if it exists. Enumerative approaches work well in practice when the number of components for building programs is not too large. In particular, if the programming language contains constants that may assume many different values (e.g., integers), then enumerative algorithms, at least without assistance of other techniques, would not fare well due to the combinatorial explosion of possible values.

The main axes of improvement for enumerative approaches are:

- pruning semantically redundant programs,
- prioritizing the search, i.e., changing the order in which candidate programs are being visited, so that programs that are more likely requested by a synthesis task can be visited earlier,
- reusing already enumerated programs, either for construction of more complex programs, or by accumulating knowledge about the synthesis task.

Example uses of enumerative techniques for program synthesis are:

- EUSolver [11], which enumerates small expressions and then tries to perform, using decision tree learning algorithm [163], a *unification* into a single program with if-then-else branches. As such, EUSolver requires the if-then-else instruction present in the programming language, or a capability to simulate one by means of other instructions.
- Providing a baseline for other synthesis algorithms by a simple brute force enumeration assisted with some minor pruning to improve scalability, as was done by Katayama [92] for generation of all type-correct functional programs. He later extended this approach into a much more efficient MagicHaskeller system [93].
- Superoptimization of the machine code, for which enumerative search can give guarantees that the generated program is the shortest possible [131, 158].

2.5.3.2 Constraint solving

Approaches based on constraint solving reduce a program synthesis task to an instance of some other[4] constraint satisfaction problem (e.g., linear programming, SAT, SMT), for which an efficient specialized solver exists. Thus, approaches based on constraint solving work in two main steps [77]:

1. constraint generation,
2. constraint resolution.

[4]Program synthesis can be considered to be a constraint satisfaction problem.

In the constraint generation step, the program synthesis task specification and the space of possible programs are encoded in the form of constraints, so that values of free variables correspond to programs. During the constraint resolution phase, a specialized solver searches for such values of the free variables that lead to the satisfaction of the constraints. If the solver succeeds, then a program corresponding to the valuation of variables is extracted and returned as a final solution.

Example constraint solving approaches to program synthesis:

- SKETCH synthesis system [174, 176], designed for programming by sketching, in which the synthesis task was reduced to an instance of the SAT problem.
- *Oracle-guided component-based synthesis* [87], in which a linear loop-free program is constructed from several components. Constraint solving by reduction to the Satisfiability Modulo Theories (SMT) problem (Section 3.2) was used to find, on the basis of collected input-output examples, a new program and an input that distinguishes it from the previous program passing the examples.
- *Template-based synthesis* [183, 184], in which a synthesis task was reduced to an instance of SMT problem, and the templates (very similar to sketches in [174, 176], but more flexible) provided additional constraints on the structure of programs.

2.5.3.3 Deductive approaches

Deductive approaches to program synthesis try to construct a correct program purely by means of transformation of the provided task specification (they "deduce" the correct program from it). Typically, several rewrite/transformation rules are prepared beforehand by an expert, and applied depending on the circumstances. A common strategy is to decompose a synthesis task into several subtasks, and then combine their solutions (divide-and-conquer). The particular way of decomposing the task is often posed as a hypothesis, and if a contradiction is reached or there is no possible compatible program completion, then the algorithm backtracks. Deductive approaches require some degree of axiomatization of the programming language and data that it operates on, which limits their practical applications.

Example uses of deductive techniques for program synthesis are:

- Transformation rules for converting a formal specification into a program [127], possibly assisted with theorem proving techniques [128, 129], especially if loops and recursion are involved.
- Automatic transformation of mathematical expressions to improve accuracy of floating point operations [149]. In this particular work, inputs are randomly sampled to detect parts of the expression leading to significant rounding errors, and then a database of rewrite rules is searched to find rules which, when applied, would reduce the overall rounding error. The process continues until no further improvement is found.
- *Data-driven domain-specific deduction* [161], which employs deduction to reduce a synthesis task into smaller synthesis subtasks. This is done by using language-specific rules (provided by a user) for operator inverses, formalized by the authors under the

name of *witness functions* [161].

2.5.3.4 Stochastic/statistical techniques

These program synthesis algorithms are based either on stochastic metaheuristic approaches, or machine learning. In the former case, a program synthesis problem is usually converted to an optimization problem (typically with the number of passed input-output examples as an objective measure), and then solved with a heuristic search algorithm. In the latter case, the algorithm tries to learn a conditional probability distribution of programs given a specification, and then uses that information to efficiently find a correct program.

Example uses of these techniques for program synthesis are:

- The stochastic SyGuS solver [10], which uses Metropolis-Hastings algorithm [135, 80] to sample programs from a grammar.
- Approaches based on genetic programming [103], described in more detail in Section 4.2, where a population of programs is stochastically improved in an evolutionary process.
- Training a machine learning model to learn a conditional probability distribution that a certain instruction should occur in the solution given a user intent. An example of such an approach is DeepCoder [17], which first trains a neural network to predict instructions based on a (small) set of input-output examples, and then uses this information to support other search algorithms, such as depth-first search (DFS), which first traverses the branches associated with instructions with high probability.
- Approaches based on neural networks, which can be classified into:
  - Supporting other search techniques in the form of a machine learning model, an example of which is the aforementioned DeepCoder.
  - *Neural program induction*, where a neural network is itself a program (potentially equipped with additional modules, such as memory) that realizes the desired computation [72, 90, 108, 168].
  - *Neural program synthesis*, in which a neural network generates an interpretable symbolic program [117, 151]. One of the interesting applications of this technique is converting simple hand drawings into graphics programs written in a subset of LaTeX [62].

2.6 Applications

There are three main categories of applications of program synthesis:

- software development,
- automation of repetitive tasks for end-users,
- discovery of knowledge.

2.6.1 Software development

In our experience, the first reaction of the people not familiar with the topic of this thesis is that if our research succeeds, then the software would "write itself", and thus programmers would lose their jobs. In reality, however, until we construct machines more intelligent than us (in which case all people, not just programmers, would be in trouble), it will always be necessary to describe in detail to the synthesizer what an application is supposed to do. This is a highly nontrivial task for most real world applications, and it is easier and cheaper to hire programmers to write the application, rather than carefully consider all edge cases to express in some formal language what the code is supposed to do. And then there is also the time it would take the synthesizer to synthesize a program that meets that specification. For these reasons, virtually all practical applications of program synthesis in software development aim for a more limited goal of assisting programmers in their job and synthesizing small code fragments.

Program synthesis can be used in software development in the following ways:

- Generating code fragments or functions correct with respect to the provided specification [11, 32, 160, 161].
- Filling in the omitted computational details in the general structure of code provided by a user – the process known as *sketching* [174, 176]. In this paradigm, a user of a synthesis system explicitly leaves "holes" in their code (i.e., placeholders for some yet unknown content), and then provides a specification in the form of input-output examples, unit tests, or logical formulas, and the synthesis system searches for a piece of code that, when put into the hole, would meet that specification.
- Fixing bugs in code [67, 102, 193].
- Improvement of an inefficient code [114, 156], including superoptimization [131, 158]. In these problems, synthesizers search for a semantically equivalent program that is more efficient.
- Generating program inverses [182]. In such a case, the original program itself constitutes the specification.

Another research field similarly focused on automation of programmer's work is *search-based software engineering* (SBSE) [78, 79], which uses various search and optimization techniques in order to facilitate the development of software. Some of the uses of SBSE include: finding the minimal number of unit tests covering all program branches, modularization of an existing code base, finding an optimal sequence of refactorings, or project planning.

2.6.2 Automation of repetitive tasks for end-users

There are many situations in which end-users of computer systems need to do certain repeatable tasks. Unfortunately, most of end-users cannot program, and thus have no effective tools for dealing with such problems. One of the solutions is using a program synthesis system specialized for a particular application, which will be able to generate a program from a simple specification provided by the end-user. Input-output examples

are a very intuitive form of specification, in contrast to logical formulas. *Programming by demonstration* [49, 115] also seems to be a good choice for end-users without programming experience, as well as natural language interfaces.

Example applications of program synthesis for automating repetitive tasks and enabling programming for end-users of computer systems are:

- Automatic inferring of text processing formulas from input-output examples, for example the FlashFill system available in the Microsoft Excel spreadsheet software [74, 75].
- Automatic generation of structured drawings from a small example of the pattern [42].
- Natural language query interfaces to databases [12, 120, 153].
- Automatic collection of structured data from websites based on demonstration [40].

2.6.3 Discovery of knowledge

Program synthesis can be also used for knowledge discovery, especially in the domains where phenomena are complex and there are many interdependent variables. Examples of this application of program synthesis include:

- Symbolic regression, which is the problem of discovering a symbolic formula expressing a relationship between variables [173, 190, 191, 192].
- Synthesis of programs for quantum computers [132, 133, 179, 180].
- Discovery of new classical algorithms, for example bitvector algorithms [76, 87], mutual exclusion algorithms [94, 95, 96, 97], and potentially even machine learning algorithms [166].

Chapter 3

Formal verification of programs

In this chapter, we will describe the techniques for proving that a program is correct with respect to its specification. After explaining the motivation for using formal verification, we will show how to formally verify correctness of programs using SAT/SMT constraint solving, an approach we used in several algorithms presented in this thesis. We conclude with some successful applications of formal verification in real-world scenarios.

3.1 Introduction

During the High-Assurance Cyber-Military Systems (HACMS) program [65], carried out by DARPA in years 2012–2021, it was demonstrated that attackers can remotely take control not only of an open source quadcopter, but also of the proprietary Boeing's Unmanned Little Bird (ULB) helicopter. This spectacular demonstration was not entirely surprising, since many cases of insecure computer systems embedded in devices were reported earlier, such as the possibility of remotely taking control over a car [41], or wireless hacking of an insulin pump to make it deliver an incorrect dosage of medicine [164]. HACMS program, however, went further in that the researchers made a serious attempt at creating a hacker-proof system by means of formal verification, and the "read team" responsible for hacking did not manage to compromise the improved versions of quadcopter and helicopter, despite having full knowledge of their software and hardware, and taking part in the development process. This experiment reportedly convinced DARPA that formal verification is a technology which can be applied to real-world systems and is worth investing in.

Nowadays, many software systems are so complex that imagining all possible usage scenarios, especially when a system is concurrent and distributed, is a considerable challenge for a human mind. This has led in the past to several costly accidents due to faults in software, such as the Ariane 5 Flight 501 failure caused by a wrong float conversion [118], or the Pentium FDIV bug caused by a subtle error in the microchip's implementation of the SRT division algorithm [60]. Some kind of assurances about program's reliability are thus required if we are to trust software that we use, and they come mainly in two forms: *unit tests*, and *formal verification*. Unit tests, which are pieces of code that confront program's output or state with the one that should be reached in a given situation, are considered a standard programming practice. They are usually easy to create and modify

when the project's requirements change, as happens in almost every software project in the industry. On the other hand, they are limited by programmer's imagination and usually are far from covering all possible situations, and thus the guarantees they provide are only partial.

The guarantees provided by formal verification are much stronger, because its goal is constructing a mathematical proof that a given program has certain properties. This uncompromising nature of formal verification makes it suitable for creating hacker-proof, and more generally fault-proof, computer systems. Notably, during this process there is no need to directly execute a program, and thus formal verification belongs to the family of techniques called *static program analysis* (in contrast, unit tests are an example of *dynamic program analysis*). Due to improvements in processing power and algorithms (e.g., SAT solving), this technology became feasible for applications in industry. However, lunches are seldom free, and formal verification involves a considerable burden of:

- writing a *formal specification*, a precise high-level model of what software is expected to do, which is a nontrivial task,
- modeling the programming language so that instructions' semantics can be used during deduction process (this step can be automated by existing tools),
- constructing a proof that the piece of software is correct, which may take considerable time depending on its complexity,
- modifying formal specification as project's requirements change.

In conclusion, the prevailing opinion is that the costs of formal verification are currently justifiable only for safety-critical and business-critical systems.

There is a popular science aphorism that "all models are wrong, some are useful", and it applies very well to formal verification. In order to prove properties of real-world computer systems, we need to create their models, and the essence of every model is abstraction (i.e., simplification by removing unimportant details). This means that all security guarantees are provided under certain assumptions ("fine print"), and one should not be lulled into a false sense of total security just because the software was proven correct. For example, typically hardware is assumed to be without faults, and software used for the construction of proof is assumed to be without bugs. There is also always the possibility that the specification itself happens to ignore certain cases of program's behavior, since the human element involved in its creation is fallible. Despite all this, if done well, formal verification allows one to design a system with as high level of security as is realistically possible.

3.2 SAT/SMT problems

One of the approaches to the task of formal verification is reducing it to some other algorithmic problem, preferably a one for which efficient solvers already exist. The *Satisfiability Modulo Theories* (SMT) problem [20, 22, 54], which is a generalization of the Boolean satisfiability (SAT) problem, is often used for formal verification because it is relatively simple and yet expressive enough for formal specifications. What is more, since it is an extension of the SAT problem, it taps into the long line of research on exact and

Table 3.1: Examples of logical formulas and their models for propositional logic.

Formula	Is satisfiable?	Model
a	yes	$a = true$
$\neg a \ \vee \ b$	yes	$a = false, \ b = true$
$a \ \wedge \ \neg a \ \wedge \ b$	no	—

Table 3.2: Examples of logical formulas and their models for first order logic extended with theories (SMT), where $x, y \in \mathbb{Z}$, $a \in \{false, true\}$, and s_1, s_2 are text strings over the English alphabet.

Formula	Is satisfiable?	Model
$(10 \cdot x = 20) \ \wedge \ a$	yes	$x = 2, \ a = true$
$\forall_{x,y} \ (x + y)^2 > x^2 + y^2$	no	—
$\texttt{str.len}(s_1 \ \texttt{++} \ s_2) = \texttt{str.len}(s_1) + \texttt{str.len}(s_2)$	yes	$s_1 = \texttt{"a"}, s_2 = \texttt{"a"}$

heuristic algorithms for solving that problem.

Definition 3.2.1. *Boolean satisfiability (SAT) problem* is the problem of determining for a given propositional formula ϕ if there exists such an interpretation of Boolean variables that will satisfy ϕ (i.e., values of variables for which ϕ will be true). Example SAT formulas are presented in Table 3.2.

Definition 3.2.2. *Satisfiability modulo theories (SMT)* problem is the problem of determining for a given first-order logical formula ϕ containing constants, operators, and variables of types defined within a set $\mathcal{T}$ of *theories* (because often several theories are used in conjunction), if there exists such an interpretation of variables that satisfies ϕ. Each theory $\tau \in \mathcal{T}$ defines a set of available symbols (constants, operators) and their semantics, and specifies the kind of expressions that can be created with them. Example SMT formulas are presented in Table 3.2.

There are several software implementations of SMT solvers, the most popular of which are Z3 [53] and CVC4 [18]. All contemporary SMT solvers accept queries written in the *SMT-LIB language* [19, 21]. A good compilation of SAT/SMT queries expressed in SMT-LIB for solving various problems can be found in [200].

3.3 Formal verification of programs

Formal verification of a program p is the task of proving that the following logical formula is true:

$$\forall_{in} \ Pre(in) \implies Post(in, p(in)), \tag{3.1}$$

where in is program's input, Pre is a logical formula called *precondition* which expresses the initial conditions imposed on the program input in for which a program is supposed to work (e.g., in TSP distances between cities are required to be greater than zero), and $Post$ is a logical formula called *postcondition* which describes the expected properties of the program's output (e.g., for a program that is meant to calculate the maximum of its

two numerical arguments, returning a bigger of the two numbers). Proving that the above formula is true is often equivalently approached by changing it to:

$$\exists_{in}\ Pre(in) \implies Post(in, p(in)), \tag{3.2}$$

where we search for an input in which disproves the formula, and if we are unable to find it, then program is correct.

The semantics of a program is typically expressed using Hoare logic [84], in which we can specify for each instruction, or more generally a sequence of instructions, the conditions satisfied before and after its execution. This can be written as:

$$\{P\}\ C\ \{Q\},$$

where C is a sequence of instructions, and P and Q are conditions on the values of program's variables (program's states) before and after the execution of C, respectively. An expression $\{P\}\ C\ \{Q\}$, known as the *partial correctness specification*, is true when a program C executed in a state satisfying P (*precondition*) either produces a state satisfying Q, or does not terminate; for the *total correctness specification*, we need to additionally enforce that a program always terminates. Hoare additionally provided a set of axioms for combining sequences of instructions and their conditions, and in this way we can transform the whole program into a logical formula specifying its behavior. The only thing that remains is to check if this formula is subsumed by the program's specification, and Equation 3.1 is one of the ways to achieve this.

Formal verification in the presence of loops and recurrence is more challenging and involves either bounded verification for a finite number of loop iterations (and thus gives incomplete guarantees), or finding loop invariants and using them to prove that a loop will eventually terminate and what are the possible program's states when that happens. In this thesis, however, we do not consider programs with loops and recurrence, and thus we omit that material.

Example 3.3.1. We will now show how to formally verify a program for solving a non-negative[1] variant of the `max2` problem, in which it is expected to return a maximum of two non-negative integers x and y. We will use SMT as the formalism for describing a program and its specification. We will verify the following program p written in Python:

```
1 def max2(x,y):
2     if x < y:
3         return x
4     else:
5         return y
```

This program is incorrect, since the greater-than operator should be reversed. The semantics of this program can be represented in SMT (Section 3.2) with theory of integers

[1] Non-negativity is introduced here only so that the precondition is not empty (true).

as:

$$\begin{aligned} x < y &\implies out = x \quad \wedge \\ x > y &\implies out = y \end{aligned}$$

where $out = p(x, y)$. This actually is an idealized (mathematical) model of p, in which integers can be of arbitrary size; for real-world applications it would be in general necessary to represent integers as bit vectors or otherwise safeguard against overflows.

The formal specification of our non-negative `max2` problem consists of precondition Pre and postcondition $Post$:

$$\begin{aligned} Pre(x, y) &\equiv x \geq 0 \ \wedge \ y \geq 0 \\ Post(x, y, out) &\equiv out \geq x \ \wedge \ out \geq y \ \wedge \ (out = x \ \vee \ out = y) \end{aligned}$$

The full verification formula is thus:

$$\begin{aligned} \exists_{x,y \in \mathbb{Z}} \ x \geq 0 \ \wedge \ y \geq 0 \implies & x < y \implies out = x \quad \wedge \\ & x > y \implies out = y \quad \wedge \\ & out \geq x \ \wedge \ out \geq y \ \wedge \ (out = x \ \vee \ out = y) \end{aligned}$$

The Z3 SMT solver applied to this program and specification produces a model $\{x = 0, \ y = 1, \ out = 0\}$ illustrating an incorrect run of the program. Once the greater-than operator is reversed, no model can be found and thus the program is correct. □

3.4 Model checking

While not directly used in our work presented in this thesis, *model checking* [15, 45, 64] is another family of formal verification techniques which are worth describing in more detail. Model checking approaches the problem of formal verification by conducting an exhaustive search through the graph of program's state space, where state is defined by a certain combination of values of the program's variables. A consequence of this is that model checking is limited to programs that can assume only a finite number of states. In the real world practice, however, all programs have finite number of possible states due to finite memory, so this is not such a severe limitation as it may seem at first. It is rather the exhaustive nature of search that limits, despite impressive efficiency achieved by model checking algorithms [36], practical uses of this technique. For the specification of program's correct behavior, model checking employs various temporal logics [63] to express allowed/expected states of the system in time (e.g., it may be verified that a certain state will be always eventually reached).

3.5 Applications

The successful applications of formal verification include:

- The mentioned in the introduction HACMS program and creation of a more secure quadcopter and helicopter, which resisted hacking attempts by a team with full access to the source code [65].
- CompCert [1, 119], the first formally verified compiler of a big subset of the C programming language, which provides high guarantees that a compiled code will behave in accordance with the semantics of the source C program. In their study on robustness of compilers, Yang et al. [197] managed to discover some bugs in CompCert (the bugs were either in the yet unverified front-end code of the development version of CompCert, or were due to the missing constraints in the specification; all bugs were fixed in the later versions), but it was still a small percentage compared to the number of bugs that was found in other C compilers.
- Amazon [143] uses model checking for verification of high-level abstract models of their critical systems, i.e., verified is the design rather than the implementation. For this task, they use the TLA+ specification language [113] and the TLC model checker [199]. According to the authors [143], using these systems allowed Amazon to avoid a range of subtle but serious problems in their applications.
- Facebook maintains the Infer [39] static program analysis tool, and uses it to verify certain properties of its mobile applications. Infer is based on separation logic [170] (an extension of the Hoare logic that allows reasoning about heap structures and pointer manipulation), and supports several programming languages, such as Java and C. It is a support tool for detecting several types of common errors (like null pointer exceptions, resource leaks, or race conditions) and does not require a formal specification to be provided by a user.
- seL4 [101] is a fully verified microkernel (a minimal functional core of an operating system that is the only software that needs to run in a privileged mode) of around 10,000 lines of C. For verification was used Isabelle/HOL framework [145], which is an interactive theorem prover in which part of the burden of conducting a proof lies on a user.
- Formal verification was also employed to verify a neural network [98] which was used in Airborne Collision Avoidance System (ACAS) to reduce the memory usage (neural network effectively compressed a huge table of parameter values and assigned correct responses). Various properties of the network were established, mostly of the kind that for a certain range of parameters the network would never return certain values (e.g., a “strong” danger message if another plane is on a collision course but is still far away).

In the future, we can expect even more successful verification projects and more secure software due to the further improvements in processing units and algorithms. Recently, Elon Musk demonstrated progress of his startup Neuralink on the design of brain-computer interfaces [142], and we definitely would like to have at least some level of certainty that a hacker attack or a sudden bug will not happen on a chip inside one’s head.

Chapter 4

Evolutionary computation for program synthesis

The purpose of this chapter is to introduce heuristic evolutionary approaches to the synthesis of computer programs. After the introduction of evolutionary algorithms and their principles of work and applications, we proceed to genetic programming, which is the main algorithm used in the rest of the thesis.

4.1 Evolutionary algorithms

While it is debated by philosophers whether art imitates life or the other way around, we can certainly say that science imitates life, at least sometimes[1]. The two most influential natural inspirations for artificial intelligence are: brains, in which a huge number of relatively simple cells connected with each other are able to adapt to the changing environment and solve hard problems relevant to the organism's survival (such as writing this thesis), and the process of natural evolution [51], which spawned all the diverse life we can observe, as well as the brains. Given the observed performance of these natural phenomena, it is unsurprising that scientists try to imitate them, although there are also cases when imitation of nature, especially if done on a superficial level, is taken too far [177].

Evolutionary computation is the area of research that tries to harness the power of evolution to achieve various goals of practical relevance in, e.g., engineering. Evolution itself might be compared to diffusion in physics, because it is simply a phenomenon that statistically happens when some capable of reproduction individuals are preferred over others (selection pressure), and the reproduction does not always produce perfect copies of the parents. Evolutionary computation uses this insight to design artificial evolution scenarios, in which a carefully devised selection pressure leads a population of individuals, representing solutions to some problem, to their desired optimal form. Although the natural evolution works in a time scale of millions of years, computers allow to simulate these simplified evolutionary processes very quickly. An algorithm realizing the process of artificial evolution is called *evolutionary algorithm* (EA), and formally it is a heuristic iterative search algorithm that maintains a working set (population) of candidate solutions that are stochastically selected and modified according to their quality (fitness).

[1] Albeit no one would exclaim that all science is quite useless; only some of it.

4.1.1 Brief history

The field of evolutionary computation originates in the three streams of research that until 1990s were conducted largely independently:

- **evolutionary programming**, first described in 1966 by Fogel [66]. It was also historically the first approach. Characteristic features include: pooling together children and parents before final selection into a new population, use of relative fitness, and mutation being the only variation operator. The representation of solutions was problem-dependent.
- **evolutionary strategies**, introduced by Rechenberg in 1973 [167]. Evolutionary strategies were designed for the optimization of real functions of real variables, and thus individuals are represented as vectors of real numbers. Both crossover and mutation are used, and the parameters of the latter are stored in the genome and evolve together with the genes of a solution.
- **genetic algorithms**, proposed by Holland in 1975 [85]. In this approach, a typical representation is a sequence of bits, and crossover exchanges bit strings between parents, while mutation flips each bit with a certain probability.

In the 1990s, the fourth **genetic programming** (GP) stream appeared, popularized by John Koza [103], although it is worth mentioning that it was Nichael Cramer [48] who first used the tree representation and tree variation operators for the evolutionary synthesis of computer programs. Around the time GP was introduced, several annual and biannual conferences on evolutionary computation were established, and it was universally agreed that, given their similarities, all four streams are different subareas of a wider field of evolutionary computation.

4.1.2 Genetic representation of individuals

In natural evolution, there is the distinction between *phenotype*, i.e., observable features of an organism, and *genotype*, which is the encoded representation of phenotype in the form of inheritable (by offspring) units called *genes*. Contrary to some early speculations by Lamarck [111, 112], based on the idea that organs not being used diminish during organism's life and this effect is inherited by its offspring, we now know that the phenotype is, for the most part, not inherited directly by offspring[2]. However, adaptive changes in behavior during lifetime can, by means of natural selection, influence species' genotype indirectly in the long run (the so called *Baldwin effect* [16]).

Since phenotype is created on the basis of information encoded in genotype, it makes sense to introduce the notion of *genotype-phenotype mapping*, which determines how individual genes influence the phenotype. The correspondence between genes and phenotypic traits is often not one-to-one: a single phenotypic trait can be affected by multiple genes, and one gene can affect many phenotypic traits. There is also nothing that prevents many different genotypes to map onto the same phenotype.

[2]Some information, however, indeed is inherited by means different than genes, and the research field of epigenetics studies these processes.

A *representation* of solutions in evolutionary algorithms is basically a way of encoding (genotype) a particular candidate solution (phenotype) to the computational problem of interest. In order to perform evolutionary search over the *search space* of all candidate solutions to the problem, we must first encode them in a form that is accepted by some variation operators (Section 4.1.9), and define a genotype-phenotype mapping for a fitness function to grade the quality of solutions (Section 4.1.6). For example, assume that the task is to find a placement of eight queens on a chessboard such that no two queens check each other (the Eight-Queens Problem). There are many possible representations for this problem. For example, positions of queens may be encoded as a set of 8 pairs of integers representing coordinates on the chessboard. This representation is trivial in the sense that it can be practically directly decoded to the original phenotype (i.e., the positioning of queens on the chessboard).

The choice of representation can be essential for the efficiency of search, and some prior knowledge about the form of a correct solution can also be taken into account. For example, we could observe that in the Eight-Queens Problem, queens can never occupy the same row or column, and since there are as many rows and columns as queens to place, we can evolve only a vector of vertical coordinates of queens (i.e., horizontal coordinates are fixed and not included in the genotype), instead of their absolute coordinates on the chessboard, thus reducing the search space.

4.1.3 Population of candidate solutions

A characteristic feature of evolution, both natural and artificial, is that it is a process which happens for a certain collection of individuals, which we call *population*. Formally, the population is a multiset, because in EA we usually allow redundancy. The size of a population is usually kept constant during the algorithm's run, but it is a choice motivated by tradition or simplicity rather than considerations regarding algorithm's effectiveness.

The benefit of keeping many competing candidate solutions in a population is the resulting parallel search. For comparison, hill climbing local search algorithms [125] iteratively modify a single solution, and sooner or later necessarily end up in some local optimum. Iterative local search [125], in which a local search algorithm is run multiple times for different starting points, partially mitigates this problem and may potentially reach global optimum given enough time. However, the subsequent runs gain no knowledge from the previous ones, and the redundancy of search increases with time. Evolutionary algorithms represent a different approach to this problem, because the initial population is scattered across the whole search space, and, at least in theory, as the time goes the algorithm accumulates knowledge regarding the good regions of search.

4.1.4 General workflow of evolutionary algorithms

We will employ here a top-down explanation strategy, meaning that we will first present the general outline of how different modules constituting evolutionary algorithm interact with each other, and then, in later sections, we will describe them in more detail. To a significant extent, the modules mirror the mechanisms of natural evolution, so we hope

Algorithm 4.1: The general scheme of evolutionary algorithms.

```
1: function EvolutionaryAlgorithm
2:     P ← InitializePopulation()
3:     Evaluate(P)
4:     while ¬TerminationCondition(P) do
5:         parents ← Selection(P)
6:         children ← Variation(parents)
7:         P ← ReplacePopulation(P, parents, children)
8:         Evaluate(P)
9:     return arg max_{p∈P} fitness(p)
```

that after reading this section the reader should have, by means of analogy, the general idea of what happens in the algorithm.

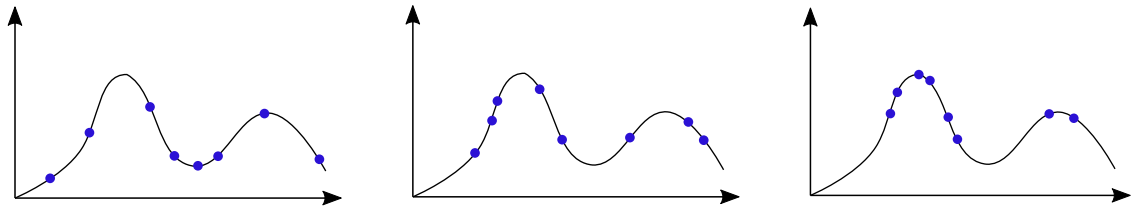
The general scheme of evolutionary algorithms is presented in Algorithm 4.1. Evolutionary algorithms start with the *initialization* of the population P (Section 4.1.5), followed by the *evaluation* (Section 4.1.6) of quality of the generated solutions. Then, until a *termination condition* (Section 4.1.7) is met, in the main loop (also called *evolutionary loop*) of evolutionary algorithm we repeatedly *select parents for reproduction* (Section 4.1.8) and apply *variation operators* (Section 4.1.9) to create offspring (named `children` in Algorithm 4.1). Then, the current population is *replaced* (Section 4.1.10) with the new one, and before the next generation (i.e., iteration of the main loop) the non-evaluated individuals in the newly created population are evaluated. If the termination condition is satisfied at some point, either by finding an optimal solution or exhausting the assumed computational budget (primarily time), then the evolutionary search is terminated and the individual with the best fitness in the population is returned.

4.1.5 Population initialization

The first step of every evolutionary algorithm is *initialization*, i.e., filling the population with some initial individuals that form the starting points for the search process. During initialization, individuals are usually generated randomly, and the exact implementation of this process depends on the selected genetic representation. In situations when it is possible, using domain-knowledge to seed the initial population with prospective individuals can have some advantages [61, 173]: preventing waste of computational effort by "reinventing the wheel", and biasing the search to the regions of search space containing good solutions (at the risk, however, of a decreased diversity of the population and leading to the search being stuck in a local optimum).

4.1.6 Fitness function

A *fitness function* is a function that evaluates the quality of a candidate solution. The value it produces, usually a scalar, is called *fitness* (or *fitness value*; in case of a vector, it would be called *fitness vector*), and is primarily used for the selection of parents for reproduction (Section 4.1.8). From the perspective of classical optimization, a fitness function is equivalent to an objective function which is to be maximized/minimized, i.e.,



(a) Generation 1. **(b)** Generation 2. **(c)** Generation 3.

Figure 4.2: A visualization of an evolutionary algorithm's run as a population of individuals traversing the fitness landscape of a problem. On the horizontal axis is presented the search space of genotypes, and on the vertical axis a corresponding value of fitness.

we search for such a genotype p^* that:

$$p^* = \arg\max_{p \in P} \text{fitness}(p).$$

As mentioned earlier, a fitness value is assigned to the phenotype, so we need to first employ genotype-phenotype mapping on the genotype.

By presenting a space of possible genotypes on the horizontal axis (or axes, depending on the assumed structure of the search space), and fitness values on the vertical axis, we arrive at the idea of *adaptive landscapes* (or *fitness landscapes*). Historically, the first to apply this technique for studying natural evolution was Sewall Wright in 1932 [196]. Using this approach, in Figure 4.2 we will illustrate how an evolutionary algorithm navigates the (genotypic) search space. Initially, in Figure 4.2a points (representing candidate solutions) are distributed randomly during initialization (Section 4.1.5). In the next generation (Figure 4.2b), their distribution changes, but since well-performing solutions were prioritized in the parent selection, their children shift slightly towards local optima. Finally, in Figure 4.2c, we see that a candidate solution with fitness close to the globally optimal value was found.

4.1.7 Termination Condition

Evolutionary algorithms are *anytime algorithms*, meaning that as the time progresses they generate better and better solutions to the problem, and thus if stopped at *any time*, some solution can be returned. As a consequence, the *termination condition* of algorithm's evolutionary loop can be adaptive or external to the algorithm itself. For example, termination condition can be based on:

- Lack of improvement in a certain time interval (stagnation).
- Exhaustion of wall-clock or processor time.
- Number of evaluated solutions (for a constant population size, it is equivalent to specifying the number of generations).

Often, some combination of these conditions (and other non-mentioned) is used – in such cases, triggering even one condition is sufficient to terminate the search.

In some applications, it is possible to determine that the optimal solution was found, for example in the Eight-Queens Problem mentioned before. In such cases, the optimality

of individuals in the population is checked in termination condition, and the search is terminated when such optimal individual is found.

4.1.8 Parent selection

Parent selection algorithm (also called *selection algorithm*, or just *selection*) is responsible for gradual improvement of the population by applying *selection pressure* to individuals. There are two contrary objectives that need to be considered:

- **exploration**, that is examining solutions from many different (and potentially yet unexplored) locations in the fitness landscape to discover new promising regions of search space.
- **exploitation**, that is trying to find better solutions in the already discovered promising regions of search space.

In order to avoid too strong bias towards exploitation, and thus potentially *premature convergence* to some local optimum, a parent selection algorithm needs to allow weaker solutions to reproduce too. There are many different types of parent selection algorithms, which vary in how they balance exploitation and exploration. For example, in *proportional selection* individuals are selected with the probability equal to the ratio of their fitness to the sum of fitness of all individuals, and in *rank selection* the ordinal ranks are used in the similar way. Nowadays, the most popular selection technique is *tournament selection*, and in applications related to program synthesis the *lexicase selection* [178] gains popularity due to its effectiveness. We will describe both of these algorithms in more detail.

4.1.8.1 Tournament selection

To select a single parent in tournament selection, we draw randomly (without replacement) k individuals from the population, and return the best of those k solutions. Note that this gives a good chance of reproduction to individuals with weaker fitness, and thus facilitates exploration. The value of k determines the strength of the selection pressure, with low values (e.g., $k = 2$) being much more lenient than larger values (e.g., $k = 7$). The appropriate value of k needs to be determined experimentally, because there is no universal method for adjusting it to a particular problem.

4.1.8.2 Lexicase selection

Lexicase selection [178] is a parent selection method, devised originally with modal problems in mind, i.e., problems for which the behavior of a target function changes significantly in different regions of the problem's domain. Lexicase selection is multiobjective by nature [109], and as such requires a fitness vector to work on, and each element of this vector is treated as a criterion to be optimized. We will call the positions (indexes) of elements in this vector *tests*; for example, if we have a fitness vector $x = [4, 2, 5]$ of some solution s, then the value obtained by s on the first test is 4, on the second 2, etc.

The pseudocode of lexicase selection is presented in Algorithm 4.3. The algorithm always finishes in the one of two states:

- Only one solution remains (lines 2-3), and it is returned.

Algorithm 4.3: The general scheme of the lexicase selection algorithm.

```
1:  function LEXICASE(tests, solutions)
2:      if |solutions| = 1 then                          ▷ Only one solution remains
3:          return solutions[0]                          ▷ Return the solution
4:      else if tests = ∅ then                           ▷ All tests were used
5:          return solutions[random(0, |solutions|)]     ▷ Return a random solution
6:      else
7:          t = RANDOMTEST(tests)                        ▷ Select a random test
8:          tests = tests \ {t}                          ▷ Remove it from tests
9:          solutions = arg max_{s∈solutions} EVAL(s, t) ▷ Leave only the best solutions on t
10:         return LEXICASE(tests, solutions)            ▷ Solve the smaller problem
```

- All tests were used (lines 4-5), and a random remaining solution is returned.

If none of these conditions is met, then a test t is selected randomly (line 7) and is removed from the remaining set of tests (line 8), and all solutions not having the best value on t are filtered out (line 9). Finally, the lexicase selection is applied once again to the remaining tests and solutions (line 10). This algorithm is guaranteed to terminate, since the number of tests decreases in each iteration.

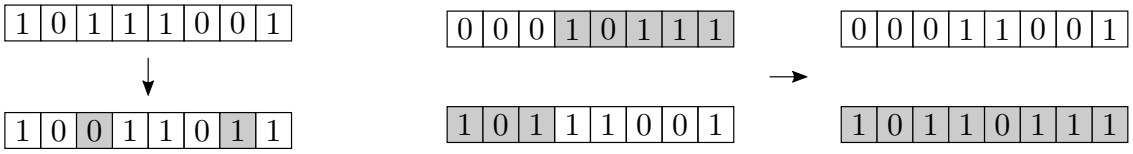
The name of this method originates from the randomly generated lexicographic ordering[3] of tests that is used in each application of lexicase selection. The tests act as filters applied sequentially in this order, and only the best performing solutions on the current test pass through to the next iteration. Sooner or later, only a single solution (or, in some cases, a subset of solutions) will remain, and it will become a parent. A consequence of sequential application of random tests is that the solutions performing well on tests that are hard for the rest of the population are rewarded and selected, even if they produce poor results on all other tests. Thus, lexicase selection implicitly attempts to create populations containing the entire pareto-front of solutions on tests.

4.1.9 Variation Operators

In local search optimization algorithms [125], a new solution to visit is selected based on the explicitly provided neighborhood relation on the search space. In evolutionary algorithms, this role is assumed (implicitly) by variation operators. These operators, as the name suggests, introduce a change to the selected parents (Section 4.1.8) so that new, and hopefully better, candidate solutions will be added to the new population. Contrary to the local search algorithms, after the perturbation is done it is not checked, if the child is better than the parent, and therefore EAs allow the quality of solutions to "locally deteriorate". This is a deliberate decision, because EA are *global optimization methods* [125], i.e., if we run them long enough, they have a chance to eventually find a global optimum independently of the starting point, and to do that they need a capability to escape the local optima.

The two standard variation operators, used in most of the works on EA, are:

[3] In the pseudocode (Algorithm 4.3), a random test is selected in each iteration, but an equivalent implementation would randomly shuffle tests once at the beginning, and then always select the first test in line 7.



(a) Mutation. **(b)** Crossover.

Figure 4.4: The examples of variation operators for a binary representation of solutions.

- **Mutation**, which takes as an argument one parent individual, and randomly changes a small fragment of its genotype to produce one offspring. An example of mutation for a binary representation of solutions is presented in Figure 4.4a.
- **Crossover**, which takes two parents and exchanges fragments of their genotypes, resulting in two offspring individuals. Often, only one of them is returned as a child. An example of crossover for a binary representation of solutions is presented in Figure 4.4b.

Typically, the variation operators are applied (with certain probability) independently of each other, meaning that, e.g., after crossover a solution can be additionally mutated.

4.1.10 Population replacement

Population replacement (also called *survivor selection*) [61] is a method of creating a new population on the basis of the current population, the parents, and the created offspring. We can distinguish the following main approaches:

Generational EA The current population is discarded, and all created offspring constitute the new population.

Steady-state EA [194] Only one offspring is generated per generation, and from the current population is removed the weakest or random solution so that the population size remains constant.

Many other replacement methods are possible, which mainly vary in the proportions of parents and offspring; for example, in the $(\mu + \lambda)$ variant of evolutionary strategies the new population consists of μ parents selected from the current population, and λ offspring created from them.

4.1.11 Applications

Evolutionary algorithms find numerous applications in many different fields, for example[4]:

- Engineering: design of vibration-resistant antenna boom for spacecraft [99], design of lens systems [24, 187], design of electronic circuits [123].
- Program synthesis/machine learning: evolving wavelet for better image compression [71], ellipse detection that is more robust than conventional methods [198].

[4] Applications of genetic programming are excluded from this list, because we dedicate the entire Section 4.2 to that paradigm.

- Construction of phylogenetic trees [46, 47].
- Timetabling [38, 148].
- Various aspects of game design: intelligence of opponents [137, 162, 186], level design [28], or even entire game design [35].
- Designing robots (*evolutionary robotics*) [33, 57, 121].
- Training weights and designing architectures of neural networks (*neuroevolution*) [52, 68, 185].
- Heuristic selection/generation (*hyperheuristics*) [23, 37, 59].

The above list is far from being comprehensive. From the perspective of heuristic algorithms, there is nothing inherently special about evolutionary algorithms, and many other (meta)heuristics could be used in their place for the same problems. However, the broad range of successful applications demonstrates that evolutionary algorithms are an effective metaheuristic for many practical problems. And notably, the solutions found by evolutionary algorithms are often innovative and more effective than those devised by human experts.

4.2 Genetic Programming

Genetic programming (GP) was introduced as an evolutionary method of inductive synthesis of programs [103]. It works by rephrasing the program synthesis problem as an optimization task, in which an unknown target program has the optimal value (e.g., the smallest numerical error on some collection of examples, or the highest number of passed test cases or unit tests). Typically, solutions have variable-length representation, which is natural for computer programs and mathematical expressions, which can vary vastly in their complexity. Sometimes, the variable-length representation is achieved indirectly by means of `nop` instructions (which does nothing) or similar tactics[5], and a fixed-length representation – this approach is the most common in linear GP [34].

In the following sections, we will describe the typical tree representation used in GP (Section 4.2.1), and then initialization (Section 4.2.2) and variation (Section 4.2.3) operators that are compatible with it. After that, we will present the standard way of calculating fitness value (Section 4.2.4), and finally the practical applications (Section 4.2.5).

4.2.1 Representation of solutions

There are several ways in which programs can be represented in GP:

- expression trees [48, 103],
- nested lists of instructions (*Push* [2, 181]),
- graphs, i.e., expressions in which subexpressions can be reused multiple times (*Cartesian GP* [138]),

[5] For example, the lines of code that save some to a regsiter, and that value is never again used.

- variable-length vectors of integers specifying the productions of some formal grammar to use (*grammatical evolution* [146]).

In this thesis, we use only the expression trees, and thus we will describe in more detail only that approach, and for the descriptions of other representations we refer an interested reader to the cited publications and other sources [159].

To facilitate programming, high-level abstractions known as *programming languages* were devised. In these languages, basic building blocks of programs can be manipulated without worrying about low-level details of a processor or memory management, because the dedicated *compiler* of the language will automatically translate the high-level source code to the machine code. Usually, the source code of high-level programming languages is not processed by a compiler directly, but an intermediary form called *abstract syntax tree* (AST) is used. ASTs are "abstract" in the sense that redundant elements, which can be inferred from the tree (e.g., parentheses), are omitted and not directly represented as nodes. In this form, the logical relation between, e.g., function and its arguments, is directly represented. Many works in GP use this type of representation, because it makes it easy to apply variation operators and the search space is reduced by not including the redundant elements of the language. The examples of tree program representation for arithmetical expressions can be seen in Figure 4.5 and 4.6.

This leads us to the description of how a typical expression tree in GP is built. The basic unit of a solution's representation in GP is an *instruction*. An instruction is basically a function, although in some applications side effects in an *environment* are involved (e.g., changes in memory, or some actions of a simulated agent). Instructions can be divided into:

- **Terminals**, which do not need any arguments to produce a value (usually they are constants and program's inputs).
- **Nonterminals**, which accept a certain number of arguments (depending on the arity of a particular instruction) and then produce a value.

In the tree on the left in Figure 4.5, $+$ and $\times$ are nonterminals, and Y, X, and 1 are terminals. To execute a program represented by a tree, we move bottom-up, from leaves to the root. Whenever we are at a nonterminal node, we substitute it for the result produced by the node's function supplied with the leaves as arguments. This process continues until only a single constant is present in the root.

Loops and recurrence, the fundamental flow control elements in programming, can be realized in this representation too. However, in this thesis we limit ourselves to the non-recursive programs without loops, so we omit the discussion of this topic.

In the *strongly-typed GP* [139], each instruction has an associated output type and the types of its arguments. This is motivated by different kinds of data that instructions may require; for example, the if-then-else instruction expects a Boolean value for the condition, while the branches may return any other value (but in vast majority of cases, both branches should return values of the same type). Strongly-typed GP imposes additional constraints on which programs are valid, and enforces them by modifying initialization and variation operators so that an invalid program cannot be produced. We will describe the modifications to these operators in the relevant sections.

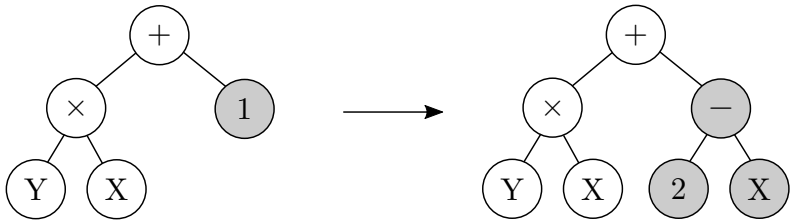


Figure 4.5: An example of mutation variation operator on tree representation. Grayed out node on the left was a root from which was generated a new branch, color in gray on the right.

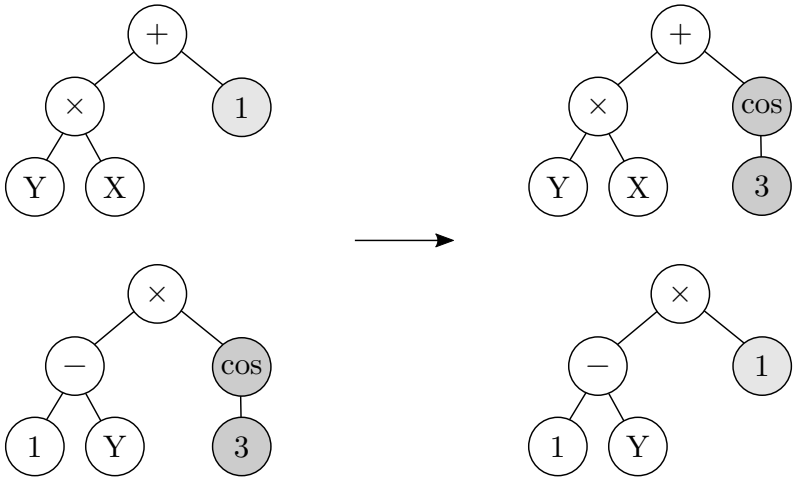


Figure 4.6: An example of the crossover variation operator on tree representation. The subtrees marked in gray are exchanged between the individuals. Depending on the implementation, either one offspring is returned, or both.

4.2.2 Population initialization

Koza [103] proposed two basic initialization algorithms for the tree program representation, which are still widely used to this day. Both have one parameter h_{max}, which defines a maximum height of the trees to be generated. The algorithms are:

- **Full**, which produces a tree with all leaves located at exactly the same depth h_{max}. To achieve this, any node at depth $h < h_{max}$ is selected randomly from a set of nonterminals, and all nodes on depth h_{max} are selected from the set of terminals. An example of a tree that could result from the application of this algorithm is presented in Figure 4.5 on the right.
- **Grow**, which constructs the tree recursively top-down selecting instructions randomly from the combined set of terminals and nonterminals, with the exception of the situation when node is at depth h_{max}, in which case it selects an instruction only from the set of terminals. Any valid tree (up to depth h_{max}) can be produced by this method, including the tree on the left in Figure 4.5, which cannot be produced by Full.

In order to improve diversification of the initial population, a combination of these methods was introduced by Koza under the name **ramped half-and-half** [103, p. 93].

In this method, a range of depths is used (in the experiments in his book, Koza used the range of $[2, 6]$), and the same number of solutions is created for each depth (so, for the range used by Koza, 20% of solutions would be generated with $h_{max} = 2$, 20% with $h_{max} = 3$, etc.). Then, for each depth, 50% of solutions are initialized using Grow, and another 50% using Full.

To adapt the initialization to the requirements of strongly-typed GP, when we are selecting a new node, we limit ourselves to only these instructions that return a value of the type required by the parent node. The return type of the root node is determined by the type that should be returned by entire program. During construction of a tree it may happen that a certain branch cannot be finished at depth h_{max} (e.g., if there are no terminals of some type) – in such cases the whole tree is usually discarded and then generated once again from the root.

4.2.3 Variation operators

In this section, we describe the most commonly used variation operators for the tree representation of programs (Section 4.2.1). Contrary to EA, in GP parents are typically *either* mutated or recombined with certain predefined probabilities, but these operations are never applied one after another on the same individual.

4.2.3.1 Subtree mutation

Subtree mutation [103, 159], which is the most often used mutation algorithm in GP, randomly selects a node in the parent tree, and then replaces (in the parent's copy) a subtree rooted there with a new randomly generated subtree (usually by the *Grow* procedure from Section 4.2.2). This process is illustrated in Figure 4.5.

Choosing nodes randomly with a uniform probability distribution leads to the undesirable growth of programs (*bloat*). To illustrate this, we can imagine a full binary tree of height h – in such a case, the number of internal nodes is $2^h - 1$, and the number of leaves is 2^h. This means that with the probability of approximately $1/2$, the node selected for mutation will be a leaf, and a new subtree will be added there, increasing the depth of the whole tree. In order to avoid this, a more elaborate node sampling strategy is often used (e.g., 90% for selecting internal nodes, and 10% for selecting leaves).

In strongly-typed GP, the same mutation procedure can be used, as long as a strongly-typed version of Grow (Section 4.2.2) is used to generate a new subtree.

4.2.3.2 Subtree Crossover

The most commonly used crossover operator in GP is *subtree crossover* [103, 159]. After two parents are provided as arguments, a node (called *crossover point*) is selected independently at random in both of them. Then, the subtree rooted at one parent's crossover point (with the crossover point included) replaces the subtree rooted at the second parent's crossover point, and vice versa. This process is visualized in Figure 4.6. The replacement is done on the copies of the parents, so that the original parents remain untouched ready to potentially produce more offspring. Although creating two offspring is very natural in this approach, according to Poli [159, p. 15] it is common to discard the second offspring

and return only one; however, both offspring are returned in the original description of subtree crossover by Koza [103, p. 101].

Choosing crossover points randomly with a uniform probability distribution leads to the undesirable effect of a very high probability of exchanging the leaves. To increase the impact of crossover on search, a similar node sampling strategy as for mutation is often used (i.e., 90% for internal nodes, 10% for leaves).

In strongly-typed GP, after selecting a crossover point in the first parent, the crossover point in the second parent is selected randomly from only those nodes that have the same return type. If no such node exists, then the crossover is aborted and the parent selection method is invoked again.

4.2.4 Fitness

The evaluation of programs in GP is typically conducted on the basis of interactions of programs with a set of *test cases* (also called *tests* for short). In most scenarios, a test case is a tuple (in, out), where in is the vector of program's inputs, and out is the target output that we want the program to produce. In some scenarios, e.g., evolution of game strategies, it is common for programs to compete against each other, and such interactions can be also counted as a test. The outputs of a program for all test cases can be collected in a single vector, which we will call *fitness vector* (often, this vector contains errors rather than raw program's outputs, or binary values indicating if a test was passed or not). This vector is then either directly used during parent selection (e.g., in lexicase selection; Section 4.1.8.2), or some aggregation of its elements is used, such as a sum or mean squared error (e.g., in tournament selection; Section 4.1.8.1).

4.2.5 Applications

In this section, we describe some interesting successful applications of GP. The works presented here are a selection of GP systems which were awarded or nominated at the annual Humies [5] competition held at the Genetic and Evolutionary Computation Conference (GECCO). For a more comprehensive review of applications of GP, we refer an interested reader to [159].

Probably the most recognized achievement of GP is that of Lohn et al. [124], who evolved an antenna for NASA's Space Technology 5 mission (sending into space microsatellites to measure the effect of solar activity on the Earth's magnetosphere). The antenna generated by GP met the mission requirements, while the antenna designed by a team of human experts did not. After the first prototype of the antenna was prepared and evaluated, the objectives of the mission changed. Impressively enough, in a month, the authors managed to evolve a new antenna, which was adapted to the new requirements and once again outperformed the one created by human experts. This is also an example of GP not used to evolve programs, but rather tree-like physical structures, since the nodes were representing rotations and branching of fragments of wire. The antennas produced by GP were highly irregular and asymmetrical, proving that evolutionary approaches can discover solutions unbiased by human's aesthetics or preconceived notions.

In the domain of software engineering, Weimer et al. [67, 193] used GP to automatically repair existing software artifacts written in the C programming language. Examples of the software being repaired are: Flex lexical analyzer [3], Null httpd web server [6], and operating system of the Microsoft's Zune media player [8]. The programs were represented as AST trees, and thus required compilation for each fitness evaluation, which was conducted on a set of unit tests, covering both the basic functionality (to avoid regression), and the situations for which the original program behaved incorrectly. The patches generated by GP were checked by both automatic fuzzers (stochastic generators of inputs which attempt to discover faults in the application's behavior), and manual code inspection, and no issues were detected.

Schmidt et al. [173] applied GP to the automatic discovery of physical laws based on measurements obtained from experiments. Starting with no prior knowledge, their GP system (re)discovered Lagrangians, Hamiltonians, and other laws of geometric invariance and momentum conservation. They have also observed that by seeding the initial population with terms of solutions for simpler, but still similar, problems (e.g., oscillations of a single pendulum, and a double pendulum), the effectiveness of search was greatly improved.

Spector et al. [179, 180] were the first to demonstrate that GP can be used for the synthesis of programs for quantum computers. Such computers operate very differently than the classical ones, and the design of quantum algorithms is hard for humans. Spector's GP system managed to automatically discover several known quantum algorithms for, e.g., the depth-two AND/OR tree problem, and the Deutsch-Jozsa problem. This line of research was continued by Massey et al. [132, 133], who managed to synthesize the quantum Fourier transform algorithm. While no fundamentally new quantum algorithm was discovered in these works, they demonstrate the theoretical possibility of such endeavors.

Recently, Real et al. [166] introduced the AutoML-Zero system, in which they used linear GP [34] to evolve from scratch, using only common mathematical operations (on scalars, vectors, and matrices; derivatives were not included), machine learning models for a set of classification tasks. Three components of machine learning algorithms were evolved in parallel: initialization, prediction, and learning step. As the evolution proceeded, many classical results were rediscovered, for example: linear models, simple neural networks trained by backpropagation, ReLU, and gradient normalization. What is interesting, the generated machine learning algorithms adapted to the characteristics of a particular task, for example, dropout-like techniques and noisy ReLU activation function were invented when the training data was scarce, and the exponential learning rate decay was invented when the training time (the number of epochs of training) was reduced.

Chapter 5

Evolutionary Program Sketching

Program synthesis by sketching [174, 176] is a paradigm in which a human expert provides a sketch of the solution (i.e., a partial program), the missing parts of which are then filled by a synthesis system. We present our approach to sketching, in which it is the evolutionary algorithm that performs the role of sketch-provider, and SMT solver is used to fill the missing parts with short provably optimal code fragments.

This chapter is based on the material published previously in [30], which was created in collaboration with Krzysztof Krawiec.

5.1 Introduction

In Chapter 2, we presented program synthesis primarily as a search problem, in which the objective is to find a program satisfying the correctness predicate (Definition 2.3.1). In this chapter, we will reformulate program synthesis as an optimization problem, which is also the typical mode of operation for genetic programming (GP; Section 4.2). More precisely, given a set of input-output examples (tests) T and a set of programs $\mathcal{L}$ (a programming language), we want to find a program $p \in \mathcal{L}$ that satisfies the maximum number of tests:

$$\max_{p \in \mathcal{L}} |(x, y) \in T : \ p(x) = y|. \tag{5.1}$$

Program synthesis by sketching [174, 176], briefly described in Section 2.6.1, eases the burden on the synthesis system by allowing users to specify a *sketch* of the solution, i.e., a partial program with *holes*. Instead of generating the whole program p from scratch, the goal of the synthesis system is to fill the holes in the provided sketch p with code fragments adhering to the programming language $\mathcal{L}$ in such a way that a specification is satisfied, or, in the optimization variant of program synthesis that we consider in this chapter, the maximum number of tests is passed:

$$\max_{\mathbf{b}} |(x, y) \in T : \ p_{\mathbf{b}}(x) = y|, \tag{5.2}$$

where $\mathbf{b}$ is an ordered list (of length equal to the number of holes in p) containing fragments of code, and $p_{\mathbf{b}}$ is a complete program created as a result of filling the holes in p with the corresponding code fragments in $\mathbf{b}$. Depending on the synthesizer's capabilities, the complexity of the elements of $\mathbf{b}$ may vary from just a constant or variable to a block of

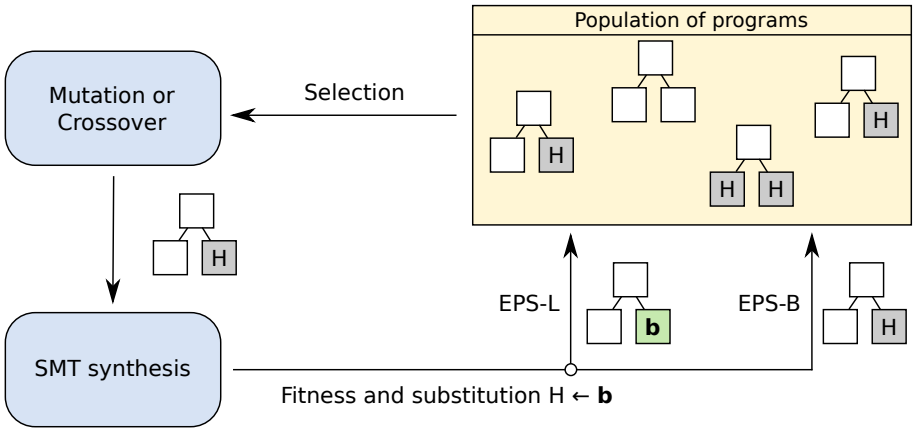


Figure 5.1: The diagram of EPS. H represents a hole, and **b** the content assigned to holes by the SMT synthesis module. The EPS-L and EPS-B variants of EPS are described in Section 5.2.1.

code. It is also worth noting that sketching can be easily applied to a mixed specification consisting of both tests and logical constraints.

While sketches make it easier to find the correct/optimal program by reducing the search space, they require additional work and knowledge from the user. The sketch proposed by the user can be suboptimal, or in the worst scenario even infeasible, meaning that it cannot be completed so as to satisfy the specification. To address these issues, we propose *Evolutionary Program Sketching* (EPS), in which genetic programming is responsible for generating sketches, and the user needs only to provide the tests.

5.2 Evolutionary Program Sketching

EPS consists of two main modules:

- **GP search**, responsible for finding good sketches.
- **SMT synthesis**, responsible for finding the content to fill the holes in sketches.

Both GP and SMT can be used to solve program synthesis problems on their own, but they have different strengths. SMT synthesis exhaustively searches the space of all possible programs while trying to prune that space as much as possible using logical deduction, which allows it to, e.g., accurately compute required values for constants, but at the cost of long runtime. GP, on the other hand, heuristically finds "good enough" solutions in reasonable time, but is not good at tweaking fine details, such as the aforementioned constants. By combining these two methods, we hope to achieve the best of both worlds.

Figure 5.1 presents the general principles of operation of EPS. GP maintains a population of candidate solutions, either sketches or complete programs, which we represent as trees. All program instructions, variables, and constants have an associated type (strongly-typed GP; Section 4.2.1). This also holds for holes, which are represented in EPS as terminal instructions of the same type as the output of the expressions they stand for. Whether holes are present in a program or not depends purely on chance, since they are treated by variation operators (initialization, mutation, crossover) as any other lexical element of the programming language $\mathcal{L}$. Holes are used only during evolutionary search,

and the final returned solution has always all holes filled.

A new population is created from the old one in the way typical for GP (Section 4.2), i.e., solutions are stochastically selected based on their fitness and recombined with each other (crossover), or simply a part of a program is randomly changed (mutation). After a new candidate solution p is created, EPS needs to compute its fitness, and this is where the SMT synthesis takes the stage. If there are no holes in p, then its fitness is simply the number of passed test cases, according to Equation 5.1. If there are holes, however, SMT solver searches for a list $\mathbf{b}$ of code fragments such that when they replace the holes, then the $p_{\mathbf{b}}$'s number of passed test cases would be the highest possible. In a single program there can be several holes (no upper limit on their number is set), and interactions between them can potentially exert big influence on the program's semantics. Thus, to assure that the program will have the best fitness possible, SMT solver needs to solve for all holes at once. To make this problem computationally more tractable, we assume that elements of $\mathbf{b}$ can be only constants and variables. We describe the details of the process of hole completion in Section 5.2.2.

5.2.1 EPS as an example of memetic algorithm

EPS falls into the wider category of *memetic algorithms* [140]. In memetic algorithms, an evolutionary algorithm is combined with some other heuristic, usually local search, which is responsible for the fine-tuning of candidate solutions. Effectively, evolutionary algorithm navigates the global landscape of the search space, while the local search finds the nearest local optima. In EPS, the role of the fine-tuning algorithm, this time non-heuristic, is taken by the SMT solver, and more precisely the optimization module [26] embedded in the Z3 SMT solver [53]. The consequence is that we have the guarantee that the best content for holes will be found.

There are two main types of memetic algorithms [104], which differ in how they handle the information obtained by the supporting heuristic:

- **Lamarckian memetic algorithms**, which replace the current candidate solution in the population with the one obtained by the supporting heuristic.
- **Baldwinian memetic algorithms**, where the current candidate solution is kept without changes, but its fitness is the same as that of the solution found by the supporting heuristic.

Similarly, we distinguish two variants of EPS depending on how they handle the hole completion list $\mathbf{b}$ and fitness $f_{\mathbf{b}}$ of $p_{\mathbf{b}}$ obtained from the SMT solver:

- **EPS-L** ("Lamarckian" EPS), in which a newly created candidate solution (sketch) p is replaced by $p_{\mathbf{b}}$ in the new population, i.e., the pair $(p_{\mathbf{b}}, f_{\mathbf{b}})$ is returned. It is, however, possible that the solver will not manage to find a solution in the assorted time budget (1.5 s), and in such a case EPS-L would return the pair $(p, 0)$.
- **EPS-B** ("Baldwinian" EPS), in which a new candidate solution p remains unchanged in the population, but has assigned the same fitness as $p_{\mathbf{b}}$, i.e., the pair $(p, f_{\mathbf{b}})$ is returned.

Table 5.2: The terminals, including different types of holes, available to variation operators in the configurations of EPS/GP considered in this thesis.

Configuration	*Terminals*		*Possible hole completion*	
	Constants	Input variables	Constants	Input variables
GP	✓	✓		
EPS_c	✓	✓	✓	
EPS_v	✓	✓		✓
EPS_{cv}	✓	✓	✓	✓

5.2.2 Filling holes in sketches

In order to find the optimal completion of holes in a sketch produced by GP, EPS creates a query to the SMT solver, which we present and describe in detail in Appendix A.7. This requires encoding the space of all possible hole completions in the form of a formal grammar, defining a formula for computing the number of passed tests (fitness), and then finding the completion of holes that leads to the best fitness. Optimization is beyond the original formulation of SMT satisfiability problem, and thus SMT solvers cannot be expected to handle it. However, in this case, a bisection algorithm may be used, because we have a discrete and bounded set of possible fitness values. By halving intervals and adding appropriate constraints, it is possible to determine the largest fitness value for which the synthesis formula is still satisfied, using only $\log_2 |T|$ solver queries. Alternatively, there are solvers with a built-in capability for optimization, like Z3 [26]. Our implementation is based on the latter, because it proved to be more efficient. In our experiments, the solver has a computational budget of 1.5 s, after which we assume that the problem was too hard and assign the worst possible fitness of 0 to the solution.

5.3 Experiment

The goal of the computational experiment is to investigate the effectiveness of EPS and to compare it with GP baselines.

5.3.1 Configurations

As mentioned in Section 5.2.1, we consider two basic variants of EPS varying in how they handle holes: **EPS-L**, and **EPS-B**. Additionally, we want to investigate the impact of the type of expressions allowed to replace holes on the effectiveness of search, and thus we consider the following types of holes:

- $\boldsymbol{c}$ – "constant holes", which can be replaced by *any* integer constant. This leads to the EPS-L_c and EPS-B_c configurations.
- $\boldsymbol{v}$ – "variable holes", which can be replaced by any input variable of the program. We consider this type of hole only for the benchmarks with more than one input variable. This leads to the EPS-L_v and EPS-B_v configurations.

Table 5.3: The shared parameters of the evolutionary algorithm.

Parameter	*Value*
Population size	250
Maximum height of initial programs	4
Maximum height of subprograms inserted by mutation	4
Constant terminals drawn from interval	$[0, 5]$
Maximum number of generations	100
Probability of mutation	0.5
Probability of crossover	0.5
Tournament size	7

Figure 5.4: The NIA grammar defining the set of considered programs. c stands for an ephemeral integer constant, v_i for the ith input variable, and h_j for the jth type of hole (in our experiments, there is always only one type of hole). `ite` stands for *if-then-else*, the conventional conditional statement.

$$
\begin{aligned}
\texttt{I ::= I + I | I - I | I * I | I / I | ite(B,I,I) |}\ & c \\
\texttt{|}\ & v_1 \ \texttt{|}\ v_2 \ \texttt{|}\ \ldots \ \texttt{|}\ v_k \\
\texttt{|}\ & h_1 \ \texttt{|}\ h_2 \ \texttt{|}\ \ldots \ \texttt{|}\ h_l \\
\texttt{B ::= I < I | I <= I | I = I | B = B | I >= I | I > I}
\end{aligned}
$$

- ***cv*** – "constant/variable holes", which can be replaced either by an integer constant, or an input variable. This leads to the EPS-L$_{cv}$ and EPS-B$_{cv}$ configurations.

The terminals present in each of these configurations are summarized in Table 5.2.

To determine if the evolutionary sketching supported by SMT solver is indeed better than pure evolution, we also consider the following GP baselines:

- ***GP*** – the GP algorithm configured in the same way as *EPS*, but with the holes excluded from the grammar in Figure 5.4.
- ***GP$_T$*** – the same as the *GP* configuration, but with a different runtime-based termination condition instead of the maximum number of generations. GP_T terminates either when a perfect solution is found, or when the average runtime of all EPS configurations on a given benchmark is reached (which implies that the EPS runs have to be conducted first). This configuration was motivated by the high runtime overhead generated by the SMT solver.
- ***GP$_{5000}$*** – the same as the *GP* configuration, but with a much bigger population size of 5000.

In Table 5.3 we present the shared parameters of GP, which were used for all introduced configurations. Tournament selection with $k = 7$ was used, as well as the standard subtree crossover and subtree mutation (Section 4.2.3). Evolutionary search terminates when a perfect solution is found, or a budget of 100 generations is exhausted.

Our implementation of all EPS and GP variants is accessible at https://github.com/iwob/EPS, and for communication with the Z3 SMT solver we use our own Python library accessible at https://github.com/iwob/pysv.

Table 5.5: The program synthesis benchmarks on which we test EPS.

Benchmark	*Arity*	*Formula*	*Tests* (T)	$\lvert T\rvert$
`Keijzer12`	2	$x_1^4 - x_1^3 + x_2^2/2 - x_2$	$x_1, x_2 \in \{-3, \ldots, 0, \ldots, 3\}$	49
`Koza1`	1	$x^4 + x^3 + x^2 + x$	$x \in \{-5, -4, \ldots, 0, \ldots, 4, 5\}$	11
`Koza1-p`		$3x^4 - 2x^3 + 6x^2 + 3x - 4$		
`Koza1-2D`	2	$x_1^4 + x_2^3 + x_1^2 + x_2$	$x_1, x_2 \in \{-3, \ldots, 0, \ldots, 3\}$	49
`Koza1-p-2D`		$3x_1^4 - 2x_2^3 + 6x_1^2 + 3x_2 - 4$		

5.3.2 Benchmarks

We test EPS on a suite of five integer polynomial benchmarks presented in Table 5.5, of which `Keijzer12` and `Koza1` are integer versions of the benchmarks from [134], and `Koza1-p`, `Koza1-2D`, and `Koza1-p-2D` are our variations on the `Koza1` benchmark. All benchmarks adhere to the NIA (nonlinear integer arithmetic) logic in SMT. In Figure 5.4 we present the programming language $\mathcal{L}$ (hole terminals presented there are used only in EPS configurations).

5.3.3 Discussion of the results

In Tables 5.6, 5.7, and 5.8 we present the results of the computational experiment in terms of the number of optimal solutions found (success rate), average fitness, and average runtime, respectively. Figure 5.9 contains more detailed statistics of the distribution of the fitness of the best-of-run programs on individual benchmarks. Based on this information, we reach the following conclusions:

- Overall, the EPS-B configurations perform better than or at least as good as the corresponding EPS-L configurations in terms of both success rate (Table 5.6), and the average end-of-run fitness (Table 5.7). This holds for 11 out of 13 pairs of corresponding EPS-B and EPS-L configurations. This is not surprising, given the conditions of the experiment (the same number of generations), since EPS-B is vastly more flexible because of the holes not being replaced in fitness evaluation. Unfortunately, Table 5.8 shows that this happens at the cost of an order of magnitude longer runtime.
- As for the comparison between different types of holes, for the Lamarckian variants (EPS-L) the best results were obtained by the EPS-L$_{cv}$ configuration, but its superiority over EPS-L$_c$ is only slight. Of note is that EPS-L$_v$ never achieved success, and the average fitness was also the poorest among the methods. As for the Baldwinian variants (EPS-B), the best results were obtained by the EPS-B$_c$ configuration, especially for the benchmarks with the arity of two, for which it outclasses the EPS-B$_{cv}$ variant that allows more flexible filling of holes. Similarly as before, the EPS-B$_v$ variant is definitely inferior, although it managed to see some successes.
- The EPS_c and EPS_{cv} variants outperformed the GP baseline on all benchmarks. The GP_T and GP_{5000} baselines, however, proved to be more challenging, and only the best of the EPS configurations, i.e., EPS-B$_c$ and EPS-B$_{cv}$, achieved better results.

Table 5.6: The success rates. An empty cell means that the success rate was zero.

	GP			EPS					
	GP	GP_T	GP_{5000}	L_c	L_v	L_{cv}	B_c	B_v	B_{cv}
`Keijzer12`			0.05			0.01	**0.39**	0.01	
`Koza1`	0.19	0.68	0.96	0.33	-	0.32	**1.00**	-	**1.00**
`Koza1-p`				0.05	-	0.03	**1.00**	-	**1.00**
`Koza1-2D`	0.01	0.12	0.20	0.02		0.11	**0.80**	0.21	0.23
`Koza1-p-2D`						0.01	**0.75**		

Table 5.7: The average end-of-run fitness.

	GP			EPS					
	GP	GP_T	GP_{5000}	L_c	L_v	L_{cv}	B_c	B_v	B_{cv}
`Keijzer12`	15.85	23.02	25.06	23.92	18.05	27.77	**39.05**	20.45	17.47
`Koza1`	5.89	9.74	10.87	9.93	-	9.83	**11.00**	-	**11.00**
`Koza1-p`	2.59	4.45	3.98	9.05	-	8.78	**11.00**	-	**11.00**
`Koza1-2D`	16.54	29.73	33.18	23.39	19.47	31.29	**45.42**	27.36	23.70
`Koza1-p-2D`	9.29	17.18	14.60	22.60	10.66	29.47	**46.23**	12.56	15.41

Table 5.8: The average runtime in seconds.

	GP			EPS					
	GP	GP_T	GP_{5000}	L_c	L_v	L_{cv}	B_c	B_v	B_{cv}
`Keijzer12`	**14.8**	11330.7	493.0	772.3	488.0	1578.6	15439.8	21172.6	28354.0
`Koza1`	**4.8**	291.0	46.3	699.8	-	801.4	652.0	-	695.8
`Koza1-p`	**4.5**	962.9	344.0	892.3	-	971.6	978.2	-	982.0
`Koza1-2D`	**16.0**	7635.8	431.9	793.1	478.7	1790.6	9076.9	16280.5	23033.8
`Koza1-p-2D`	**15.4**	9206.1	515.9	750.4	511.3	1725.7	11986.4	12390.8	27875.4

It seems that the adaptability of holes is the best that EPS can offer, and the "premature" loss of that adaptability, which happens in the EPS-L variants, makes this approach not competitive, given the SMT solver's runtime overhead.

- Concerning the average runtime, unsurprisingly the Baldwinian configurations are, by a huge margin, more time consuming (except for the simpler `Koza1` and `Koza1-p` benchmarks) than the corresponding Lamarckian variants – after all, they contain more holes, since they accumulate them during a run as the holes are introduced by the variation operators. It is also interesting that the Lamarckian configurations generally achieve the end-of-run fitness (and, to a lesser extent, success rate) that is comparable to GP_T, which was granted much larger time budgets. This, however, may be a result of bloat, which could have lessened the effectiveness of GP.

5.4 Conclusions

In this chapter, we described EPS, a novel approach to program synthesis by sketching, in which a user provides only a set of tests, and the sketches are generated automatically by

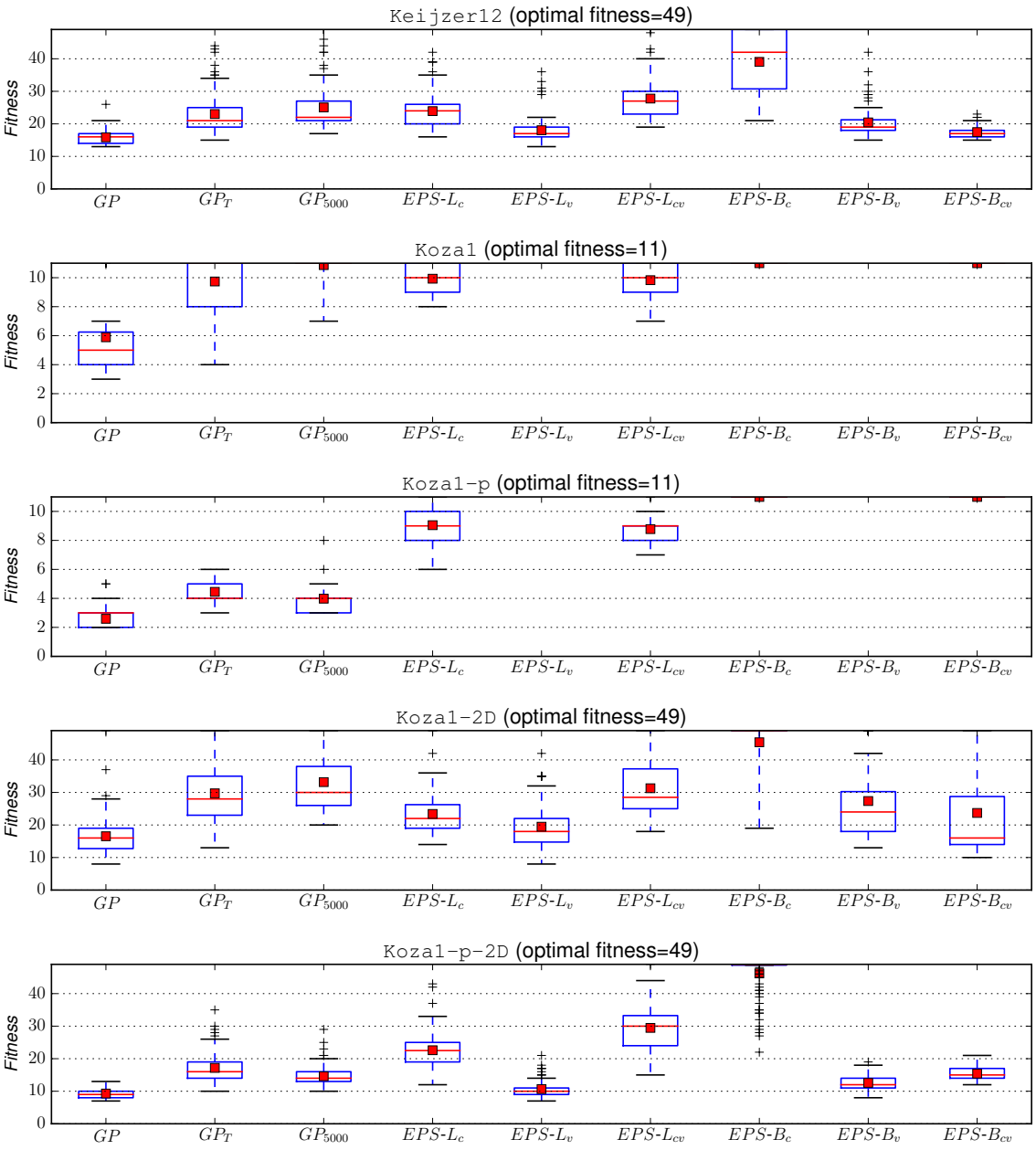


Figure 5.9: The box-and-whiskers plots of the fitness (number of passed tests) of the final solutions across all configurations and benchmarks. Boxes mark lower and upper quartiles, red line – median, red square – mean, whiskers – 1.5 of inter-quartile range below/above the corresponding quartile, and crosses – the outliers. Since the `Koza1` and `Koza1-p` benchmarks are univariate, the EPS-L_v and EPS-B_v configurations were not tested to them.

GP. We demonstrated that the Baldwinian variants of this approach with holes that can be filled by constants (i.e., EPS-B$_c$ and EPS-B$_{cv}$), outperform all baseline GP approaches, which try to find the optimal program by evolutionary search only. In EPS, the SMT solver performs logical deduction to reach the completion of holes leading to optimal fitness value, which can be posed as acquiring knowledge, both in the form of code fragments and fitness of the completed program. This knowledge is then subsequently used by GP during search to improve its effectiveness. Importantly, this technique allows to solve one of the known challenges for GP, which is finding the right values of constants – EPS equipped with an appropriate type of hole can find the required constant easily.

Admittedly, the experimental study was small, and is thus too limited to address the question if this approach would scale for more tests and more input variables. We consider this work to be a proof of concept rather than a mature approach, in contrast to the Counterexample-Driven Genetic Programming (CDGP) presented in the next chapter. Still, we decided to describe EPS, since it builds upon one of the more influential ideas in the field of program synthesis, i.e., program sketching [176].

Chapter 6

Counterexample-Driven Genetic Programming

In this chapter, we introduce Counterexample-Driven Genetic Programming (CDGP), a heuristic algorithm based on genetic programming for synthesis of provably correct programs from formal specifications. Formal specifications do not lend themselves easily to guiding evolutionary search, since the feedback they provide is binary: a program either passes the specification or not. To mitigate this problem, we use a deductive reasoning solver to verify solutions and find inputs for which programs behave incorrectly, and then these inputs are used to create additional test cases. The set of test cases grows with time, providing more fine-grained feedback as the search continues.

This chapter is based on the material published previously in [32, 106, 107], which was created in collaboration with Krzysztof Krawiec and substantial support from Jerry Swan. Special thanks go to John Drake for the help in presenting this work at the International Joint Conferences on Artificial Intelligence (IJCAI'18).

6.1 Introduction

Genetic programming (GP), as introduced in Section 4.2, is an evolutionary approach used for inductive program synthesis from a set of examples. Specifying a task with a set of examples is not only convenient for humans, but it also facilitates the creation of measures of candidate programs' quality. And while it is common knowledge that "testing shows the presence, not the absence of bugs" [56], unit tests are prevalent in software engineering, since they are a good compromise between the guarantees of correctness they offer on the one hand, and the effort of creation and maintenance on the other. In the domain of program synthesis from examples, where de facto unit tests (in the form of examples) are the only explicit criteria of correctness, this entails an inherent inductive aspect of the problem and the need for *generalization* to avoid overfitting.

There is, however, a group of applications where even the smallest fault is unacceptable, with safety-critical systems being the most prominent example. The main characteristic of safety-critical systems is that the cost of any failure, no matter how small, can be potentially very high. The situation is even worse if there is an adversarial party interested in the failure of said system and willing to actively search for errors. Examples of safety-

critical systems include medical equipment, expensive engineering projects, or integrated circuits. Another area of applications where faults are associated with a very high cost is mathematical modeling. For example, when scientists are searching for a mathematical model with certain properties based on few observations of some phenomenon, then any error in meeting those properties may render the model useless from the perspective of science, and potentially dangerous when subsequently applied in practice.

Formal specifications (also sometimes called *contracts* in the context of software engineering or programming languages [136]), described earlier in Section 2.5.1, are one of the solutions to the problem of generalization. They can either be the only source of information about program's expected behavior, or an additional set of constraints on top of examples. In this chapter, we will consider the first scenario, i.e., complete formal specification, for the task of integer-valued symbolic regression and operations on text strings. The second scenario will be considered in Chapter 7 for the problem of real-valued symbolic regression in the presence of noise.

We will begin with presenting existing work on heuristic program synthesis from formal specifications. After the context is laid out, we will proceed to describing CDGP, our approach based on GP to solving the problem of program synthesis from formal specification. The chapter will be concluded with an experimental study.

6.2 Related work

6.2.1 Formal specifications in GP

There are few previous works that combined evolutionary search with formal specification of program's behavior. Arguably, the biggest challenge is the lack of a readily available method of computing fitness. For the set of test cases, usually considered as a task specification in the GP literature, one can simply count the number of tests for which candidate program returns correct answer. In the case of formal specification, however, a program either satisfies it or not, so all incorrect candidate programs would have the worst fitness, and thus there would be no selection pressure towards improvement. Another challenge associated with formal specifications is the necessity of proving that a solution is correct for every possible input, and fast deductive reasoning solvers, as well as improved computer hardware, made finding such proofs feasible for larger problems rather recently. This progress in proof efficiency was especially important for generate-and-test metaheuristics like GP, because they need to call such solvers repeatedly during their runtime.

A natural way to overcome the problem of fitness bottleneck for formal specifications is to somehow decompose the full specification into a set of smaller independent parts which programs could still be evaluated upon. We can notice here that a similar process is happening behind the scenes for sets of test cases – programs, instead of being checked for correctness on all tests at once with a binary feedback, are evaluated on individual tests and those evaluations are then aggregated in some way. Similarly, a formal specification given as a formula in the conjunctive normal form, can be divided into a set of predicates, and programs can be evaluated on each of the predicates independently. That way, instead of a binary feedback, a more flexible spectrum of program correctness emerges. During evolutionary search, incorrect programs are rewarded for meeting any of the predicates,

and then, for example by means of the crossover operator, two programs satisfying different predicates may be combined into a single program satisfying both predicates at the same time.

To the best of our knowledge, most previous works on heuristic program synthesis from formal specifications are based on the idea outlined above, i.e., dividing specification into smaller parts, usually single subformulas, and then for each of them checking, if it is satisfied by a candidate program. The number of satisfied formulas can then be used as a fitness measure to be optimized. Some past works considered also an alternative approach based on the creation of test cases from the failed verification attempts during runtime of the search algorithm, and then using these tests to compute fitness. CDGP falls into the second category.

An approach due to Colin Johnson [88] incorporated model checking (Section 3.4) with the specification of the task expressed via Computation Tree Logic (CTL) to evolve finite state machines, and was used to learn a controller for a simple vending machine. The fitness was computed as the number of independent CTL formulas that were satisfied by a given program. A similar approach by He et al. [82], the *Hoare logic-based GP*, computes fitness as the number of postcondition clauses which can be inferred from the precondition and the program being evaluated. Instead of model checking, the Hoare logic [84] is used for the specification of the task and verification.

From 2008, Katz and Peled authored a series of papers combining model checking and GP [94, 95, 96, 97], in which they progressively refine their MCGP tool based on Linear Temporal Logic (LTL). The program specification consists of several independent LTL properties. Katz and Peled distinguished several levels of passing an LTL property (i.e., met for all inputs, met for only some inputs, met for no input), which they verified using model checking [94, 95]. Apart from the various levels of correctness and different formalism, this approach is very similar to the two previously described. For parametric programs (i.e., with unbounded input size), the authors abandoned the idea of providing full correctness guarantees and tested programs on counterexamples found by model checking [96, 97]. It is worth noting that in [96] Katz and Peled briefly considered using an SMT solver for verification of parametric programs instead of model checking, and used counterexamples to provide for more granular fitness in a similar spirit as CDGP. However, they only reported trying to solve a simple problem, and seemingly abandoned this line of research after that.

The use of coevolutionary GP to synthesize programs from formal specifications in the first order logic (augmented with arrays and arithmetic operators) was researched by Arcuri and Yao [13, 14]. They maintained separate populations of tests (generated from the specification) and programs within a competitive coevolution framework, in which programs were rewarded for passing tests, and tests were rewarded for failing programs. The fitness of programs was calculated using a heuristic that estimated how close a postcondition was from being satisfied by the program's output for specific tests. While allowing the synthesis of programs with GP from formal specifications, this approach provides no guarantees that the returned program will be consistent with the specification for all possible inputs.

6.2.2 Deductive program synthesis methods

In the domain of deductive program synthesis methods, the closest approach to CDGP is Counterexample Guided Inductive Synthesis (CEGIS), which was first described by Solar-Lezama et al. [176] in 2006, but the name itself was coined two years later in [175]. CEGIS is a very general scheme of combining an inductive program synthesizer with a formal verification procedure. It is assumed that the formal verification procedure either proves that a program is correct (thus terminating the search), or returns a counterexample otherwise. On the other hand, the inductive program synthesizer takes as an input a set of test cases (counterexamples) and produces a program that works correctly for that set. CEGIS operates in a loop: one starts from a randomly generated test case, from which the synthesizer produces a program. This program is then verified by the formal verification procedure, and if the program is incorrect a counterexample is returned. This counterexample is then added to the set of test cases, and the process continues until a globally correct program is found. From that perspective, CDGP is an instance of CEGIS, where the inductive program synthesizer is GP, and the verification procedure is realized by an SMT solver.

6.3 Counterexample-Driven Genetic Programming

A feedback obtained from formal verification can be only twofold: success or failure. On the other hand, to be effective, search-based synthesis algorithms such as GP require a more graded guidance through the search space. Thus, in order to synthesize programs with guarantees of correctness using search-based techniques, the primary problem is to elicit more detailed information about the quality of candidate programs from the specification.

The approaches considered previously in the literature (Section 6.2) focus mostly on the decomposition of specification into independent constraints, and the number of individually satisfied constraints constitutes candidate program's fitness. In contrast, *Counterexample-Driven Genetic Programming* (CDGP) solves this problem by 1) collecting counterexamples from failed verification attempts, and 2) constructing new test cases from the collected counterexamples and using them as a basis to compute fitness in the conventional way. To the best of our knowledge, approaches similar to CDGP, with the exception of [96], were not previously considered in the domain of search-based synthesis methods.

The conceptual diagram of CDGP is presented in Figure 6.1, where it is divided into several independent modules:

GP search A module responsible for generating new candidate programs. In the case of CDGP it is GP, but in general it can be any other generate-and-test search algorithm.

Testing A procedure for evaluating candidate programs and returning their fitness computed on the collected test cases T_c. In the simplest scenario, the fitness value is the number of tests for which program returns a correct output.

$\boldsymbol{T_c}$ A set of test cases. T_c is initially empty, and is gradually filled with tests created from counterexamples as the algorithm runs.

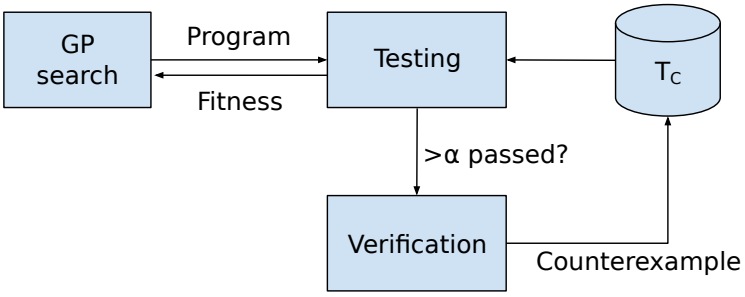


Figure 6.1: The conceptual diagram of CDGP.

Verification Performs formal verification using a deductive solver (e.g., SMT solver). Only solutions which pass a certain ratio α of tests in T_c are verified ($\alpha = 0$: verify always; $\alpha = 1$: all tests need to be passed before solution is verified). If the verification ends with an `unsat` answer, then no counterexample was found and CDGP terminates returning the correct candidate program as a final solution. Otherwise, a new test is added to T_c.

The reader might have noticed at this point that CDGP can work with a hybrid program specification that comprises both formal constraints and test cases – it suffices to initialize T_c with the tests from the specification. In this chapter, however, we make the following assumptions about formal specification:

- It is the only source of information about the search goal, i.e., the program to synthesize.
- It can be in any form accepted by the formal verifier. In other words, search algorithm cannot make any prior assumptions about the structure or content of formulas constituting the specification.

6.3.1 Verification/Evaluation of programs

Formal verification is a computationally costly procedure. Thus, for efficiency reasons, in CDGP verification is performed only if a program passes at least a ratio α of tests. If a candidate program is incorrect for at least one test case, then we already know that it will also not satisfy the formal specification. However, we might still want to perform verification in order to increase the number of test cases. Solver's proof process is guided by various heuristics, and, as our experiments have confirmed, the solver can produce a different counterexample than the ones already collected in T_c for which program being verified was incorrect. Without that capability, the only sensible setting for α would be 1.0 (i.e., all tests need to be passed). The technical details of the verification query to the SMT solver are presented in Appendix A.3.

The evaluation loop in CDGP is presented in Algorithm 6.2. As was mentioned earlier, in CDGP the fitness of solutions is computed based on the test cases collected in T_C (function Eval). Since formal verification step is both 1) dependent on the candidate program's fitness (or, more precisely, binary evaluation vector) through tests ratio α, 2) a criterion of program correctness, it makes sense to consider it as a part of the evaluation process. If

Algorithm 6.2: Evaluation in CDGP, given the current population P, the current set of tests T_c, program specification $(Pre, Post)$, and tests ratio α, returns either a correct verified solution, or the evaluated population together with an updated set of tests.

```
1:  function CdgpEval(P, T_c, (Pre, Post), α)
2:      T ← ∅                                          ▷ Working set of tests
3:      for all p ∈ P do                               ▷ Evaluation loop
4:          p.eval ← Eval(p, T_c, (Pre, Post))
5:          if p.eval ≥ α · |T_c| then                 ▷ At least α · |T_c| tests are passed
6:              c ← Verify(p, (Pre, Post))
7:              if c = ∅ then return p                 ▷ Correct program
8:              else
9:                  t_c ← CreateTest(c, (Pre, Post))
10:                 T ← T ∪ {t_c}                      ▷ Add newly created test t_c to T
11:     return (P, T_c ∪ T)
```

Algorithm 6.3: Evaluation of a single program in CDGP, given a program p, the current set of tests T_c, and program specification $(Pre, Post)$, returns a binary evaluation vector (we assume here that 1 = passed test).

```
1:  function Eval(p, T_c, (Pre, Post))
2:      eval ← []                                      ▷ Empty vector
3:      for all t ∈ T_c do
4:          res ← Run(p, t.in)                         ▷ Run program on test's input
5:          if t.out = null then                       ▷ Incomplete test
6:              eval ← eval ∪ CorrectOutput(res, t.in, (Pre, Post))
7:          else                                       ▷ Complete test
8:              if res = t.out then
9:                  eval ← eval ∪ [1]                  ▷ Correct program output
10:             else
11:                 eval ← eval ∪ [0]                  ▷ Incorrect program output
12:     return eval
```

formal verification (Verify) determines that a program p satisfies the specification, then that program is returned as a correct solution. Otherwise, a counterexample is returned, and it needs to be converted (by CreateTest) to a proper test case before it is added to T_c. A counterexample returned by solver is a logical model, with the interpretation of an input for which the program behaves incorrectly. For a proper test case, however, besides input we also require expected output, and thus the need for transformation of a counterexample to a test case, details of which are presented in Section 6.3.3.

During the evaluation of a given population P, all new test cases are collected in a temporary set T, and that set is merged into T_C only at the end of a generation. Tests ratio α is always applied to T_c, and thus whether program will be verified or not is independent of the order of programs in the population. When using CDGP for generational, as opposed to steady state, evolution scheme, then a small issue may arise because T_c is initially empty, which causes all programs in P to be verified. As a result, T_c may grow in size immensely already after the first generation. In order to prevent this, an additional parameter $maxNewTestsPerIter$ (not presented in Algorithm 6.2) limits how many new

tests can be added to T in a single generation.

6.3.2 Complete and incomplete tests

The process of candidate program evaluation is, in fact, more complex than Figure 6.1 may suggest. Depending on the characteristics of the synthesis problem, we are forced to consider two types of tests:

Complete tests Tests of the form (in, out), where out is the only correct answer for input in. For example, $(\{x = 2, y = 5\}, 5)$ for the `max2` problem.

Incomplete tests Tests of the form $(in, null)$, where there are many correct answers for in, or the expected answer is defined relative to the program's output for some other inputs. For example, $(\{x = 2, y = 5\}, null)$ for the `max2` problem if we remove the requirement that the result is equal to one of the inputs.

Because of these differences, complete and incomplete tests need to be evaluated differently.

For complete tests, since we know the correct answer and that it is the *only* correct answer, we can directly execute a program on the test's input in and compare the result with the test's expected output out. This is exactly the same as it is usually done in GP.

For incomplete tests, we similarly compute the program's output for test's input in, but the task of determining whether it is correct is relegated to the Satisfiability Modulo Theories (SMT) solver. Given the program's output, test's input, and formal specification, the solver tries to conduct a proof that the output satisfies the specification (the technical details are described in Appendix A.4). A consequence is that evaluation of incomplete tests is much slower than that of complete tests, and CDGP will always try to create complete tests whenever possible.

The process of evaluation of a single program p is presented in Algorithm 6.3. We iterate over all tests, and for each of them we need to determine if a program is correct or not. In order to do this, we begin with executing p on the test's input (function Run). Then, depending on whether the test is complete or incomplete, we respectively either compare the result with the expected output of the test, or use a solver to determine correctness (function CorrectOutput).

6.3.3 Creation of test cases

Complete tests can be created only if for the test's input in there is only one correct output – we will call this a *single-output property*. We can distinguish two variants of this property:

- Local single-output property.
- Global single-output property.

Figure 6.5 presents example specifications illustrating these properties.

Definition 6.3.1. *Local single-output property* states that for a given input in and formal specification $(Pre, Post)$:

$$Pre(in) \implies \forall_{y_1,y_2}\ Post(in, y_1) \wedge Post(in, y_2) \Leftrightarrow y_1 = y_2$$

Algorithm 6.4: Function for the transformation of counterexamples into test cases, given counterexample c, and program specification $(Pre, Post)$, returns either a complete test, or an incomplete test.

```
1:  function CreateTest(c, (Pre, Post))
2:      out_1 ← FindOutput(c, (Pre, Post))          ▷ Use solver to find correct output
3:      if out_1 = ∅ then                            ▷ No correct output
4:          Exception("The specification is contradictory.")
5:      else if GlobalSingleOutput((Pre, Post)) then
6:          return (c, out_1)
7:      else
8:          Post' ← Post ∪ {f(c) ≠ out_1}            ▷ f is a target function in Post
9:          out_2 ← FindOutput(c, (Pre, Post'))
10:         if out_2 = ∅ then
11:             test ← (c, out_1)                    ▷ Complete test
12:         else
13:             test ← (c, null)                     ▷ Incomplete test
14:         return test
```

where y_1 and y_2 are some possible values of program's output.

Definition 6.3.2. *Global single-output property* is an extension of the local single-output property to all possible inputs:

$$\forall_{in}\ Pre(in) \implies \forall_{y_1,y_2}\ Post(in, y_1) \wedge Post(in, y_2) \Leftrightarrow y_1 = y_2$$

In order to check, whether the global single-output property holds, an appropriate query to the solver is created (a detailed description is presented in Appendix A.6) and executed once at the beginning of a CDGP run (for clarity, in Algorithm 6.4 this call is inside CreateTest function). If solver proves that this property holds, then only complete tests will be created during this run of CDGP. Otherwise, a check for the local single-output property will be performed during the creation of a test case from a counterexample.

The process of transforming a counterexample to a test case is presented in Algorithm 6.4. It starts with a call to the FindOutput function, which uses the solver to find some correct output out_1 for the counterexample (technical details are described in Appendix A.5). We can distinguish three main scenarios in the CreateTest function:

- If no correct output out_1 was found, then the specification is contradictory. It may at first seem a little counterintuitive, but if a function is undefined at some point, then any value returned by a program should be accepted as correct. The absence of a correct value implies that there are at least two mutually incompatible constraints which result in the synthesis task having no solution.
- If some correct output out_1 was found and we have ascertained that the global single-output property holds, then we can immediately return a complete test composed of the counterexample and the correct output.
- If some correct output out_1 was found and we have ascertained that the global single-output property *does not* hold, then we need to check for the local single-output

$$f(x,y) \geq x \quad \wedge$$
$$f(x,y) \geq y \quad \wedge$$
$$(f(x,y) = x \ \vee \ f(x,y) = y)$$

(a) Global single-output.

$$f(0,0) = 0 \quad \wedge$$
$$f(x,y) = f(y,x)$$

(b) Local single-output at $(0,0)$.

$$f(x,y) \geq x + y$$

(c) None.

Figure 6.5: Examples of formal specifications in the context of global and local single-output properties.

property by once again using the FindOutput function, but this time adding an additional constraint to exclude the previous value out_1. If an additional correct output out_2 is found, then an incomplete test will be created; otherwise, it will be a complete test with the correct output out_1 found earlier.

6.4 Design of experiments

In this section, we describe experimental setup and some implementation details of CDGP, such as search operators.

6.4.1 Benchmark suite

All benchmarks used in the experiments are expressed in the SyGuS format [165] (Section 2.4), so each program synthesis task description consists in:

- A signature (declaration) of the function to be synthesized (target function), i.e., its name, arguments and their types, type of the function's output.
- A context-free grammar defining the syntax of a function body.
- A formal specification of the target function's expected behavior represented as a set of logical constraints using symbols defined in a certain fixed theory (e.g., theory of integer arithmetic).

The benchmarks can be divided into two families:

- **LIA (Linear Integer Arithmetic) benchmarks** – a program to be synthesized consists of arithmetical operators and a conditional if-then-else statement. The list of benchmarks used is presented in Table 6.6, and the grammar[1] for solutions in Figure 6.7. All LIA benchmarks can be downloaded from: https://github.com/iwob/CDGP/tree/master/resources/benchmarks_phd/cdgp/LIA.

  The `Max`, `Search`, and `Sum` benchmarks were taken from the Conditional Linear Integer Arithmetic track of SyGuS 2017 competition; the remaining benchmarks are of our own design. Some benchmarks (`IsSeries`, `IsSorted`, `Search`) interpret input arguments as a fixed-length ordered sequence of type `I`. In the `IsSeries` and `IsSorted` benchmarks, the program is required to return 1 if the arguments form,

[1] The original SyGuS benchmarks did not have grammar explicitly specified, meaning that any expression adhering to the LIA logic is a valid program. The grammar we assumed in CDGP is a subset of the full original grammar, and is slightly biased toward common arithmetical functions.

respectively, an arithmetic series or are sorted in ascending order, and 0 otherwise. In the `Search-`n benchmarks, a correct program should return a 0-based index of the last argument in an 'array' of length n formed by the other arguments (which are constrained by a precondition to be already sorted). Hence, for instance, `Search2(3,7,1)=0`, `Search2(3,7,4)=1`, and `Search2(3,7,10)=2`.

- **SLIA (Strings with Linear Integer Arithmetic) benchmarks** – a program to be synthesized consists of arithmetical operators and string manipulation operators. The list of benchmarks used is presented in Table 6.8, and the grammar[2] of solutions in Figure 6.9.

  The benchmarks we use in our experiments are based on the benchmarks from the *Programming By Example Strings* track of SyGuS 2017 competition[3]. Since in these benchmarks a task is specified by means of input-output examples, and in this chapter we are concerned only with synthesis from full formal specification, we converted the input-output examples to a formal specification representing the same intended task. This often required addition of task-specific preconditions in order to make the specification fully describe the intended task. Our converted benchmarks can be downloaded from: https://github.com/iwob/CDGP/tree/master/resources/benchmarks_phd/cdgp/SLIA.

To visually distinguish the two families of benchmarks in the text, names of the LIA benchmarks begin with a capital letter, and names of the SLIA benchmarks begin with a lower case letter.

6.4.2 Program representation

In our implementation of CDGP, we evolve complete SMT-LIB expressions ('programs') represented as trees, in which internal nodes correspond to nonterminals, and leafs to terminals of the benchmark's grammar. Because of SMT-LIB's LISP-like functional structure, tree representation typical for GP is a natural and convenient choice here. Each node, besides the operator's name, stores also information about the grammar's production it was created from. This allows the search operators (Section 6.4.3) to easily replace any given node with a new randomly generated one so that candidate programs always satisfy the grammar provided by a user.

6.4.3 Search and selection operators

Since in SyGuS problems the search space is explicitly defined by a formal grammar, it is necessary to ensure that programs produced by GP conform to that grammar. In order to achieve this, we use a strongly-typed variant of GP [139], where each production of the supplied grammar defines a separate type, and search operators are designed so that they always respect type constraints.

[2]The grammar is not consistent across SyGuS benchmarks; for example, sometimes certain instructions are not present. Additionally, constant terminals in the grammar differ across benchmarks.

[3]The original SyGuS 2017 string benchmarks can be downloaded from: https://github.com/SyGuS-Org/benchmarks/tree/master/comp/2017/PBE_Strings_Track

Table 6.6: LIA benchmarks. The input type is $\texttt{I}^n$ and the output type is `I` (`I`=integer). Some functions were tested in variants with different arities.

Name	*Arity*	*SyGuS*	*Semantics*
CountPos	2, 3, 4		The number of positive arguments
IsSeries	3, 4		Do arguments form an arithmetic series?
IsSorted	4, 5		Are arguments in ascending order?
Max	4	✓	The maximum of arguments
Median	3		The median of arguments
Range	3		The range of arguments
Search	2, 3, 4	✓	The index of an argument among the other arguments
Sum	2, 3, 4	✓	The sum of the first two arguments with a sum ≥ 15

Figure 6.7: The grammar defining the domain of LIA programs. v_i is the ith input variable, `ite` is the conditional statement, `%` is the modulo operator. The starting symbol is `I`.

```
I ::= C | I + I | I - I | I * C | I / C | I % C | ite(B,I,I)
      | v1 | v2 | ...  | vn
C ::= -10 | -9 | ...  | -1 | 0 | 1 | ...  9 | 10
B ::= and(B,B) | or(B,B) | not(B)
      | I < I | I <= I | I = I | I >= I | I > I
```

Table 6.8: SLIA benchmarks. Input type is `S` or $\texttt{S}^2$, and the output type is `S` (`S`=string).

Name	*Arity*	*Semantics*
dr-name	1	Extract first name from full name and prepend it with "Dr."
firstname	1	Extract first name from full name
initials	1	Extract initials name from full name
lastname	1	Extract last name from full name
combine	2	Combine first and last name into full name
combine-2	2	Combine first and last name into first name followed by initial
combine-3	2	Combine first and last name into initial followed by last name
combine-4	2	Combine first and last name into last name followed by initial
phone	1	Extract the first triplet of digits from a phone number
phone-1	1	Extract the second triplet of digits from a phone number
phone-2	1	Extract the third triplet of digits from a phone number
phone-3	1	Put first three digits of a phone number in parentheses
phone-4	1	Change all '-' in a phone number to '.'

Figure 6.9: The grammar defining the domain of String programs (some instruction names were changed for clarity). *inputs* are the input variables, *constants* is a benchmark-specific set of constants of the same type as the production, `++` is a string concatenation. Boolean operators are never used and thus omitted. The starting symbol is `S`.

```
S ::= " " | S ++ S | replace(S,S,S) | charAt(S,I) | fromInt(I)
      | substring(S,I,I) | inputs | constants
I ::= constants | I + I | I - I | len(S) | indexOf(S, S, I)
      | fromString(S)
```

Initialization Starting from the starting symbol of the grammar, a program tree is recursively constructed by randomly choosing expressions from the right-hand sides of the productions. Once the depth of any node of a program tree reaches 3, terminals are picked whenever possible. If the depth exceeds 4, the tree is discarded and the process starts anew.

Mutation A random node in the program tree is picked, and then replaced with a randomly generated subtree. To conform to the grammar, the process of subtree construction starts with a grammar production of the type corresponding to the picked location (e.g., if the return type of the picked node is `I`, the generation of the replacing subtree starts with the production `I` of the grammar).

Crossover A random node x is drawn from the first parent program, and then one creates a list L of the nodes in the second parent that have the same type. If L is empty, a node from the first parent is drawn again and the procedure is repeated. Otherwise, a node y is drawn uniformly from L, and the subtree rooted in x in the first parent is exchanged with the subtree rooted in y in the second parent. This process is guaranteed to terminate, since both parent trees always feature at least one node of the type associated with the root node (`I` for LIA and `S` for SLIA) and the root nodes are allowed to be swapped.

As for selection operators, in our experiments we compare the effectiveness of:

Tournament selection (Section 4.1.8.1), where fitness is computed as the number of *not* passed test cases (so that 0 is an optimal value). The size of the tournament used in our experiments was 7.

Lexicase selection (Section 4.1.8.2), where at every selection step solutions that do not pass a given test are filtered out.

To control bloat, we set the limit on the maximum height of programs to 12. Additionally, if tournament selection has to choose between programs with the same fitness, it will prefer the shortest one (this is also known as *lexicographic parsimony pressure* [126]).

6.4.4 Population replacement

The GP literature usually considers the **generational** variant of population replacement, in which a new population is created in one step directly from a parent population. For CDGP, this design choice has a potentially important consequence, namely that during initialization of the evolutionary algorithm, because of the set T_C of test cases being empty, all programs are being verified, leading usually to a substantial amount of created test cases. This may lead to a higher number of costly program verifications (in case of incomplete tests) and the search process trying to satisfy many test cases (objectives) at the same time.

In order to verify the impact of this phenomenon, and compare it with a more gradual approach, we decided to also test the **steady state** variant (Section 4.1.10) of population replacement, in which a single iteration step consists of replacing one candidate solution with another created from mutation or crossover. In this mode of operation, verification is not conducted during initialization of the evolutionary algorithm (although it technically

Table 6.10: Parameters of CDGP and GPR used in the computational experiments.

Parameter	*Value*
Number of runs	50
Population size	500
Maximum number of generations	∞
Maximum runtime in seconds	1800
Solver timeout in seconds	3
Probability of mutation	0.5
Probability of crossover	0.5
Tournament size	7
Maximum height of initial programs	4
Maximum height of trees inserted by mutation	4
Maximum height of programs in population	12

could), but after the evaluation of each newly created candidate solution. This implies that at most one new test will be added to T_C in each generation, and that after the first generation $|T_C| = 1$. Notice, that if $\alpha > 0.0$, then the next program verification will be conducted only when some program works correctly for that initial test, which may slow the search in the beginning if the test is hard to satisfy. This also makes this approach the most similar to the CEGIS scheme described earlier (Section 6.2.2).

In the steady state variant, besides a selection operator, there is also a *deselection* operator responsible for removing solutions from the population – in the experiments we assume that it is the same as the selection operator but with a reversed ordering, so, for example, in tournament deselection, for removal, a candidate solution with the worst fitness among those considered will be selected.

6.4.5 SMT solver

The SMT solver used in our experiments is Z3 [53, 55] developed by Microsoft Research, which is one of the most performant and widely-used noncommercial open source SMT solvers. This choice was arbitrary, and no Z3-specific features were used.

All queries to the solver are expressed in SMT-LIB language version 2.5 [19, 21], recognized by most contemporary SMT solvers. Since the SyGuS benchmark specification language is build upon SMT-LIB, almost all constraints, declarations, and defined functions can be directly translated into their SMT-LIB counterparts.

The calls to SMT solver are the main efficiency bottleneck of CDGP. The time needed to solve a given query varies widely depending on its characteristics and particular benchmark, and for this reason we set a limit of 3 seconds for solving a query. If the solver does not respond in that time, then the `unknown` status is returned and appropriately handled by CDGP (e.g., if `unknown` is returned during verification, no new test case is created and search process is not terminated). Another consequence of this efficiency bottleneck is that the primary termination criterion for CDGP in our experiments is wall-clock time, rather than the number of generations. This makes the comparison between different configurations of CDGP, especially those differing by tests ratio α, more fair.

6.4.6 Baseline: GPR

In order to investigate the role of counterexamples in guiding search, we devised GPR (GP Random), a baseline variant of CDGP which, after the verification of a program, discards a counterexample (if it was found) and instead generates a test with random input. The correct output of this test is determined in the same way as in CDGP (Section 6.3.3).

For the LIA benchmarks, we narrowed the range of randomly generated inputs to $[-100, 100]^n$, where n is the arity of the function being synthesized. For the SLIA benchmarks, on the other hand, due to the presence of rather restrictive preconditions, GPR was not used, because most random sequences would be uninformative for the search process, as function's output is undefined for them – which possibly is an additional advantage of the test acquiring strategy used by CDGP for such precondition-heavy problems.

6.5 Experiment 1: Evaluation of CDGP on LIA benchmarks

The goal of the first experiment was to assess the effectiveness of CDGP and its baseline GPR on the LIA benchmarks (Table 6.6). The experiment described here is similar to that conducted in [32], but with an improved code base[4], a newer version of Z3 SMT solver[5], and reduced maximum computation time from 1 hour to 30 minutes. Experiments were performed on a computational cluster with computers equipped with Intel i7-4790 3.60GHz CPU. The dimensions of the experiment are:

- Method: CDGP, GPR.
- Population replacement: generational (`G`), steady state (`S`).
- Selection operator: tournament with $k = 7$ (`tour`), lexicase (`lex`).
- Tests ratio α: 0.0, 0.25, 0.5, 0.75, 1.0.

In the following, we will denote a particular algorithm's configuration using the notation:

```
<method>/<population replacement>/<selection>/<tests ratio>
```

using the abbreviations provided above, so, for example, a generational variant of CDGP with tests ratio $\alpha = 0.5$ and lexicase selection is represented by `CDGP/G/lex/0.5`. Sometimes '$*$' will be used to represent all values of a certain parameter; for example, `CDGP/*/lex/*` would represent all CDGP configurations with the lexicase selection.

Tables 6.11 and 6.12 present success rates on the LIA benchmarks for, respectively, CDGP and GPR configurations. The problem of program synthesis from formal specification is essentially an *uncompromising program synthesis problem* [83], meaning that partially correct solutions are of no interest. For this kind of problems, a success rate, i.e., the ratio of times when a correct solution was found, is the most natural measure of effectiveness.

We start by observing that CDGP outperfomed GPR on practically all benchmarks for both generational and steady state variants, and on many of them by a significant

[4] https://github.com/iwob/CDGP, master branch, commit `1aefd7f` from 2nd September 2020.
[5] https://github.com/Z3Prover/z3, master branch, commit `2fb914d` from 27th July 2020.

Table 6.11: Success rate of **CDGP** configurations on the **LIA benchmarks**. An empty cell means that the success rate was zero.

	generational										*steadyState*									
	Tour					*Lex*					*Tour*					*Lex*				
	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0
CountPos2	0.92	0.98	0.94	**1.00**	0.96	**1.00**	**1.00**	**1.00**	**1.00**	0.92	0.98	0.96	0.98	0.96	0.76	**1.00**	**1.00**	**1.00**	**1.00**	0.94
CountPos3	0.32	0.42	0.44	0.66	0.50	0.94	**0.96**	0.92	0.94	0.40	0.12	0.14	0.18	0.50	0.14	0.32	0.46	0.52	0.90	0.58
CountPos4		0.02		0.02		0.36	0.38	0.48	**0.52**	0.04				0.04		0.04	0.10	0.14	0.32	
IsSeries3	0.64	0.66	0.66	0.86	0.54	**0.92**	0.84	0.84	0.84	0.34	0.26	0.36	0.36	0.72	0.58	0.60	0.50	0.60	0.86	0.48
IsSeries4	0.20	0.08	0.22	0.46	0.32	**0.48**	0.44	0.36	0.46	0.10	0.08	0.02	0.10	0.40	0.30	0.04	0.08	0.12	0.20	0.34
IsSorted4	0.94	0.90	0.94	0.98	0.96	**1.00**	**1.00**	0.98	**1.00**	0.80	0.90	0.90	0.98	0.98	0.68	0.98	0.98	0.96	0.98	0.80
IsSorted5	0.86	0.86	0.98	**1.00**	0.82	0.94	**1.00**	0.98	0.94	0.54	0.72	0.68	0.92	0.96	0.52	0.90	0.92	0.86	0.90	0.66
Max4	0.98	**1.00**	0.98	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.90	0.88	0.98	**1.00**	0.88	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
Median3	0.98	0.96	0.96	**1.00**	0.96	**1.00**	**1.00**	**1.00**	**1.00**	0.98	0.70	0.76	0.86	0.98	0.66	0.98	0.98	**1.00**	**1.00**	0.98
Range3	0.82	0.86	0.84	0.94	0.92	**1.00**	0.98	0.98	**1.00**	0.92	0.26	0.34	0.68	0.98	0.64	0.76	0.84	0.84	0.92	0.94
Search2	0.96	0.94	0.92	0.98	0.92	**1.00**	**1.00**	**1.00**	0.98	0.86	**1.00**	0.94	0.96	0.98	0.70	0.98	0.94	0.98	**1.00**	0.84
Search3	0.74	0.88	0.96	0.98	0.94	0.96	**1.00**	0.96	0.98	0.88	0.80	0.90	0.96	0.94	0.46	0.98	0.94	0.92	**1.00**	0.88
Search4	0.62	0.74	0.66	0.88	0.76	0.90	0.96	**0.98**	0.94	0.68	0.52	0.58	0.78	0.70	0.06	0.64	0.70	0.78	0.90	0.78
Sum2	0.62	0.32	0.52	**1.00**	0.96	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.58	0.22	0.94	0.98	0.90	**1.00**	**1.00**	**1.00**	**1.00**	0.98
Sum3	0.12	0.38	0.66	0.96	0.72	**1.00**	**1.00**	**1.00**	0.98	0.68	0.02	0.12	0.34	0.82	0.22	0.60	0.78	0.82	0.96	0.68
Sum4		0.02	0.10	0.46	0.12	0.54	0.68	0.66	**0.72**	0.06			0.02	0.42		0.06	0.10	0.22	0.42	0.16
Mean	0.61	0.63	0.67	0.82	0.71	0.88	0.89	0.88	0.89	0.64	0.49	0.49	0.63	0.77	0.47	0.68	0.71	0.73	0.84	0.69
Rank	15.03	13.38	12.88	5.50	11.91	3.84	3.31	4.38	3.94	13.62	16.81	17.44	13.41	8.47	17.81	10.78	10.31	8.91	5.84	12.44

Table 6.12: Success rate of **GPR** configurations on the **LIA benchmarks**. An empty cell means that the success rate was zero.

	generational										*steadyState*									
	Tour					*Lex*					*Tour*					*Lex*				
	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0
CountPos2	0.70	0.70	0.68	0.88	0.82	**0.96**	0.80	0.86	0.88	0.78	0.22	0.34	0.36	0.64	0.48	0.66	0.84	0.82	0.86	0.78
CountPos3	0.08	0.02	0.10	0.16	0.40	**0.56**	0.54	0.48	0.54	0.04				0.08	0.12	0.04	0.06	0.06	0.26	0.44
CountPos4						**0.18**	0.14	0.16										0.04		
IsSeries3					0.06	0.06	0.02	0.04	0.04	**0.08**					**0.08**					0.06
IsSeries4																				
IsSorted4	0.02	0.04			0.20	0.62	0.54	0.64	0.66	0.30					0.22	0.42	0.44	0.54	0.48	**0.70**
IsSorted5					0.02	0.08	0.08	0.14	0.08	0.04										**0.18**
Max4	0.76	0.72	0.74	**1.00**	**1.00**	0.98	0.98	**1.00**	0.96	0.90	0.22	0.22	0.30	0.88	0.98	0.94	0.92	**1.00**	0.92	0.96
Median3	0.90	**0.98**	0.96	0.96	0.90	**0.98**	0.96	0.92	0.96	0.90	0.08	0.10	0.08	0.64	0.64	0.80	0.78	0.86	0.92	0.90
Range3	0.84	0.86	0.94	0.94	**1.00**	0.98	0.98	0.98	0.98	0.92		0.02	0.20	0.64	0.52	0.36	0.48	0.70	0.92	0.98
Search2	0.94	**0.98**	**0.98**	**0.98**	0.92	0.92	**0.98**	0.90	0.96	0.82	0.78	0.72	0.80	0.80	0.74	0.94	**0.98**	0.90	0.96	0.82
Search3	0.82	0.92	0.84	0.88	0.80	0.90	**0.96**	0.94	0.92	0.70	0.26	0.38	0.60	0.68	0.32	0.64	0.70	0.80	0.92	0.60
Search4	0.64	0.60	0.76	0.78	0.52	0.88	0.84	**0.90**	0.80	0.56	0.04	0.20	0.20	0.20	0.12	0.10	0.32	0.54	0.76	0.66
Sum2	0.98	0.96	0.92	0.96	**1.00**	0.92	**1.00**	0.96	0.98	**1.00**	0.08	0.08	0.12	**1.00**	**1.00**	0.88	0.96	0.96	0.98	0.94
Sum3	0.36	0.44	0.52	0.58	**0.82**	0.72	0.76	0.76	0.68	0.40			0.02	0.14	0.36	0.20	0.20	0.36	0.42	0.76
Sum4		0.02	0.02	0.08	0.24	**0.40**	**0.40**	0.36	0.26	0.04					0.02			0.02	0.04	0.14
Mean	0.44	0.45	0.47	0.51	0.54	0.63	0.62	0.63	0.61	0.47	0.11	0.13	0.17	0.36	0.35	0.37	0.42	0.47	0.53	0.56
Rank	11.91	10.75	10.84	8.38	7.28	4.84	4.47	4.94	5.56	10.00	17.56	17.09	16.47	14.03	12.28	14.00	12.16	10.53	8.81	8.09

margin. There is, however, a big difference between GPR's effectiveness for generational and steady state configurations; the latter proved to be much less successful, while the former was not that far behind CDGP on the easier benchmarks.

It seems that extreme values of α are not optimal for CDGP on the LIA benchmarks, and $\alpha = 0.75$ managed to be on average the best. We speculate that, in the case of "conservative" $\alpha = 1.0$ variant, the low amount of created tests hampers progress because fitness is not granular enough to give a good feedback to the evolutionary search process. On the other hand, costly formal verification is the efficiency bottleneck of CDGP, and thus configurations with low α ("non-conservative"), while generating more tests, spend also more time waiting for the solver. For GPR, the overall influence of α seems to be more nuanced and configuration-dependent, but here also $\alpha = 0.75$ consistently achieves good success rate.

Table 6.13: Success rate of CDGP and GPR, aggregated across the LIA benchmarks.

	CDGP				*GPR*			
	generational		*steadyState*		*generational*		*steadyState*	
	Tour	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*
1.0	0.71	0.64	0.47	0.69	0.54	0.47	0.35	0.56
0.75	0.82	**0.89**	0.77	0.83	0.51	0.61	0.36	0.53
0.5	0.67	0.88	0.63	0.73	0.47	0.63	0.17	0.47
0.25	0.63	**0.89**	0.49	0.71	0.45	0.62	0.13	0.42
0.0	0.61	0.88	0.49	0.68	0.44	0.63	0.10	0.37

Table 6.14: Average end-of-run size of T_c, aggregated across the LIA benchmarks.

	CDGP				*GPR*			
	generational		*steadyState*		*generational*		*steadyState*	
	Tour	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*
1.0	37	87	28	75	550	504	1967	1330
0.75	388	414	706	1202	2383	787	10305	5687
0.5	895	541	2196	2679	2871	862	14270	7474
0.25	1140	539	2939	3031	3202	885	16154	9073
0.0	1457	533	3232	3274	3222	876	**18799**	11006

Table 6.15: Average runtime of CDGP and GPR (in seconds), aggregated across the LIA benchmarks.

	CDGP				*GPR*			
	generational		*steadyState*		*generational*		*steadyState*	
	Tour	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*
1.0	687	790	1137	701	940	1060	1279	934
0.75	471	**354**	573	451	960	816	1291	994
0.5	852	416	925	724	1080	778	1585	1130
0.25	947	437	1162	759	1120	801	1657	1252
0.0	976	440	1155	816	1131	790	1675	1329

The above observations regarding the number of generated tests are supported by Table 6.14, where we can see that indeed the number of created tests is much higher for low values of α. We can also notice that, for the same value of α, GPR generates an order of magnitude more tests than CDGP. This is caused by the redundancy of counterexamples found by the solver, since independent calls to the solver are likely to produce the same counterexamples, which CDGP does not guard against; GPR, on the other hand, generates tests randomly and thus "collisions" are rather rare. Additionally, we can observe that GPR with tournament selection produces on average many more tests than with lexicase selection. This, however, might be simply the effect of lexicase selection itself being more costly than tournament selection, and thus resulting in fewer algorithm iterations overall (which is supported by the data regarding number of iterations collected during experiments). Interestingly, this does not seem to be the case for CDGP, because for steady state configurations the number of collected tests for tournament and lexicase configurations seems to be comparable for the most part.

As for the differences between generational and steady state configurations, it seems that generational configurations were better for basically all setups. In Table 6.14 we can see that while steady state configurations tend to generate much more tests than their generational counterparts, this is not followed by the improvement in success rate. We can speculate that the initial portion of 10 tests usually obtained by the generational configurations during initialization forces solutions to approach some minimal level of quality before more tests are generated, while steady state configurations start from blank

slate and generate many redundant non-interesting tests, leading to more time wasted on verification.

Lexicase selection, which in many past experiments proved to be superior [83], here also seems to dominate. This can be seen more clearly in Table 6.13, where we can see that lexicase selection is almost always on average better than the corresponding tournament configurations. The configurations of CDGP with generational mode of population replacement, lexicase selection, and $\alpha \leq 0.75$ (`CDGP/G/Lex/`$\alpha \leq 0.75$) boast both the biggest success rate over all configurations, and the shortest runtime (Table 6.15).

Statistical analysis

For a statistical analysis of the results in Tables 6.11 and 6.12, Friedman's test for multiple achievements of multiple subjects [91, 154] was used. Friedman's test was conducted separately for CDGP and GPR configurations, with 20 configurations on 16 benchmarks per each test. The obtained p-values were very small, respectively $1.0 \cdot 10^{-33}$ (CDGP) and $4.11 \cdot 10^{-25}$ (GPR), meaning that in each case at least one variant was significantly better than some other.

In order to find the pairs of configurations with significantly different effectiveness (success rate), the Wilcoxon-Nemenyi-McDonald-Thompson post-hoc test (also known as the Nemenyi post-hoc test) [86, 154] was used. As a result, complicated graphs of inter-configuration significance were obtained. Rather than presenting them in full, we will provide here only the most interesting general observations:

- `CDGP/G/Lex/`$\alpha \leq 0.75$ configurations dominated all `CDGP/*/Tour/`$\alpha \neq 0.75$ configurations. `CDGP/*/*/0.75` configurations were never dominated in this comparison and dominated several other configurations.
- `GPR/G/Lex/`$\alpha \leq 0.75$ configurations dominated all `GPR/S/Tour/`$*$ and all other `GPR/*/*/0.0` configurations.

The best CDGP variant (`CDGP/G/Lex/0.25`) and the best GPR variant (`GPR/G/Lex/0.25`) were then compared using Wilcoxon signed-rank test [195], which yielded the p-value equal to 0.00097. Thus, after taking into account Tables 6.11 and 6.12, we may conclude that the best configuration of CDGP is significantly better than the best configuration of GPR. At the same time, however, the worst configuration of CDGP (`CDGP/S/Tour/1.0`) performed on most benchmarks worse than the best configuration of GPR, but Wilcoxon signed-rank test yielded a p-value 0.14, and thus the result is not statistically significant. It is clear, however, that parametrization (with selection, population replacement, etc.) has big impact on the efficiency of CDGP, which can be concluded from many configurations of GPR having higher average success rate and lower runtime than some of the weaker CDGP configurations.

6.6 Experiment 2: Evaluation of CDGP on SLIA benchmarks

The goal of the second experiment was to assess the effectiveness of CDGP on SLIA benchmarks (listed in Table 6.8). The experiment was similar to that conducted in [32],

Table 6.16: Success rate of **CDGP** configurations on the **SLIA benchmarks**. An empty cell means that the success rate was zero.

	generational										*steadyState*									
	Tour					*Lex*					*Tour*					*Lex*				
	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0
combine	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.98	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
combine-2	0.94	**1.00**	0.98	**1.00**	0.98	0.96	0.98	**1.00**	0.98	0.98	0.82	0.96	0.98	**1.00**	**1.00**	0.98	0.96	**1.00**	**1.00**	**1.00**
combine-3	0.92	0.96	0.96	**1.00**	0.96	0.98	0.96	**1.00**	**1.00**	0.96	0.86	0.90	0.96	0.98	**1.00**	**1.00**	0.96	**1.00**	**1.00**	**1.00**
combine-4	0.84	0.96	0.94	**1.00**	**1.00**	0.86	0.98	0.94	**1.00**	0.98	0.78	0.86	0.96	0.98	0.98	0.96	0.96	0.98	**1.00**	**1.00**
dr-name			0.10	0.26	0.22		0.08		0.06	0.06		0.08	0.14	**0.56**	**0.56**		0.10	0.28	0.22	0.46
firstname	0.72	0.64	0.76	0.88	0.96	0.42	0.56	0.58	0.46	0.48	0.44	0.44	0.74	0.96	**0.98**	0.50	0.58	0.68	0.74	0.82
initials		0.02	0.02		0.02					0.02		0.02		0.02	**0.12**		0.02	0.02	0.10	0.04
lastname	0.12	0.54	0.68	0.88	0.84	0.16	0.24	0.18	0.34	0.40	0.14	0.28	0.70	**0.94**	**0.94**	0.34	0.42	0.30	0.62	0.60
phone	0.32	**1.00**	**1.00**	**1.00**	**1.00**	0.42	**1.00**	**1.00**	0.98	**1.00**	0.74	**1.00**	**1.00**	**1.00**	**1.00**	0.78	**1.00**	0.98	**1.00**	**1.00**
phone-1	0.20	**1.00**	**1.00**	0.98	**1.00**	0.40	0.98	**1.00**	**1.00**	**1.00**	0.54	**1.00**	**1.00**	**1.00**	**1.00**	0.66	**1.00**	**1.00**	**1.00**	**1.00**
phone-2	0.08	**1.00**	0.98	0.92	0.96	0.14	0.94	0.92	0.96	0.86	0.10	0.96	**1.00**	0.98	0.96	0.20	0.92	**1.00**	0.96	0.94
phone-3		0.60	0.60	0.56	0.66		0.40	0.38	0.48	0.36		**0.84**	0.82	0.78	0.76		0.68	0.72	0.46	0.60
phone-4		0.02	0.02	0.04				0.02		0.02		**0.08**	0.04		0.02				0.04	0.04
Mean	0.40	0.67	0.70	0.73	0.74	0.41	0.62	0.62	0.64	0.62	0.42	0.65	0.72	0.78	0.79	0.49	0.66	0.69	0.70	0.73
Rank	17.12	9.23	9.00	7.81	8.08	16.58	12.73	11.50	11.31	11.31	17.69	10.85	9.04	6.38	5.15	14.23	11.04	8.31	6.50	6.15

with the same changes as in Experiment 1. The dimensions of the experiment are:

- Method: CDGP.
- Population replacement: generational (`G`), steady state (`S`).
- Selection operator: tournament with $k = 7$ (`tour`), lexicase (`lex`).
- Tests ratio α: 0.0, 0.25, 0.5, 0.75, 1.0.

Table 6.16 presents the success rates obtained by CDGP on SLIA benchmarks (as mentioned in Section 6.4.6, GPR was not feasible as a baseline for this family of benchmarks). We notice that, at the first glance, effectiveness of CDGP was fairly similar across all configurations. Certain benchmarks, like for example the `combine` family, are very simple and were solved by many configurations in every run. Some other benchmarks, like for example the `phone` family, proved to be very hard for CDGP configurations with $\alpha = 0$, but it was simple for other configurations (at least `phone`, `phone-1`, and `phone-2` were). `dr-name` benchmark was solved in more than 50% of runs only by steady state tournament selection configurations with $\alpha \geq 0.75$, and they also proved to be the best overall when success rates and runtimes were averaged across all benchmarks (Table 6.17 and Table 6.19).

We can observe in Table 6.17 that tournament selection configurations tended to be better than their lexicase counterparts, which is contrary to the results obtained on the LIA benchmarks. This suggests that the characteristics of a particular benchmark, or family of benchmarks, may play an important role in how the evolutionary search conducted by CDGP proceeds. Despite lexicase selection's efficiency on uncompromising problems [83], for the SLIA benchmarks it happened to be worse.

Other observation is that the success rate seems to consistently rise with the value of α. This may be caused by a much longer time required by the solver to solve string queries as compared to integer queries, and, in the presence of time limit, reducing the number of such queries is more important than providing more test cases. In Table 6.19 we can observe that average runtimes have a similar tendency, i.e., they are lower with the higher α values. To account for the impact of failed runs on the average runtime, in Table 6.20

Table 6.17: Success rate of CDGP, aggregated across the SLIA benchmarks.

	CDGP			
	generational		*steadyState*	
	Tour	*Lex*	*Tour*	*Lex*
1.0	0.74	0.62	**0.79**	0.73
0.75	0.73	0.64	0.78	0.70
0.5	0.70	0.62	0.72	0.69
0.25	0.67	0.62	0.65	0.66
0.0	0.40	0.41	0.42	0.49

Table 6.18: Average end-of-run size of T_c, aggregated across the SLIA benchmarks.

	CDGP			
	generational		*steadyState*	
	Tour	*Lex*	*Tour*	*Lex*
1.0	12	14	4	7
0.75	20	21	8	15
0.5	48	37	57	52
0.25	62	42	116	88
0.0	95	76	**212**	149

Table 6.19: Average runtime of CDGP (in seconds), aggregated across the SLIA benchmarks.

	CDGP			
	generational		*steadyState*	
	Tour	*Lex*	*Tour*	*Lex*
1.0	689	903	**467**	589
0.75	717	891	496	617
0.5	815	918	648	686
0.25	870	919	783	729
0.0	1310	1261	1234	1072

Table 6.20: Average runtime of *successful* runs of CDGP (in seconds), aggregated across the SLIA benchmarks.

	CDGP			
	generational		*steadyState*	
	Tour	*Lex*	*Tour*	*Lex*
1.0	295	364	**121**	143
0.75	321	369	139	117
0.5	384	371	193	184
0.25	417	389	230	181
0.0	561	488	443	327

we present the average runtimes of successful runs only, and we can see that those times also tend to go down with higher values of α.

The success rates of generational and steady state configurations do not vary by much, but steady state configurations seem to be slightly better, and their runtimes noticeably shorter. Similarly as before, the simplest explanation could be the number of created test cases, however for $\alpha \leq 0.5$ it actually is higher (Table 6.18) than for the corresponding generational configurations, while runtime is still shorter, suggesting that something else is at play here. It is possible, however, that this boost to runtime is simply a result of generational configurations conducting verification of all candidate solutions during initialization (since T_C is initially empty); this is supported by a more detailed investigation of the logs of the experiment, where easy `phone` variants were solved very quickly by steady state approaches, while generational ones took much more time because of having to process many more calls to the solver, which additionally were very costly for that particular benchmark (in contrast, `combine` family was easy for the solver to verify, and for those benchmarks the steady state population replacement does not give a clear runtime advantage).

Statistical analysis

Similarly as for LIA benchmarks, we apply Friedman's test for the statistical analysis of the results. For 20 configurations of CDGP on 13 benchmarks the test yielded a p-value $8.97 \cdot 10^{-16}$, so we can infer that there is at least one pair of significantly different CDGP configurations.

After using Nemenyi post-hoc test, the following observations can be made:

- Only the four `CDGP/*/*/0.0` configurations were dominated in this comparison, and there was only one configuration that managed to dominate them all at once: `CDGP/S/Tour/1.0`.

- `CDGP/S/Lex/`$\alpha \geq 0.5$, `CDGP/G/Tour/`$\alpha \geq 0.75$, and `CDGP/S/Tour/0.75` configurations dominated all `CDGP/*/*/0.0` configurations, excluding `CDGP/S/Lex/0.0`.

These results support the earlier observation that very low values of α are problematic on the SLIA benchmarks.

6.7 Experiment 3: Comparison of CDGP with exact program synthesis algorithms

The goal of the third experiment was to compare CDGP, a heuristic program synthesis algorithm, with the exact approaches from the literature. For the purpose of comparison, two such algorithms were selected: EUSolver [11], and CVC4 [169]. The set of benchmarks comprises all the LIA and SLIA benchmarks from Experiment 1 and Experiment 2.

6.7.1 Tested algorithms

Two configurations of CDGP and one of GPR were selected for this experiment:

- **CDGP$_{\text{LIA}}$** – the best variant of CDGP for LIA benchmarks, as indicated by Experiment 1, i.e., `CDGP/G/Lex/0.25`.
- **CDGP$_{\text{SLIA}}$** – the best variant of CDGP for SLIA benchmarks, as indicated by Experiment 2, i.e., `CDGP/S/Tour/1.0`.
- **GPR$_{\text{LIA}}$** – the best variant of GPR for LIA benchmarks, as indicated by Experiment 1, i.e., `GPR/G/Lex/0.25`.

The results of these CDGP/GPR configurations were taken from the experiments conducted in Sections 6.5 and 6.6.

As for the exact program synthesis algorithms, we selected the following:

- **EUSolver**[6] [11] – an approach based on enumerating both terms and predicates from the grammar until it is determined by a decision learning algorithm that they cover all the collected points (counterexamples), the set of which is initially empty. Then, the candidate program is constructed based on the created decision tree (the process called *unification*) and verified. If the verification succeeds, a correct solution has been found and the search terminates; otherwise, a counterexample is added to the set of points and the enumeration procedure continues. Since a conditional instruction (or its semantic equivalent) is required to perform the unification, EUSolver is limited to the problems containing this construct in their grammars.
- **CVC4**[7] [169] – an SMT solver with the capability of solving SyGuS problems using refutation-based synthesis, equipped with efficient counterexample-guided techniques for quantifier instantiation. In this approach, the synthesis task is expressed using a logical formula with a universal quantifier over possible programs. This for-

[6] https://bitbucket.org/abhishekudupa/eusolver/, master branch, commit `cedce0c` from 16th June 2020.
[7] CVC4 1.8, https://github.com/CVC4/CVC4, released on 19th June 2020.

Table 6.21: Success rate of the exact program synthesis algorithms and the best CDGP/GPR configurations.

	EUSolver	CVC4	$CDGP_{LIA}$	$CDGP_{SLIA}$	GPR_{LIA}
CountPos2	**1.00**	**1.00**	**1.00**	0.76	0.80
CountPos3	**1.00**	**1.00**	0.96	0.14	0.54
CountPos4	**1.00**	**1.00**	0.38	0.00	0.14
IsSeries3	**1.00**	**1.00**	0.84	0.58	0.02
IsSeries4	**1.00**	**1.00**	0.44	0.30	0.00
IsSorted4	**1.00**	**1.00**	**1.00**	0.68	0.54
IsSorted5	**1.00**	**1.00**	**1.00**	0.52	0.08
Max4	**1.00**	**1.00**	**1.00**	0.88	0.98
Median3	**1.00**	**1.00**	**1.00**	0.66	0.96
Range3	**1.00**	**1.00**	0.98	0.64	0.98
Search2	**1.00**	**1.00**	**1.00**	0.70	0.98
Search3	**1.00**	**1.00**	**1.00**	0.46	0.96
Search4	**1.00**	**1.00**	0.96	0.06	0.84
Sum2	**1.00**	**1.00**	**1.00**	0.90	**1.00**
Sum3	**1.00**	**1.00**	**1.00**	0.22	0.76
Sum4	**1.00**	**1.00**	0.68	0.00	0.40
combine	-	**1.00**	**1.00**	**1.00**	-
combine-2	-	0.00	0.98	**1.00**	-
combine-3	-	0.00	0.96	**1.00**	-
combine-4	-	0.00	**0.98**	**0.98**	-
firstname	-	**1.00**	0.56	0.98	-
initials	-	0.00	0.00	**0.12**	-
lastname	-	**1.00**	0.24	0.94	-
phone	-	**1.00**	**1.00**	**1.00**	-
phone-1	-	**1.00**	0.98	**1.00**	-
phone-2	-	**1.00**	0.94	0.96	-
phone-3	-	**1.00**	0.40	0.76	-
phone-4	-	**1.00**	0.00	0.02	-

Table 6.22: Average runtime (in seconds) of *successful* runs of the exact program synthesis algorithms and the best CDGP/GPR configurations. 'n/a' in a cell means that there were no successful runs.

	EUSolver	CVC4	$CDGP_{LIA}$	$CDGP_{SLIA}$	GPR_{LIA}
CountPos2	0.3	**0.0**	60	342	97
CountPos3	0.5	**0.0**	559	444	526
CountPos4	1.2	**0.0**	841	n/a	925
IsSeries3	0.3	**0.0**	335	489	1223
IsSeries4	0.3	**0.0**	919	633	n/a
IsSorted4	0.3	**0.0**	132	368	443
IsSorted5	0.3	**0.0**	281	315	899
Max4	0.4	**0.0**	41	224	50
Median3	0.5	**0.0**	77	478	42
Range3	0.7	**0.0**	276	393	112
Search2	0.3	**0.0**	83	466	37
Search3	0.3	**0.0**	85	545	59
Search4	0.3	**0.0**	335	755	209
Sum2	0.2	**0.0**	36	122	103
Sum3	0.3	**0.0**	225	862	372
Sum4	0.3	**0.0**	997	n/a	738
combine	-	**0.1**	17	1.4	-
combine-2	-	n/a	56	**3.0**	-
combine-3	-	n/a	85	**5.0**	-
combine-4	-	n/a	136	**9.5**	-
firstname	-	**2.1**	756	139	-
initials	-	n/a	n/a	**375**	-
lastname	-	**8.4**	768	167	-
phone	-	**1.0**	624	4.3	-
phone-1	-	**2.9**	544	**2.9**	-
phone-2	-	**1.3**	641	8.5	-
phone-3	-	**63**	1054	688	-
phone-4	-	**15**	n/a	796	-

mula is first negated, and then the solver tries to prove its unsatisfiability. The final solution is constructed from the proof of unsatisfiability.

These algorithms were run only once for each benchmark, so their success rate is either 1.0 or 0.0.

6.7.2 Results and discussion

EUSolver and CVC4 outclass, by a far margin, CDGP on the LIA family of benchmarks – each of them not only solves every task perfectly (Table 6.21), but also almost instantly (Table 6.22). We speculate that the main problem of CDGP is the amount of queries to the solver, which the exact methods avoid.

For the SLIA benchmarks, only a comparison with CVC4 could be made, since in the benchmarks' grammar there was no conditional instruction, and thus EUSolver could not employ the unification that it relies on. On this family of benchmarks, CDGP proved to be more competitive. While CVC4 still solved 7 out of 13 benchmarks almost instantly, it did not manage to find any solution in the given time limit of 30 minutes for the rest of the benchmarks, while $CDGP_{SLIA}$ managed to solve all of them, although in case of `initials` and `phone-4` only in a few runs.

Interestingly, CDGP produces in many cases much shorter programs than the exact approaches (Table 6.23). GP is known to produce messy, bloated programs, and in our

Table 6.23: Average size of solutions (number of nodes in the simplified expression tree) found by the exact program synthesis algorithms and the best CDGP/GPR configurations. 'n/a' in a cell means that there were no successful runs.

	EUSolver	CVC4	$CDGP_{LIA}$	$CDGP_{SLIA}$	GPR_{LIA}
CountPos2	40	**29**	42	34	45
CountPos3	177	87	77	**47**	80
CountPos4	662	**25**	67	n/a	69
IsSeries3	**10**	12	57	30	19
IsSeries4	**18**	22	83	33	n/a
IsSorted4	**13**	28	39	30	51
IsSorted5	**16**	36	51	29	48
Max4	37	76	46	**35**	53
Median3	114	167	60	**38**	58
Range3	422	568	86	**36**	68
Search2	**19**	88	30	25	26
Search3	37	187	40	**28**	43
Search4	61	309	59	**29**	55
Sum2	**10**	17	20	16	26
Sum3	41	70	47	**30**	58
Sum4	93	160	**66**	n/a	**66**
combine	-	5	**4**	5	-
combine-2	-	n/a	**9**	10	-
combine-3	-	n/a	**8**	**8**	-
combine-4	-	n/a	10	**9**	-
firstname	-	**8**	10	9	-
initials	-	n/a	n/a	**21**	-
lastname	-	**11**	15	14	-
phone	-	**4**	6	7	-
phone-1	-	**4**	7	6	-
phone-2	-	**4**	7	7	-
phone-3	-	8	9	**7**	-
phone-4	-	**7**	n/a	17	-

implementation of CDGP we automatically perform a simplification step after the search terminates using the simplification feature offered by Z3. For fairness of comparison, program sizes in Table 6.23 were computed after all the found solutions were subjected to the simplification[8] using Z3. The inspection of the solutions generated by EUSolver and CVC4 revealed that they overuse conditional instructions in order to cover all possible cases instead of finding more streamlined solutions. Though the SyGuS benchmarks do not provide test sets since the tasks are fully defined by the specification, we hypothesize that, in the case of partial specification, the programs synthesized by CVC4 and EUSolver would generalize poorly compared to those produced by CDGP.

6.8 Conclusions

In this chapter, we have described CDGP, a novel approach for combining evolutionary search and formal reasoning to synthesize programs satisfying formal specifications. CDGP relies on collecting counterexamples from failed verification attempts, and uses them to create test cases, which then provide search gradient for GP. The computational experiments showed that CDGP is capable of solving many specification-based problems in the domains of integer arithmetic and text strings manipulation. A comparison with GPR, a baseline variant of CDGP which discards counterexamples and generates tests randomly,

[8]Simplification in Z3 sometimes makes programs longer as it converts them to a canonical form (e.g., '$<$' are replaced with the negation of '$\geq$'). In Table 6.23 are presented the sizes of the shorter of the two programs: before or after the simplification.

showed that counterexamples used by CDGP are more beneficial for search than the test cases generated randomly.

A comparison with existing exact approaches from the literature shows that much work still needs to be done for evolutionary search heuristics to be competitive in the domain of program synthesis from specification, especially for the LIA class of problems. The strategy represented by EUSolver, i.e., enumeration of simple terms and their unification using a conditional statement, proves to be very effective in practice. However, it also requires that a conditional statement is present in the grammar, which was the reason why we could not use EUSolver for SLIA benchmarks; CDGP has no such constraints. CVC4, a logical deduction based approach, is unable to solve some SLIA benchmarks which CDGP does and often produces very long solutions for LIA benchmarks, but that aside it generally outclasses CDGP in terms of speed.

Chapter 7

Counterexample-Driven Symbolic Regression

In the previous chapter, we investigated the possibility of using counterexamples to successfully solve the task of program synthesis from complete formal specification using GP. In this chapter, we consider what needs to be modified in CDGP in order to handle real-valued symbolic regression problems in the presence of noise, with a task specification consisting of both a training set of examples and formal constraints. In order to differentiate this new approach from CDGP, we dubbed it *Counterexample-Driven Symbolic Regression* (CDSR).

This chapter is based on the material published previously in [31], which was created in collaboration with Krzysztof Krawiec.

7.1 Introduction

Symbolic regression problems typically involve as a starting point only a set of input-output examples representing the expected behavior of the modeled system. A *regressor*, by which we mean an executable artifact analogous to a classifier but predicting real values instead of labels, is expected to both minimize an error on the training set and generalize well on inputs not seen during training. This aspect of generalization places symbolic regression between machine learning problems rather than pure optimization problems. In machine learning, the generalization power of an algorithm is typically evaluated in a *quantitative* fashion as an aggregated error on examples not seen during training (*test set*). This is a natural approach, since without additional knowledge or assumptions about the data, e.g., consistent patterns between outputs for different inputs, the regressor's behavior on the test set is the only available measure of generalization.

However, if we consider a scenario in which formal constraints expressing these complex patterns in data are part of a problem specification, then we can additionally evaluate generalization in a *qualitative* fashion as a level to which the constraints are satisfied by the synthesized regressor. We use the term "formal constraints" to emphasize that they are expressed in some formal language (e.g., first-order logic), as opposed to input-output examples.

Theoretically, we can imagine a scenario in which formal constraints are generated

automatically from input-output examples, possibly in addition to some prior formalized domain knowledge. For example, from the following set of training examples of the form (in, out): $\{(0,1),(1,2),(2,3)\}$, we could infer that a "good" solution would always return a larger number than the input. However, there are many (in fact infinitely many) constraints that can be conceived in this way, and many of them, e.g., $f(in) \leq 3$, would be considered trivial at best, and unfounded at worst. This illustrates a significant challenge of choosing an appropriate frame of reference, i.e., the properties which can be reasonably inferred from data.

In many practical scenarios, however, those properties can be supplied by the user, who either knows beforehand that they are true of the system in question (domain knowledge), or simply finds them convenient or beneficial. For instance, a regression model that is meant to serve as a controller may be not allowed to output a negative value, as that would damage some hardware component; in medicine, the predicted dose of an active substance may need to monotonously increase with patient's weight, etc. In this chapter, we will focus solely on the scenario that formal constraints are provided by a user of the synthesis system, and we will not attempt to infer additional constraints (either from data or the user-provided constraints) or modify the existing ones.

In optimization problems, constraints often prove to be very helpful in finding good solutions – a good example may be the simplex algorithm [50] for the linear programming problem, which utilizes the fact that optimal solution, if it exists, must lie on the boundary spanned by the constraints.[1] While generate-and-test metaheuristics are unable to use constraints to such an extent and still be applicable to a large set of different problems, we still speculate that taking constraints into account during search can be beneficial in terms of generalization. Furthermore, we hypothesize that standard constraint-agnostic regression methods, while generalizing well in the *quantitative* sense (i.e., related to the aggregation of multiple point-wise classifier interactions on test set), does not fare that well in terms of the *qualitative* generalization based on the expected high-level properties of a final solution. In order to investigate these hypotheses, we propose an extension of CDGP called CDSR (Counterexample-Driven Symbolic Regression) and compare it with state of the art machine learning regression algorithms.

This chapter is organized as follows. First, in Section 7.2 we define the task of *symbolic regression with formal constraints* (SRFC), and in Section 7.3 provide a range of examples of formal constraints of practical relevance. After the background is properly set up, we proceed to the description of CDSR (Section 7.4), a method for solving SRFC tasks that relies on genetic programming (GP) combined with formal verification. After that, we describe two computational experiments conducted in this work: verifying effectiveness of the state of the art constraint-agnostic regression algorithms (Section 7.5), and analyzing different variants of CDSR (Section 7.6). Section 7.7 concludes the chapter.

[1] It must be noted, however, that constraints are an essential part of the definition of the linear programming problem. Still, we can use them directly to find the solution, instead of, e.g., using generate-and-test methods.

7.2 Symbolic Regression with formal constraints

Russell and Norvig [172, p. 695] define the task of *supervised learning* as follows[2]:

Definition 7.2.1. (*Supervised learning*) Given a training set T of n example input-output pairs

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n),$$

where each y_j was generated by an unknown function $y = f(x)$, discover a function h that approximates the true function f.

If f returns values from a finite set of discrete labels, then the task is called *classification*. On the other hand, if f returns a number, then the task is called *regression*. Regression is typically approached by deciding on some parameterized family of models H, and then selecting, by means of tuning these parameters, such $h \in H$ that best fit the data. For example, in linear regression it is assumed that H is a set of all linear combinations of inputs, and the least-squares method [172] is used to find appropriate coefficients, thus determining h. In contrast, *symbolic regression* (SR) is an approach in which the structure of a model is not assumed a priori, but is being optimized during learning together with the parameters. This is usually achieved by representing the model as an explicit mathematical formula and gradually modifying it to better fit the data – a prominent example of this strategy is GP [103]. This freedom of structure is especially important when a knowledge about the problem is insufficient to narrow down the model's structure. Models produced by symbolic regression algorithms are also easy to interpret, which puts symbolic regression algorithms as a potentially important tool for interpretable machine learning [58, 141]. Applications of symbolic regression are numerous, and include, among others, materials science [192], chemical sciences [191], and discovery of physical laws from observations [173, 190].

In our previous work [31], we have defined *Symbolic Regression with Formal Constraints* (SRFC), a variant of symbolic regression task in which, alongside a set of example input-output pairs, there is also given a set of formal constraints which the function h is supposed to satisfy. This task falls into a broader category of *supervised learning with constraints* [89], which can be defined as follows:

Definition 7.2.2. (*Supervised learning with constraints*) Given a training set T of n example input-output pairs

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n),$$

where each y_j was generated by an unknown function $y = f(x)$, and a set of constraints C, discover a function h that approximates the true function f while satisfying all constraints in C.

Each constraint in C is a logical formula that should be satisfied by the model h for an (often infinite) subset of f's domain, for example $\forall x : h(x) \geq 0$, or $\forall x : h(x) = h(-x)$. Technically, nothing prevents a constraint from defining function's output for a single input

[2] We introduced minor notation changes to facilitate the discussion in this chapter.

(e.g., $h(3) = 7$), making it similar to the input-output pairs in T, but differing in being a *hard constraint* not allowing h for any error on that input. When $C = \emptyset$, supervised learning with constraints reduces to the classical supervised learning.

We are now ready to define *Symbolic Regression with Formal Constraints (SRFC)* task:

Definition 7.2.3. *Symbolic Regression with Formal Constraints (SRFC)* is a special case of supervised learning with constraints (Definition 7.2.2), where all y_n are real numbers (i.e., it is a regression task). Additionally, the "symbolic" part requires that an approximating function h (model) is chosen from a set $\mathcal{M}$ of all possible mathematical expressions over a certain formal grammar.

This definition of SRFC is similar in spirit to that of Syntax-Guided Synthesis (SyGuS) [10], where solutions are required to be constructed based on the provided formal grammar. Contrary to SyGuS, however, in SRFC there is a set of training examples, which are not necessarily supposed to be fitted perfectly (as this may lead to overfitting), but rather serve as a basis for the discovery of the model that explains them most adequately. The explicit mention of "formal grammar" may seem unnecessarily limiting, but it is intended to serve as a sort of boundary between symbolic and non-symbolic (or sub-symbolic) methods. In neural networks, which we would like to *not* be classified as a symbolic regression method, usually weights of a fixed network architecture are optimized, and thus no one considers this learning paradigm to be a search over some formal grammar. However, this does not change the fact that theoretically a formal grammar representing a family of functions learnable by a particular neural network could be defined. Perhaps we should, after [155], require a synthesized symbolic expression to be concise, and thus exclude neural networks on the grounds of their complex representation. We acknowledge that our distinction here is not perfectly precise, but we leave it to the future authors, since the definitions of symbolic regression that we have found in the literature are rather vague regarding their exact boundaries.

7.3 Examples of formal properties of practical relevance

In this section, we present a number of formal constraints (properties) that we find likely to occur in practical applications of SRFC. The word "constraint", while correct, brings to mind mostly reducing the number of valid assignments of values to variables. We will occasionally use the word "property" instead, to emphasize the qualitative aspect the final solution is expected to possess.[3] For each property, we discuss plausible usage scenarios and provide its specification in SMT-LIB [19, 21], the standard language for communication with SMT solvers. In the following, `f` denotes a function that should meet the constraint in question.

Symmetry with respect to arguments. Many multivariate models are expected to be symmetric with respect to the order of their arguments. Examples include the equivalent

[3] Of course, the only way of exhibiting a certain property is by adhering to constraints preventing situations in which the property is not held; this is only a change in perspective.

resistance of a number of electrical resistors (chained or arranged in parallel), or the force of gravity that remains the same if the interacting masses are swapped. In SMT-LIB, this property can be expressed as:

```
(assert (= (f x y) (f y x)))
```

In SRFC, this assertion would be included in the set of constraints C, while the tests/examples would be placed in T. However, let us emphasize again that the assertion requires `f` to meet the constraint for *all* arguments `x` and `y` satisfying the precondition, not only for those present in T. When a CDSR run with such a constraint in C ends with success, then the synthesized model is guaranteed to be symmetric with respect to its arguments.

Symmetry with respect to argument's sign. It is sometimes desirable to require models to be even functions ($f(x) = f(-x)$) or odd functions ($-f(x) = f(-x)$). For instance in classical physics, the direction of the restoring force of a spring depends on the direction of displacement, which implies that the dependency in question is an odd function $F(x) = -kx$, where k is the spring constant. Expressing such properties is straightforward:

```
(assert (= (f x) (f (- x))))
```

Such symmetry may be also useful when constraining multivariate models, where it may be selectively applied to individual variables. A bivariate model `f(x,y)` can be demanded to be even with respect to `x` with the following assertion:

```
(assert (= (f x y) (f (- x) y)))
```

Range. There are multiple scenarios in which domain knowledge excludes certain ranges of values from f's codomain. In classical physics, mass cannot be negative and velocity cannot exceed the speed of light. In econometrics, employee's wage cannot be negative. In medicine, it may not make sense to estimate patient's life expectancy to more than 120 years. The last of these examples can be expressed in SMT-LIB as:

```
(assert (<= (f x y) 120.0))
```

Monotonicity. Monotonicity is one of the most common properties expected from models induced from data. In transportation, for instance, the cost of delivery is almost always a monotonically increasing function of distance (or time). Such a constraint can be encoded as:

```
(assert (forall ((x Real)(x1 Real))
  (=> (> x1 x)  (> (f x1) (f x)))
))
```

Using a quantifier that way has a negative consequence that no counterexamples will be generated since the variables are bounded, and the method introduced in this chapter (CDSR; Section 7.4) requires them for successful operation. Including such a constraint is, however, still beneficial, because it stops CDSR from terminating with a program not satisfying it. In CDSR_p variant (to be introduced in Section 7.4.1), where satisfaction of

each individual constraint is represented in the solution's fitness vector, such constraints have even more substantial impact on search.

Convexity/concavity. Convex models are often sought after, because they can be later efficiently optimized. Convexity of a univariate function can be defined using Jensen's inequality:

$$\forall_{x,y,t\in[0,1]}\ f(tx + (1-t)y) \leq tf(x) + (1-t)f(y).$$

Similarly as for monotonicity, convexity constraint requires a universal quantifier:

```
(assert (forall ((t Real)(x Real)(x1 Real))
  (=> (and (>= t 0.0) (<= t 1.0))
    (<= (f (+ (* t x) (* (- 1.0 t) x1)))
        (+ (* t (f x)) (* (- 1.0 t) (f x1)))))
))
```

Changing this constraint to concavity would simply require replacing `<=` with `>=` in the quantified formula; replacing it with `<` would demand the function to be strictly convex.

Slope. In many applications, it may be known that the rate of change of model's output with respect to its input cannot exceed certain threshold. For instance, a body free-falling in Earth's gravitational field cannot accelerate faster than 9.81 m/s^2. This kind of constraints can be expressed with the derivative. However, in order to compute derivative in a way supported by SMT solver, we need to use an approximation:

$$f'(x) = \frac{f(x+\epsilon) - f(x)}{\epsilon},$$

where ϵ is some very small number.

In the following SMT-LIB example, we assume that the expected derivative of a function f for $x = 1.0$ is 2.0, $\epsilon = 10^{-6}$, and a tolerance for error is 0.001:

```
(define-fun df ((x Real)) Real
    (/ (- (f (+ x 0.000001)) (f x)) 0.000001))
(assert (=> (= x 1.0) (<= (abs (- (df x) 2.0)) 0.001)))
```

Note that this constraint affects only the slope of `f` at point 1.0, while not determining the desired *value* of `f` at that point. Therefore, this requirement cannot be alternatively enforced by providing input-output tests in T that would implicitly constrain the slope, because such tests would also necessarily determine the value of f.

Discussion. The above list presents only the simplest and most common properties. Other examples include:

- periodicity: $f(x) = f(x + kT), k \in \mathbb{Z}$,
- additivity: $f(x + y) = f(x) + f(y)$,
- multiplicativity: $f(x \cdot y) = f(x) \cdot f(y)$.

Compound constraints can be easily created by combining elementary constraints with logical conjunction. Also, all above properties can be defined either globally (i.e., in the entire domain of the considered function) or locally (i.e., in an interval, at a given point, or otherwise constrained part of function's domain).

7.4 Counterexample-Driven Symbolic Regression

In this section, we describe the modifications we have made to CDGP (Section 6.3) in order to handle SRFC problems (Section 7.2). We dubbed the resulting algorithm *Counterexample-Driven Symbolic Regression* (CDSR). The overview of changes in CDSR is presented in Table 7.1.

The following challenges prevent the use of CDGP in its original form:

1. **Presence of noise** – input-output examples are represented in our SyGuS benchmarks as equality constraints (i.e., $f(x) = y$), and if any noise has disturbed x or y, then the equality will not hold even for the optimal solution. Adding a tolerance range to the equality is limiting for many types of noise, for example Gaussian noise can happen to be, although with a very small probability, arbitrarily high. Additionally, f may magnify any noise present in x.
2. **Formal verification criterion** – in CDGP, whether a solution undergoes formal verification or not depends on the parameter α representing the ratio of tests which must be passed in order for the solution to be verified. However, in the presence of noise even a perfect candidate solution will practically never have no error on a test, and thus the notion of "passing a test" needs to be clarified.
3. **Generalization** – CDGP works with a formal specification fully describing the behavior of a target function, which means that generalization beyond "training set" is not a concern. In contrast, formal constraints in CDSR perform a rather auxiliary role, because the expected behavior of the function is mostly defined by the provided input-output examples (training set). Thus, additional mechanisms need to be introduced in order to bias GP towards finding interesting well-generalizing solutions for points beyond the training set, instead of overfitted solutions that are perfect on the training set.

Challenge 1. was solved by seeding the internal set of test cases T_C (Section 6.3) with all input-output examples in the specification, instead of initializing T_C as an empty set and including the input-output examples as constraints for formal verification. This at first may seem like a natural solution without any negative side effects, but there are many situations in which it is a combination of tests and constraints that uniquely defines a function or family of functions. For example, when a function f is linear, i.e., $f(x_1, \ldots, x_n) = \sum_i a_i x_i + b$, and an input consisting of zeros is included in the training set with the expected output being b, then as a result the set of all straight lines is constrained into a much smaller subset. However, when this input-output example is not taken into account during formal verification, then an incorrect linear function g might be decided to satisfy the constraints, although at the cost of higher error on the training examples. This is, however, not that important in practice, since usually there are several training examples which will sufficiently penalize such solutions.

This leads us to challenge 2. and the formal verification criterion. We decided to use a threshold with the default of 5% of the target output of a given test. Setting this threshold too low may stop counterexamples from being generated, and setting it too high may result in too many counterexamples and a significant time used on their evaluation.

Table 7.1: A summary of differences between CDGP and CDSR.

	CDGP	**CDSR**
Verification criterion	the ratio α of passed complete and incomplete tests	the ratio α of passed incomplete tests and complete tests with an error under a given threshold (5% of the expected output)
Validation set	no	yes
Termination criterion (besides 1800 s time budget)	successful formal verification	lack of improvement on validation set for 25 generations
Formal constraints part of fitness	no	yes in CDSR_p variant

Finally, challenge 3. was solved by employing a validation set and the early stopping technique [70, 144] used often during training of neural networks, which stops the optimization process when the validation error of currently the best solution does not improve in a certain time window (in our experiments: 25 generations). A solution with the lowest error on validation set is also remembered and returned as a final solution of a CDSR run.

7.4.1 CDSR with properties in fitness (CDSR_p)

We have also considered additional ways to increase the impact of the expected properties (formal constraints) on search in CDSR. This led us to the creation of a new variant of CDSR, which we dubbed CDSR_p (*CDSR with properties in fitness*). In CDSR_p, the information regarding satisfaction of individual constraints is stored directly in the candidate program's fitness vector. This gives evolution an additional incentive to produce programs that exhibit the desired properties. The satisfaction of individual constraint is checked by means of formal verification in a process practically identical to that described in Section 6.3.1, with the exception that only a single constraint is included in the verification query. Thus, the solver verifies the program's correctness on that constraint, and then to the candidate solution's fitness vector is appended either 0 (if constraint was satisfied) or 1 (if it was *not* satisfied) for each constraint.

CDSR_p has one parameter w_c representing a "weight" that a constraint has in the fitness vector. The weight impacts the fitness vector by duplicating all constraints' entries in the vector w_c times. For example, if the first constraint was satisfied and $w_c = 5$, then the fitness vector of that solution would look like this:

$$[0, 0, 0, 0, 0, \{\text{other constraints}\}, \{\text{errors on training set}\}]$$

We will now examine the impact of w_c depending on the selection algorithm used. In ϵ-lexicase [110] (a variant of lexicase selection for regression problems; described in Section 7.6.2), the impact of every constraint's entry in the fitness vector is the same as that of a single input-output test. Thus, by setting w_c appropriately, we can achieve the desired trade-off between constraints and tests during evolution. In the example above,

the satisfaction of the first constraint is effectively worth as much as passing five input-output tests. The situation is, however, more complex for tournament selection, which in our implementation of CDSR ranks the candidate solutions on the basis of aggregation of tests by means of MSE. Depending on the magnitude of errors in the benchmark, the impact of an individual constraint can be either very big or very small, but still an appropriate w_c setting should allow to achieve a desired average, benchmark-dependent, level of trade-off between the importance of passing input-output examples and meeting formal constraints.

7.5 Experiment 1: Standard regression algorithms in the presence of formal constraints

In the first experiment, we will examine several commonly used constraint-agnostic regression algorithms from the literature in order to see how well they generalize in the qualitative sense, i.e., constraints satisfied by the synthesized model (regressor). As these algorithms accept as input only a training set of examples, the only way for them to produce a model that exhibits the desired properties is by means of induction from examples.

7.5.1 Regression algorithms and machine learning framework

The algorithms selected for this experiment, together with a grid of hyperparameters they were tested on, are presented in Table 7.2. We mirrored the selection of algorithms and their hyperparameters from the work by Orzechowski et al. [147], and we have also used the open source framework they kindly shared[4].

Similarly as in [147], to compare different parameterizations of the algorithms we used 5-fold cross validation, which is also the default setting in the scikit-learn Python library. This means that all training examples are partitioned into 5 sets ("folds"), and there are 5 iterations of learning in which one of the folds, different each time, constitutes a validation set used to compute the regression error, and the other folds constitute the training data. The final quality of a single parameterization of the algorithm is then computed as an average validation error obtained in these 5 iterations. The final performance of a given algorithm (i.e., of its best parameterization) is determined as follows: we select the parameterization that produced the smallest average validation error, and that parameterization is tested on the test set for objective comparison with other regression algorithms. The above cross validation procedure is repeated 10 times for different partitioning of data into training set and test set, and the average test error is the final measure of quality of the regression algorithm.

One of the criteria used to evaluate the algorithms is the number of satisfied formal constraints. Normally, we would use an SMT solver to perform a full formal verification to determine whether a produced regressor satisfies a given constraint or not. However, this poses difficulties, because some of the regressors used in this study involve operations not supported by NRA logic in contemporary SMT solvers, for example logarithms or trigonometric functions. There are also practical challenges of reconstructing regressor's

[4] https://github.com/EpistasisLab/srbench

Table 7.2: The standard regression algorithms that we used in the experiment, and their hyperparameters which were tested. This table is reproduced from [147], since our experimental setup mirrored for the most part the one in that work; we have not used all algorithms tested there, and below are presented only those that we had.

Algorithm name	Parameter	Values
AdaBoostRegressor	'n_estimators'	$\{10, 100, 1000\}$
	'learning_rate'	$\{0.01, 0.1, 1, 10\}$
GradientBoostingRegressor	'n_estimators'	$\{10, 100, 1000\}$
	'min_weight_fraction_leaf'	$\{0.0, 0.25, 0.5\}$
	'max_features'	{'sqrt', 'log2', None}
KernelRidge	'kernel'	{'linear', 'poly', 'rbf', 'sigmoid'}
	'alpha'	$\{1e{-}4, 1e{-}2, 0.1, 1\}$
	'gamma'	$\{0.01, 0.1, 1, 10\}$
LassoLARS	'alpha'	$\{1e{-}04, 0.001, 0.01, 0.1, 1\}$
LinearRegression	default	default
MLPRegressor	'activation'	{'logistic', 'tanh', 'relu'}
	'solver'	{'lbfgs', 'adam', 'sgd'}
	'learning_rate'	{'constant', 'invscaling', 'adaptive'}
RandomForestRegressor	'n_estimators'	$\{10, 100, 1000\}$
	'min_weight_fraction_leaf'	$\{0.0, 0.25, 0.5\}$
	'max_features'	{'sqrt', 'log2', None}
SGDRegressor	'alpha'	$\{1e{-}06, 1e{-}04, 0.01, 1\}$
	'penalty'	{'l2', 'l1', 'elasticnet'}
LinearSVR	'C'	$\{1e{-}06, 1e{-}04, 0.1, 1\}$
	'loss'	{'epsilon_insensitive', 'squared_epsilon_insensitive'}
XGBoost	'n_estimators'	$\{10, 50, 100, 250, 500, 1000\}$
	'learning_rate'	$\{1e{-}4, 0.01, 0.05, 0.1, 0.2\}$
	'gamma'	$\{0, 0.1, 0.2, 0.3, 0.4\}$
	'max_depth'	$\{6\}$
	'subsample'	$\{0.5, 0.75, 1\}$

equivalent symbolic formula from the Python objects in the scikit-learn library. Our solution to these challenges was relaxing the notion of verification to that of *approximate verification*, in which for each constraint we checked whether it is satisfied for several points in the benchmark's inputs domain (Table 7.3) in a grid-like fashion. Because the verification of constraints of so many solutions[5] is costly, we used the following policy for assigning the number of points depending on the benchmark's arity: 41/variable (arity 1; 41 points in total), 11/variable (arity 2; 121 points in total), and 7/variable (arity 3; 343 points in total). Approximate verification allows us to simply use in the verification query a regressor's output instead of its symbolic equivalent formula, which is compensated by the need to perform such partial verification many times for different points, and the guarantees this method offers are also only partial.

This technique was motivated by an observation that "deceptive" cases are relatively rare in practice and, for example, if a symmetry constraint ($f(x, y) = f(y, x)$) is satisfied for 10 different pairs of values of x and y, then it is quite likely that it will be also satisfied for other pairs. Our observations during Experiment 2 (Section 7.6), where for CDSR runs we could compare the results of *approximate verifier* with those of a full verification,

[5] 50 (runs per configuration) × 12 (total different configurations) × 22 (total number of benchmarks with and without noise) = 13200. And we also need to verify all solutions produced by constraint-agnostic regression algorithms (2200).

Table 7.3: The SRFC benchmarks used in our experiments. $U(a, b)$ stands for a uniform distribution in range $[a, b]$, inclusive for a and b.

Benchmark	Arity	Solution	Training set	# constraints
gravity	3	$f(m_1, m_2, r) = \frac{6.674 \cdot 10^{-11} m_1 m_2}{r^2}$	$U(0.0001, 21)$	4
keijzer5	3	$f(x, y, z) = \frac{30xz}{(x-10) \cdot y^2}$	$U(-10, 11)$	3
keijzer12	2	$f(x, y) = x^4 - x^3 + \frac{y^2}{2} - y$	$U(-10, 11)$	6
keijzer14	2	$f(x, y) = \frac{8}{2+x^2+y^2}$	$U(-10, 11)$	4
keijzer15	2	$f(x, y) = \frac{x^3}{5} + \frac{y^3}{2} - y - x$	$U(-10, 11)$	3
nguyen1	1	$f(x) = x^3 + x^2 + x$	$U(-10, 11)$	3
nguyen3	1	$f(x) = x^5 + x^4 + x^3 + x^2 + x$	$U(-10, 11)$	3
nguyen4	1	$f(x) = x^6 + x^5 + x^4 + x^3 + x^2 + x$	$U(-10, 11)$	3
pagie1	2	$f(x, y) = \frac{1}{1+x^{-4}} + \frac{1}{1+y^{-4}}$	$U(-10, 11)$	3
res2	2	$f(r_1, r_2) = \frac{r_1 r_2}{r_1 + r_2}$	$U(0.0001, 21)$	3
res3	3	$f(r_1, r_2, r_3) = \frac{r_1 r_2 r_3}{r_1 r_2 + r_1 r_3 + r_2 r_3}$	$U(0.0001, 21)$	5

corroborate this hypothesis.

7.5.2 Benchmarks

Since, to the best of our knowledge, there is no established set of benchmarks for SRFC problems, we adapted several well known regression benchmarks from [134] (page 8, table 3) and manually defined some formal constraints for them. Additionally, we included three benchmarks (`gravity`, `res2`, `res3`) from our previous work [31] based on the well known laws of physics – Newton's law of universal gravitation, and the equivalent resistance of two and three resistors connected in parallel, respectively. The general characteristics of all 11 SRFC benchmarks used in this study are presented in Table 7.3, and they can be downloaded from: https://github.com/iwob/CDGP/tree/master/resources/benchmarks_phd/cdsr.

The benchmarks are divided into two groups:

- **noNoise**, where benchmarks are generated from the ground truth formulas without any distortions.
- **withNoise**, where both inputs x_j and outputs y_j, the same as generated for the corresponding *noNoise* benchmark, are distorted by a multiplicative Gaussian noise with $\mu = 1$ and $\sigma = 0.01$ (i.e., $x'_j = x_j \cdot \mathcal{N}(1, 0.01)$).

Each benchmark consists in:

- Preconditions specifying which function's arguments are valid. For example, for *gravity* they specify that valid are $m_1, m_2, r > 0$.
- 500 samples generated from a benchmark-dependent uniform distribution specified in Table 7.3. All samples are also required to be correct with respect to the pre-

Table 7.4: The list of preconditions and constraints for all SRFC benchmarks used in this study. The constraints are expected to be satisfied when the function's input satisfies a precondition.

Benchmark: gravity
Solution: $f(m_1, m_2, r) = \frac{6.674 \cdot 10^{-11} m_1 m_2}{r^2}$
Precondition: $m_1, m_2, r > 0$
Constraints:
$f(m_1, m_2, r) = f(m_2, m_1, r)$
$f(m_1, m_2, r) \geq 0$
f monotonically increases w.r.t. m_1
f monotonically increases w.r.t. m_2

Benchmark: keijzer5
Solution: $f(x, y, z) = \frac{30xz}{(x-10) \cdot y^2}$
Precondition: $y \neq 0, x \neq 10$
Constraints:
$x = z = 0 \implies f(x, y, z) = 0$
$x = y = z \ \wedge \ x > 10 \implies f(x, y, z) > 0$
$x = y = z \ \wedge \ x < 10 \implies f(x, y, z) < 0$

Benchmark: keijzer12
Solution: $f(x, y) = x^4 - x^3 + \frac{y^2}{2} - y$
Precondition: none
Constraints:
$x \geq 0 \implies f(x, y) \leq f(-x, y)$
$y \geq 0 \implies f(x, y) \leq f(x, -y)$
$x = y = 0 \implies f(x, y) = 0$
$x \leq 0 \implies f$ monotonically decreases w.r.t. x
$y \geq 1 \implies f$ monotonically increases w.r.t. y
$y \leq 1 \implies f$ monotonically decreases w.r.t. y

Benchmark: keijzer14
Solution: $f(x, y) = \frac{8}{2 + x^2 + y^2}$
Precondition: none
Constraints:
$f(x, y) \geq 0$
$f(x, y) \leq 4$
$f(x, y) \leq f(0, 0)$
$f(x, y) = f(y, x)$

Benchmark: keijzer15
Solution: $f(x, y) = \frac{x^3}{5} + \frac{y^3}{2} - y - x$
Precondition: none
Constraints:
$x = y = 0 \implies f(x, y) = 0$
$x = -y \ \wedge \ x \leq 0 \implies f(x, y) \geq 0$
$x = -y \ \wedge \ x \geq 0 \implies f(x, y) \leq 0$

Benchmark: nguyen1
Solution: $f(x) = x^3 + x^2 + x$
Precondition: none
Constraints:
$x > 0 \implies f(x) \geq 0$
$x < 0 \implies f(x) \leq 0$
$x > 0 \implies f(x) \geq f(-x)$

Benchmark: nguyen3
Solution: $f(x) = x^5 + x^4 + x^3 + x^2 + x$
Precondition: none
Constraints:
$x > 0 \implies f(x) \geq 0$
$x < 0 \implies f(x) \leq 0$
$x > 0 \implies f(x) \geq f(-x)$

Benchmark: nguyen4
Solution: $f(x) = x^6 + x^5 + x^4 + x^3 + x^2 + x$
Precondition: none
Constraints:
$x > 0 \implies f(x) \geq 0$
$x < 0 \implies f(x) \geq -0.75$
$x > 0 \implies f(x) \geq f(-x)$

Benchmark: pagie1
Solution: $f(x, y) = \frac{1}{1 + x^{-4}} + \frac{1}{1 + y^{-4}}$
Precondition: $x, y \neq 0$
Constraints:
$f(x, y) \geq 0$
$f(x, y) \leq 2$
$f(x, y) = f(y, x)$

Benchmark: res2
Solution: $f(r_1, r_2) = \frac{r_1 r_2}{r_1 + r_2}$
Precondition: $r_1, r_2 > 0$
Constraints:
$f(r_1, r_2) = f(r_2, r_1)$
$f(r_1, r_2) \leq r_1 \ \wedge \ f(r_1, r_2) \leq r_2$
$f(r_1, r_2) > 0$

Benchmark: res3
Solution: $f(r_1, r_2, r_3) = \frac{r_1 r_2 r_3}{r_1 r_2 + r_1 r_3 + r_2 r_3}$
Precondition: $r_1, r_2, r_3 > 0$
Constraints:
$f(r_1, r_2, r_3) = f(r_2, r_1, r_3)$
$f(r_1, r_2, r_3) = f(r_3, r_2, r_1)$
$f(r_1, r_2, r_3) = f(r_1, r_3, r_2)$
$f(r_1, r_2, r_3) \leq r_1 \ \wedge \ f(r_1, r_2, r_3) \leq r_2 \ \wedge \ f(r_1, r_2, r_3) \leq r_3$
$f(r_1, r_2, r_3) > 0$

Listing 7.5: An example of our modified SyGuS format (notice the presence of a 'precondition' term) for the `res2 noNoise` benchmark.

```
(set-logic NRA)
(synth-fun res2 ((r1 Real)(r2 Real)) Real)
(declare-var r1 Real)
(declare-var r2 Real)
(constraint (= (res2 5.62255 16.57643) 4.1984724747037925))
(constraint (= (res2 15.88644 18.40775) 8.527205801040935))
...
(constraint (= (res2 11.33382 0.88174) 0.8180944997036567))

(precondition (and (> r1 0.0) (> r2 0.0)))
(constraint (= (res2 r1 r2) (res2 r2 r1)))
(constraint (and (<= (res2 r1 r2) r1) (<= (res2 r1 r2) r2)))
(constraint (> (res2 r1 r2) 0.0))

(check-synth)
```

conditions. Samples were generated only once before the experiment, and in every algorithm's run they are randomly partitioned into training set (375 samples) and test set (125 samples). The cross-validation framework described earlier works only on the 375 training samples, and after it finishes, the most promising regressor is tested on the remaining 125 test samples.

- A set of formal constraints we devised based on the properties of solutions. Constraints are enforced only for the values specified as valid by preconditions. For example, for *gravity* we selected symmetry with respect to masses (i.e., $g(m_1, m_2, r) = g(m_2, m_1, r)$), non-negative codomain ($g(m_1, m_2, r) \geq 0$), and increasing monotonicity with respect to masses. Table 7.4 presents a detailed listing of preconditions and constraints for all benchmarks.

The benchmarks are represented in two formats, different for CDSR and constraint-agnostic regression algorithms. The former employs the SyGuS format [165], modified with an additional 'precondition' term because our implementation of CDSR requires preconditions to be specified explicitly (in pure SyGuS format an equivalent semantic can be achieved by means of an additional implication in the 'constraint' terms). The latter employs the TSV (Tab Separated Values) format, in which formal constraints and preconditions are omitted. This was purely a pragmatic choice, since TSV is readily accepted by the implementation of the regression algorithms that we used.

An example specification of the SRFC task in our modified SyGuS format can be found in Listing 7.5. In this format, similarly as in the original SyGuS format, on the level of syntax there is no distinction between tests and constraints, which is understandable, given that it was created for the task of program synthesis from a full formal specification. Thus, in order to seed the T_c with the training set in CDSR we need to recognize tests among other constraints, and we do it by detecting equality constraints where one of the arguments is an application of the target function to some constants, and the other argument is a constant.

Table 7.6: The average ratio of satisfied constraints for all benchmarks (N=Noise). Success rate greater than 0% was obtained only by algorithms with a ratio of 1.0 of satisfied properties, and their success rate was also 100%. We have applied a shading which marks the worst value as white and the best as dark gray, and the best value achieved on a given benchmark is in bold.

	Ada-Boost	Gradient-Boosting	Kernel-Ridge	Lasso-Lars	Linear	Linear-SVR	MLP	Random-Forest	SGD	XG-Boost
gravity	**0.50**	**0.50**	0.00	**0.50**	**0.50**	**0.50**	0.00	**0.50**	**0.50**	**0.50**
keijzer12	0.17	0.00	**0.33**	0.17	0.17	0.17	0.00	0.00	0.17	0.17
keijzer14	0.75	0.50	**1.00**	**1.00**	0.25	0.50	0.25	0.75	0.25	0.75
keijzer15	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**
keijzer5	0.33	**0.67**	**0.67**	**0.67**	0.33	0.33	0.33	**0.67**	**0.67**	0.33
nguyen1	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
nguyen3	**1.00**	**1.00**	0.00	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
nguyen4	**1.00**	**1.00**	0.67	0.67	0.67	0.67	**1.00**	**1.00**	0.67	**1.00**
pagie1	**0.67**	0.33	**0.67**	0.00	0.00	0.33	0.00	**0.67**	0.33	**0.67**
res2	**0.67**	**0.67**	**0.67**	0.33	0.33	0.33	0.33	**0.67**	0.33	**0.67**
res3	**0.80**	**0.80**	**0.80**	0.20	0.20	0.20	0.00	**0.80**	0.20	**0.80**
gravityN	**0.50**	**0.50**	**0.50**	**0.50**	0.25	**0.50**	0.00	**0.50**	**0.50**	**0.50**
keijzer12N	**0.17**	0.00	**0.17**	**0.17**	**0.17**	**0.17**	0.00	0.00	**0.17**	**0.17**
keijzer14N	0.75	0.50	**1.00**	**1.00**	0.25	0.50	0.25	0.75	0.25	0.75
keijzer15N	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**
keijzer5N	0.33	**0.67**	**0.67**	**0.67**	0.33	0.33	0.33	**0.67**	**0.67**	0.33
nguyen1N	**1.00**	**1.00**	0.67	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
nguyen3N	**1.00**	**1.00**	0.33	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
nguyen4N	**1.00**	**1.00**	0.67	0.67	0.67	0.67	**1.00**	**1.00**	0.67	**1.00**
pagie1N	0.67	0.33	**1.00**	0.33	0.00	0.67	0.33	0.33	0.00	0.33
res2N	**0.67**	**0.67**	**0.67**	0.33	0.33	0.33	0.33	**0.67**	0.33	**0.67**
res3N	**0.80**	**0.80**	**0.80**	0.20	0.20	0.20	0.00	**0.80**	0.20	**0.80**
Mean	0.66	0.62	0.59	0.55	0.42	0.50	0.40	0.66	0.48	0.64
Rank	4.36	4.89	4.82	5.45	6.95	6.07	7.27	4.50	6.16	4.52

7.5.3 Discussion of the results

In the following, we will evaluate regression algorithms on the following criteria:

- The ratio of satisfied constraints to the total number of constraints of a given benchmark, representing the degree to which the model exhibits the expected properties of the target function, which can be observed when one considers many data points (qualitative evaluation of generalization). This information is presented in Table 7.6.
- The MSE on test set, which represents an aggregated numerical error on examples not seen during training (quantitative evaluation of generalization). This information is presented in Table 7.7.

The following observations can be made:

1. As expected, there was a high variability in the difficulty level of benchmarks. For example, the constraints of `nguyen` benchmarks were almost always satisfied. On the other end of the spectrum are benchmarks like `keijzer12` or `keijzer15`, for which only a small fraction of constraints were satisfied per run. For the satisfiability of individual constraints for every benchmark, refer to Tables B.4 and B.5 in Appendix B.
2. Certain algorithms were much better than other at capturing high-level function's behavior: `AdaBoost`, `GradientBoosting`, `RandomForest`, and `XGBoost` obtained the best ranks with similar overall efficiency. Out of those, `AdaBoost` had both the

Table 7.7: The median MSE on test set for all benchmarks (N=Noise). We have applied a row-wise shading which marks the worst values in a row as white and the best as dark gray, and the best value achieved on a given benchmark is in bold.

	Ada-Boost	Gradient-Boosting	Kernel-Ridge	Lasso-Lars	Linear	Linear-SVR	MLP	Random-Forest	SGD	XG-Boost
gravity	**2.6E-14**	5.5E-14	5.5E-14	5.6E-14	5.5E-14	5.6E-14	5.5E-14	5.6E-14	5.6E-14	4.2E-14
keijzer12	4.6E5	4.2E3	**3.9E2**	9.0E6	9.1E6	9.1E6	1.9E3	3.6E3	9.1E6	1.6E3
keijzer14	1.1E-2	2.5E-2	**2.6E-4**	6.4E-2	6.4E-2	6.3E-2	3.9E-3	1.3E-3	6.4E-2	1.2E-3
keijzer15	1.3E3	2.1E2	**1.4E-10**	1.0E4	1.0E4	1.0E4	8.2E0	5.2E2	1.0E4	1.3E2
keijzer5	7.5E4	1.1E7	2.8E9	1.1E7	2.3E7	**4.3E4**	1.2E7	1.4E7	8.3E6	2.7E8
nguyen1	6.5E2	5.2E1	**2.1E-9**	5.0E4	5.0E4	5.1E4	5.9E1	2.6E1	5.0E4	1.5E1
nguyen3	2.8E7	1.3E6	**2.1E3**	6.0E8	6.0E8	6.0E8	7.7E4	1.3E6	6.0E8	7.4E5
nguyen4	5.1E9	1.6E8	3.2E9	7.1E10	7.1E10	7.1E10	**1.3E7**	3.1E7	7.2E10	3.6E7
pagie1	1.3E-2	**1.2E-3**	3.0E-3	1.4E-1	1.4E-1	1.4E-1	1.3E-3	1.1E-2	1.4E-1	2.7E-3
res2	1.0E-1	1.8E-2	**3.1E-5**	1.4E0	1.4E0	1.4E0	7.2E-4	1.6E-2	1.4E0	8.2E-3
res3	1.5E-1	2.4E-2	**3.1E-3**	6.4E-1	6.4E-1	6.4E-1	4.1E-3	1.7E-2	6.4E-1	8.9E-3
gravityN	2.1E-14	1.4E-14	1.2E-14	1.4E-14	1.3E-14	1.4E-14	1.4E-14	1.4E-14	1.4E-14	**9.9E-15**
keijzer12N	4.8E5	3.8E4	2.4E4	8.9E6	8.8E6	8.8E6	**2.3E4**	3.2E4	8.9E6	3.5E4
keijzer14N	1.1E-1	7.1E-2	**5.9E-4**	2.1E-1	2.1E-1	2.1E-1	6.0E-3	2.5E-2	2.1E-1	1.2E-2
keijzer15N	1.4E3	2.8E2	**6.7E1**	1.1E4	1.1E4	1.1E4	8.5E1	7.0E2	1.1E4	2.2E2
keijzer5N	**4.5E6**	1.4E7	4.3E7	1.4E7	2.6E7	**4.5E6**	9.8E6	1.9E7	1.6E7	1.1E8
nguyen1N	9.8E2	3.9E2	2.6E2	4.0E4	4.0E4	4.0E4	**2.5E2**	3.2E2	4.0E4	3.3E2
nguyen3N	1.4E7	1.0E7	**7.5E6**	5.9E8	5.9E8	5.9E8	7.7E6	1.1E7	5.9E8	1.0E7
nguyen4N	4.9E9	8.9E8	4.4E8	1.1E11	1.1E11	1.1E11	**3.7E8**	7.8E8	1.2E11	1.0E9
pagie1N	2.5E-2	7.6E-3	3.0E-3	1.8E-1	1.8E-1	1.9E-1	**1.9E-3**	2.2E-2	1.8E-1	7.7E-3
res2N	1.3E-1	2.0E-2	5.5E-3	1.0E0	9.9E-1	1.0E0	**4.0E-3**	2.1E-2	1.0E0	1.1E-2
res3N	1.7E-1	2.0E-2	**4.8E-3**	7.8E-1	7.8E-1	7.8E-1	1.3E-2	3.4E-2	7.8E-1	1.3E-2
Rank	5.52	4.36	2.52	7.98	7.86	7.77	2.57	4.61	8.25	3.55

highest average ratio of satisfied constraints, and the best average rank.

3. The best MSE on test set was achieved by `KernelRidge`, which had also a decent ratio of satisfied constraints. In contrast, the second algorithm with the best MSE, i.e., `MLP` (multilayer perceptron), achieved the lowest ratio of satisfied constraints.
4. The presence of noise, at least at the assumed magnitude (normal distribution with $\sigma = 1\%$ of the value being distorted), did not have much effect on the number of satisfied constraints. What is interesting, in some cases the noise led to the increased ratio of satisfied constraints, for example `KernelRidge` on `gravity` benchmark. MSE on test set was also often lower after noise was added, though sometimes (`keijzer5` benchmark) it was much higher for certain algorithms.
5. For every algorithm, the ratio of satisfied constraints is almost always better than that achieved by linear regression. This suggests that certain properties of linear functions, since only those can be satisfied by `Linear`, are also preserved in many nonlinear models and are relatively easy to satisfy. An analogous observation can be made for MSE, but this was expected since target functions are nonlinear, and nonlinear regression algorithms should, in theory, handle them better.

The statistical analysis was conducted using Friedman's test for multiple achievements of multiple subjects [91, 154], with the Wilcoxon-Nemenyi-McDonald-Thompson post-hoc test (also known as the Nemenyi post-hoc test) [86, 154]. For the ratio of satisfied constraints, there was one significant difference: `AdaBoost` satisfied more constraints (p-value = 0.047) than `MLP`. For the MSE on test set, there were several statistically significant differences – `LassoLars`, `Linear`, `LinearSVR`, and `SGD` were dominated (p-values < 0.025)

Table 7.8: The parameters of CDSR and CDSR_p that remained fixed during the experiment.

Parameter	*Value*
Number of runs	50
Population size	1000
Maximum number of generations	∞
Maximum runtime in seconds	1800
Solver timeout in seconds	3
Probability of mutation	0.5
Probability of crossover	0.5
Tournament size	7
Maximum height of initial programs	4
Maximum height of trees inserted by mutation	4
Maximum height of programs in population	12
Validation set improvement window (generations)	25

Figure 7.9: The grammar of programs generated by CDSR. v_i is the ith input variable, and $U(-1, 1)$ is an ephemeral constant sampled from $[-1, 1]$.

`R ::= R + R | R - R | R * R | R / R |` v_1 | v_2 | ... | v_n | $U(-1, 1)$

by all other approaches with the exception of `AdaBoost`, which was dominated (p-value $= 0.03$) only by the best under this criterion `KernelRidge`.

7.6 Experiment 2: Comparison of different variants of CDSR

In the second experiment, we examine the efficiency and generalization power of CDSR[6] and its variants in different configurations, and then compare them with the standard regression algorithms from Experiment 1. To facilitate comparison, the set of benchmarks remains the same (see Section 7.5.2).

7.6.1 Configuration of CDSR

Metaheuristic algorithms, such as GP, usually have several important hyperparameters and components that can immensely impact the search process. CDSR inherits all hyperparameters (e.g., population size, probability of mutation) and components (e.g., selection algorithm, mutation/crossover operator) of GP since it is an extension of it, and adds several of its own. The most important hyperparameters of CDSR which remain constant throughout the experiment are presented in Table 7.8.

The instruction set of CDSR contains only standard arithmetic operators (+, -, *, /), and the formal grammar is presented in Figure 7.9. Division is not protected, and division by 0 is always interpreted as leading to a wrong output. The fitness function interprets such cases as infinite error values.

[6] https://github.com/iwob/CDGP, master branch, commit `444d9ec` from 4th July 2021.

Table 7.10: The dimensions of the experiment.

CDSR:

- **Selection:** tournament, ϵ-lexicase
- **Tests ratio α:** 0.75, 1.0

CDSR$_\mathbf{p}$:

- **Selection:** tournament, ϵ-lexicase
- **Tests ratio α:** 0.75, 1.0
- **w_c:** 1, 5

Table 7.11: The average ratio of satisfied constraints, aggregated across all benchmarks.

	CDSR		CDSR$_\mathrm{p}$			
	Tour	*Lex*	*Tour*		*Lex*	
w_c			1	5	1	5
α=0.75	0.26	0.22	0.30	0.29	0.32	**0.41**
α=1.0	0.25	0.21	0.31	0.29	0.32	**0.41**

Table 7.12: The average number of evaluated solutions (in thousands), aggregated across all benchmarks.

	CDSR		CDSR$_\mathrm{p}$			
	Tour	*Lex*	*Tour*		*Lex*	
w_c			1	5	1	5
α=0.75	52.2	37.3	25.4	26.2	9.3	9.4
α=1.0	**62.3**	55.1	27.2	26.7	9.4	9.3

Similarly as for CDGP, we used in this experiment the Z3 [53, 55] SMT solver.

7.6.2 Dimensions of the experiment

Here, we present the "dimensions" of this experiment, i.e., the hyperparameters/components that we control for and want to gain some insight into their impact on the results. Two variants of CDSR are considered: 'vanilla' CDSR (Section 7.4), and CDSR$_\mathrm{p}$ (Section 7.4.1). The dimensions of the experiment are presented in Table 7.10.

The use of ϵ-lexicase [110] was motivated by the fact that in CDSR test outcomes are continuous (real numbers), and thus if similarly as in the standard lexicase selection (Section 4.1.8.2) we would select only the best individuals for a given test, then it would usually limit our set of candidates for reproduction to a single solution after just one iteration of the lexicase loop. To prevent this, ϵ-lexicase, for each test separately, computes a threshold, defined as the median absolute deviation (MAD) [157] (this is the variant of ϵ-lexicase presented in equation 5 in [110]). MAD is computed by a formula:

$$\mathrm{MAD}(e_t) = \mathrm{median}_j\ (|e_{t_j} - \mathrm{median}_k(e_{t_k})|),$$

where e_t is a vector of errors for all tests t, and e_{t_j} is an error of a solution j on the test t. In other words, MAD is a median deviation of the data from the median. For each test t selected in an ϵ-lexicase's iteration, solutions pass it when they commit on it an absolute error lower than $\mathrm{MAD}(e_t)$ of the best error achieved in the population for that test.

7.6.3 Discussion of the results

Table 7.11 presents the average ratios of satisfied constraints for CDSR and CDSR$_\mathrm{p}$, aggregated across all benchmarks. Note, that this is a different information than success rate (a 'success' is defined as a situation in which all constraints are satisfied). We can observe that α did not make a big difference in the number of constraints satisfied by the algorithms. Thus, for clarity, we decided to include in the main body of this work only the results for one of the α values, and present the complete results of CDSR and its

Table 7.13: Success rates for all benchmarks (N=Noise).

	CDSR		CDSR_p			
	Tour	*Lex*	*Tour*		*Lex*	
α	1.0	1.0	1.0		1.0	
w_c			1	5	1	5
gravity	0.02	**0.08**	0.00	0.00	0.02	0.06
keijzer12	0.04	0.10	0.00	0.00	0.04	**0.12**
keijzer14	0.58	0.00	0.64	0.70	0.20	**0.72**
keijzer15	0.00	0.06	0.00	0.02	0.14	**0.28**
keijzer5	0.02	0.04	0.02	0.00	0.02	**0.08**
nguyen1	**0.90**	0.42	**0.90**	0.86	0.62	0.64
nguyen3	**0.40**	0.12	0.36	0.26	0.22	0.38
nguyen4	0.12	0.12	0.24	0.24	0.34	**0.50**
pagie1	0.32	0.10	0.42	0.24	0.40	**0.58**
res2	**0.90**	0.84	0.86	0.80	0.66	0.76
res3	0.10	**0.40**	0.00	0.00	0.28	0.22
gravityN	0.00	0.00	0.00	0.00	0.00	**0.04**
keijzer12N	**0.02**	0.00	0.00	0.00	0.00	**0.02**
keijzer14N	0.50	0.00	0.62	0.64	0.26	**0.74**
keijzer15N	0.02	0.02	0.00	0.02	0.14	**0.22**
keijzer5N	0.00	0.06	0.02	0.02	**0.10**	0.06
nguyen1N	0.36	0.38	0.40	0.50	**0.70**	0.52
nguyen3N	0.22	0.20	0.24	0.28	0.24	**0.36**
nguyen4N	0.28	0.18	0.28	0.24	**0.42**	0.40
pagie1N	0.32	0.08	0.44	0.18	0.44	**0.74**
res2N	0.54	0.66	0.70	0.42	**0.72**	0.58
res3N	0.02	**0.30**	0.00	0.00	0.18	0.24
Mean	0.26	0.19	0.28	0.25	0.28	0.38
Rank	3.84	4.07	3.82	4.25	3.16	1.86

Table 7.14: The average ratio of satisfied constraints for all benchmarks (N=Noise).

	CDSR		CDSR_p			
	Tour	*Lex*	*Tour*		*Lex*	
α	1.0	1.0	1.0		1.0	
w_c			1	5	1	5
gravity	0.40	0.46	0.41	0.39	0.48	**0.53**
keijzer12	0.10	0.12	0.46	0.40	0.43	**0.56**
keijzer14	0.62	0.03	0.67	**0.73**	0.23	**0.73**
keijzer15	0.03	0.07	0.03	0.03	0.17	**0.36**
keijzer5	0.23	0.19	0.23	0.16	0.20	**0.32**
nguyen1	**0.90**	0.42	**0.90**	0.86	0.66	0.69
nguyen3	0.40	0.12	0.37	0.26	0.33	**0.50**
nguyen4	0.12	0.12	0.24	0.25	0.35	**0.56**
pagie1	0.48	0.40	0.67	0.56	0.73	**0.84**
res2	**0.91**	0.88	0.87	0.86	0.77	0.83
res3	0.10	0.45	0.10	0.14	**0.46**	0.42
gravityN	0.38	0.38	0.41	0.42	0.47	**0.49**
keijzer12N	0.15	0.13	0.38	0.36	0.35	**0.49**
keijzer14N	0.56	0.01	0.66	0.70	0.33	**0.77**
keijzer15N	0.03	0.03	0.05	0.05	0.17	**0.29**
keijzer5N	0.21	0.23	0.24	0.19	0.33	**0.35**
nguyen1N	0.45	0.42	0.49	0.59	**0.77**	0.68
nguyen3N	0.39	0.33	0.35	0.45	0.37	**0.52**
nguyen4N	0.31	0.22	0.35	0.30	**0.50**	0.49
pagie1N	0.52	0.37	0.74	0.61	0.74	**0.88**
res2N	0.69	0.77	**0.81**	0.62	**0.81**	0.78
res3N	0.05	0.38	0.10	0.10	0.33	**0.50**
Mean	0.37	0.30	0.43	0.41	0.45	0.57
Rank	4.36	4.84	3.30	3.98	2.91	1.61

variants in Appendix B. We can observe there that while the ratio of satisfied constraints is similar for both α values, the MSE on test set was overall much better for $\alpha = 1.0$ (see Table B.3), and thus we continue our analysis only for that superior value. This was, however, an interesting result in itself, because it suggests that counterexamples collected for regression problems may not be worth the computational effort involved in their evaluation.

We can make the following observations based on Tables 7.13–7.15:

1. Overall, the highest success rate and satisfiability ratio, both on benchmarks with and without noise, was obtained by CDSR_p/Lex/w_c=5. This proves that the additional focus on constraints was effective. This was, however, achieved at the cost of a significantly worse MSE, especially for CDSR_p with lexicase selection.
2. For CDSR_p, lexicase achieves better constraint satisfiability than the corresponding tournament selection variants. At the same time, however, when we consider MSE on the test set, then tournament variants of CDSR_p are better than the lexicase ones. Interestingly, the situation is reversed for 'vanilla' CDSR, where lexicase leads to lower MSE on test set, while the amount of satisfied constraints is on a similar level as for tournament selection.
3. When the satisfiability ratio is considered, tournament selection is less robust in the presence of noise compared to lexicase selection. It is worth noting, that for CDSR_p with $w_c = 1.0$, the amount of satisfied constraints remained roughly the same after the noise was applied, which cannot be said about corresponding tournament

Table 7.15: The median MSE on test set for all benchmarks (N=Noise).

	CDSR		$CDSR_p$			
	Tour	*Lex*	*Tour*		*Lex*	
α	1.0	1.0	1.0		1.0	
w_c			1	5	1	5
gravity	7.3E-15	**6.7E-16**	3.4E-15	1.1E-14	7.2E-15	7.4E-15
keijzer12	**3.7E-2**	3.0E-1	6.5E2	7.0E2	6.0E2	4.7E2
keijzer14	1.3E-1	**7.7E-5**	1.3E-1	1.2E-1	9.6E-2	1.3E-1
keijzer15	1.1E2	**5.1E-1**	1.2E3	1.3E3	2.9E2	1.5E2
keijzer5	4.9E6	1.7E6	1.3E7	**1.4E6**	1.3E8	1.2E8
nguyen1	**2.7E-27**	3.0E-27	3.0E-27	2.8E-27	2.4E-2	2.8E-2
nguyen3	6.0E-23	1.8E1	6.2E-23	**5.4E-23**	6.0E2	1.5E3
nguyen4	**7.4E-21**	3.5E3	4.4E-2	9.3E-21	1.8E5	4.0E5
pagie1	1.1E-1	**4.4E-3**	1.1E-1	1.0E-1	1.0E-1	1.2E-1
res2	2.5E-31	2.6E-31	**2.4E-31**	2.5E-31	3.1E-31	2.6E-31
res3	1.4E-1	**2.4E-4**	6.3E-1	6.9E-1	7.8E-2	2.1E-1
gravityN	3.3E-15	**1.2E-15**	2.1E-14	1.3E-14	2.1E-14	1.1E-14
keijzer12N	**2.8E4**	2.9E4	3.4E4	3.7E4	3.3E4	2.9E4
keijzer14N	1.2E-1	**1.1E-4**	1.1E-1	1.1E-1	1.1E-1	1.2E-1
keijzer15N	8.5E2	**7.0E1**	1.3E3	1.4E3	1.5E2	1.5E2
keijzer5N	3.4E6	**2.7E6**	2.8E6	6.8E6	7.5E7	8.2E6
nguyen1N	2.9E2	**2.4E2**	**2.4E2**	2.5E2	2.7E2	2.8E2
nguyen3N	4.9E6	**4.3E6**	4.9E6	5.0E6	5.2E6	5.5E6
nguyen4N	6.4E8	7.4E8	**6.1E8**	6.3E8	7.0E8	**6.1E8**
pagie1N	1.1E-1	**6.7E-3**	1.1E-1	8.6E-2	8.6E-2	1.4E-1
res2N	**3.9E-3**	4.1E-3	4.1E-3	**3.9E-3**	4.1E-3	4.2E-3
res3N	1.3E-1	**1.5E-3**	6.5E-1	5.9E-1	2.7E-1	1.5E-1
Rank	2.95	2.09	3.70	3.64	4.09	4.52

variants.

4. Table 7.12 provides some insight into the number of solutions evaluated by each algorithm. Lexicase is a more costly selection scheme than tournament selection, so its lower results were expected. It was, however, a little surprising, how small was the number of evaluated solutions for $CDSR_p$/Lex variants. We speculate that this could be the reason for their worse results in terms of MSE on test set.

Statistical analysis with the Friedman test and Nemenyi post-hoc test showed that when all benchmarks are taken into account, then $CDSR_p$/Lex/w_c=5 satisfies significantly more constraints (p-values < 0.035) than all other configurations except $CDSR_p$/Lex/w_c=1, which in turn satisfies more constraints (p-value $= 0.01$) than the worst CDSR configuration, i.e., CDSR/Lex. For MSE on test set, there were only two significant differences: CDSR/Lex was better (p-values < 0.002) than both $CDSR_p$/Lex variants.

7.6.4 Comparison with constraint-agnostic regression algorithms

We will compare here two CDSR configurations, which proved the best in Experiment 2:

- **CDSR/Lex/α=1.0** – the best average MSE on test set out of all CDSR variants. We will call this configuration $CDSR_{MSE}$.
- **$CDSR_p$/Lex/α=1.0/w_c=5** – the best ratio of satisfied constraints out of all CDSR variants. We will call this configuration $CDSR_{SAT}$.

Table 7.16: A comparison of the best constraint-agnostic regression algorithms with the best configurations of CDSR. The values in this table are compiled from Tables B.6–B.9 in Appendix B, and were averaged over all benchmarks, i.e., both *noNoise* and *withNoise*.

	Ada-Boost	Kernel-Ridge	CDSR_{MSE}	CDSR_{SAT}
(avg. rank) median tests set MSE	3.50	2.27	**1.41**	2.82
(avg. rank) satisfiability ratio	**2.05**	2.16	3.61	2.18
(avg. rank) success rate	2.82	3.14	2.34	**1.70**
(avg.) satisfiability ratio	**0.66**	0.59	0.30	0.57
(avg.) success rate	0.27	0.18	0.19	**0.38**
(avg.) runtime (s)	75.59	**27.49**	1058.95	1729.27

with two regression algorithms with the best results in Experiment 1:

- **AdaBoost** – the best ratio of satisfied constraints and success rate..
- **KernelRidge** – the best MSE on test set.

A summary of the results of the selected algorithms for all benchmarks is presented in Table 7.16, and the full results are presented in Appendix B in Tables B.6–B.9. The following observations can be made:

1. Surprisingly, both constraint-agnostic regression algorithms managed to satisfy on average more constraints than our best CDSR approaches. Despite that, there are some constraints which are easy for CDSR to satisfy, while at the same time hard for constraint-agnostic algorithms, for example: equality constraint for `keijzer12` ($x = y = 0 \implies f(x,y) = 0$), the symmetry constraint for `pagie1` ($f(x,y) = f(y,x)$), the output bound for `res2` and `res3` ($f(r_1,r_2) \leq r_1 \ \wedge \ f(r_1,r_2) \leq r_2$). Examples to the contrary can also be found, for example: output bound for `keijzer15` ($x = -y \ \wedge \ x \leq 0 \implies f(x,y) \geq 0$), but even in such cases there are configurations of CDSR which have the probability of $1/3$ for satisfying that constraint.
2. While constraint-agnostic regression algorithms satisfied the highest number of constraints, CDSR boasts the best total success rate of 38%. Its successes are also much more evenly distributed in the set of benchmarks, in contrast to the constraint-agnostic algorithms which either have a success rate of 0% or 100%. This property was also reflected in the better average ranks of CDSR on this criterion.
3. Yet another surprise was that CDSR_{MSE} managed to achieve a much better average rank on MSE on test set, and generally the most of the lowest MSE scores were obtained by either of the CDSR variants. We expected this to be the other way around, i.e., CDSR, and especially CDSR_{p}, having higher numbers of satisfied properties at the cost of higher MSE. One might be tempted to speculate that the improvement of MSE was caused by the presence of constraints in fitness, but CDSR_{MSE} was a 'vanilla' CDSR variant not taking them into account during search other then by generating counterexamples.
4. Standard machine learning regression algorithms achieve their results much faster

than CDSR, which employs formal verification to check if the constraints are satisfied or not. However, optimizing CDSR for maximum speed was not our priority, and some relatively easy improvements (conceptually, at least) could potentially significantly reduce the execution time (e.g., approximate verification of constraints in CDSR_p instead of full verification with the SMT solver).

A statistical analysis conducted on Tables B.7 and B.8, again with the Friedman statistical test and Nemenyi post-hoc test, revealed that all algorithms satisfied significantly more constraints than CDSR_MSE (p-values < 0.002), and CDSR_MSE achieved significantly lower MSE than `AdaBoost` and CDSR_SAT (p-values < 0.009). Also, `KernelRidge` had significantly lower MSE than `AdaBoost` (p-value $= 0.009$).

7.7 Conclusions

In this chapter, we adapted CDGP to symbolic regression problems, and dubbed the resulting approach Counterexample-Driven Symbolic Regression (CDSR). The motivation for applying CDSR to symbolic regression problems was its focus on formal constraints, which are rarely taken into account in symbolic regression and thus pose an interesting avenue of research. We consider the satisfiability ratio of formal constraints to be an interesting measure of generalization. Contrary to the quantitative evaluation of generalization represented by an aggregation of point-wise errors, constraints describe behavior that a function exhibits over multiple data points, and thus can be thought of as a higher-order generalization, or, as we called it in this thesis, qualitative generalization.

We conducted several computational experiments. First, we assessed a degree to which common machine learning regression algorithms satisfy formal constraints. Their results were impressive, given that they have not taken it explicitly into account during training. As a second step, we tested various configurations of CDSR on the same set of benchmarks. Overall, CDSR satisfied fewer constraints than constraint-agnostic algorithms, but it traded it for much better success rate. Additionally, CDSR managed to satisfy, in at least some of its runs, many constraints that other regression algorithms had problems with, while achieving better MSE on test set. Standard regression algorithms are, however, much more time-efficient in producing their results. CDSR has also a limitation of its own, that is it requires a solver equipped with a theory for a given domain, and transcendental functions, such as logarithms and trigonometric functions, are not supported by contemporary SMT solvers and must be approximated.

Chapter 8

Neuro-Guided Genetic Programming

In this chapter, we present our preliminary work into prioritizing the search in program space. To this aim, we automatically acquire knowledge about problem's structure by training a neural network on a training set of selected program synthesis tasks. Predictions of the network are then used by the modified mutation and initialization operators of genetic programming to promote the instructions indicated as promising by the network.

The work presented in this chapter is based on [122], which was created in collaboration with Paweł Liskowski and Krzysztof Krawiec.

8.1 Introduction

A program synthesis problem is a set of program synthesis tasks (Definition 2.3.2), similarly as the SAT problem is a set of all possible SAT instances. In many practical applications, however, it is often the case that the majority or all of instances that one may ever need to solve constitute only a certain limited subset of the original problem – for instance, the SAT instances which we need to solve may be only Horn clauses (i.e., disjunctive clauses with at most one non-negated literal), and thus a more efficient dedicated solver can be used (in fact, HORNSAT subproblem of SAT is solvable in linear time [150, p. 94]). This is an example of *domain knowledge* about the structure of a problem, and our ultimate objective is to design algorithms that make the most of it to efficiently solve a given task.

The notion of domain knowledge is, unfortunately, hard to formalize, so instead we will approach it by providing some examples. In the field of metaheuristics, it is common for algorithms' behavior to be explicitly dependent on a set of (hyper)parameters (e.g., probability of mutation in evolutionary algorithms). When performance of the algorithm on a given problem is systematically better for certain values of parameters, then we can say that these values are a form of domain knowledge. Another possibility is introducing to the search algorithm a bias towards promising regions of search space, either inherent to the algorithm (e.g., regularization), or dynamically determined based on the particular problem instance. From these examples we can attempt to formulate an informal definition of domain knowledge, i.e., any information that makes the algorithm more efficient on the problems of interest.

In this chapter, we investigate the possibility of automatic acquisition of such knowledge by means of machine learning, and we attempt to bias the evolutionary search so that

it concentrates on regions of search space that are more likely to contain good solutions. Balog et al. [17] presented a framework for acquiring and using domain knowledge in the context of program synthesis, which they called *Learning Inductive Program Synthesis* (LIPS). This framework consists of the following components [17]:

- a domain specific language (DSL) specification,
- a data generation procedure (generation of a training set),
- a machine learning model that maps input-output examples to program attributes,
- a search procedure that searches program space in an order guided by the model.

They also presented an implementation of the LIPS framework in the form of DeepCoder, in which DSL was a language for manipulating lists of integers, a neural network was used as a machine learning model, and there was a selection of different search procedures ranging from depth-first search to SAT-based Sketch program synthesis system [174]. When they compared these search procedures augmented with DeepCoder to their base versions, they noted a significant improvement of efficiency.

Inspired by their work, we decided to apply these principles in the realm of evolutionary program synthesis, and more precisely genetic programming (GP). Since the DeepCoder's source code was not shared by the authors, we reimplemented their system based on their paper [17], while introducing certain minor improvements. The main novelty of this chapter is thus the examination of how different variants of GP behave when augmented with such domain knowledge.

This chapter is organized as follows. In Section 8.2 we describe our version of the DeepCoder system for domain knowledge acquisition but using GP as a search procedure. After that, in Section 8.3 we evaluate different variants of Neuro-Guided GP and selected baselines to assess the impact of the acquired knowledge on the effectiveness of search. Section 8.4 concludes the chapter.

8.2 Neuro-Guided Genetic Programming

In this section, we present our genetic programming synthesis system supported by a trained neural network for prioritizing search, which we called *Neuro-Guided Genetic Programming*. This approach has two distinct phases:

- *Offline learning phase*, in which a training set of program synthesis tasks and their solutions are used to train a neural network to predict, based on input-output examples, which instructions will be present in the solutions (synthesized correct programs).
- *Online solving phase*, which is an application of Neuro-Guided GP to solve a particular program synthesis task, in which the neural network determines the probabilities of using individual DSL instructions in the search process.

The idea of the approach is to train the neural network once (big initial commitment), and then repeatedly use it to solve synthesis tasks more effectively. We will describe these two phases in more detail in the following sections, but first we present the domain specific language (DSL) in which synthesized programs are expressed, since it has significant im-

Listing 8.1: Example program from the DSL (P2 from [17]).

```
a ← [int]
b ← [int]
c ← ZipWith (-) b a
d ← Count (>0) c
```

Listing 8.2: Example program from the DSL (P4 from [17]).

```
x ← [int]
y ← [int]
c ← Sort x
d ← Sort y
e ← Reverse d
f ← ZipWith (*) d e
g ← Sum f
```

pact on various aspects of the synthesis system (e.g., network's inputs and layers, program representation, and variation operators in GP).

8.2.1 Domain-specific language

We decided to synthesize programs in the same programming language as in [17], which is a domain-specific language for operations on lists of integers. Programs in that language are very similar to programs in linear GP [34] because they are fixed-length sequences of instructions, and the results produced by previous instructions are accessible for the subsequent instructions. A single instruction is of the form:

$$var \coloneqq function(arg_1, arg_2, \ldots, arg_k)$$

where var is a fresh variable, $function$ is a function from the DSL, and the arguments arg_i are variables defined earlier, lambdas, or predicates. The set of functions which can be used in instructions comprises the typical list manipulation functions Head, Last, Take, Drop, Access, Minimum, Maximum, Reverse, Sort, Sum, and the higher-order functions (i.e., functions taking as an argument some other function) Map, Filter, Count, ZipWith, Scanl (for the exact semantics of the instructions we refer to [17, p. 17]). The DSL contains also lambdas and predicates which can be used only as arguments for the higher-order functions:

- lambdas of type `int` → `int` for Map: add1, sub1, multMinus1, mult2, mult3, mult4, div2, div3, div4, square.
- predicates of type `int` → `bool` for Filter and Count: >0, <0, isOdd, isEven.
- lambdas of type `(int, int)` → `int` for ZipWith and Scanl: +, -, *, min, max.

The DSL involves two datatypes which can be assigned to variables: integers (`int`) and lists of integers (`[int]`), and a boolean (`bool`) datatype which is used as a return type of the predicates.

Example programs in the DSL are presented in Listings 8.1 and 8.2. The first k

instructions, where k is the number of inputs of a program to be synthesized, represent the inputs of the program and make them accessible for the subsequent instructions. These k instructions are determined by the signature of the program, and as such they are fixed and do not count towards length of a program. The output of a program is the value of the last assigned variable.

8.2.2 Offline learning phase

8.2.2.1 Encoding of the input-output examples

Input-output examples of a given benchmark (program synthesis task) have the form (X_i, Y_i), where i is the index of a given input-output example (there are five of them per benchmark), X_i denotes inputs to the program, and Y_i the correct output. We consider only programs with at most two inputs, and thus $X_i = [X_{i1}\ X_{i2}]$. Programs return only a single output value, so $Y_i = [Y_{i1}]$. A single input-output example is thus represented as a vector $T_i = [X_{i1}\ X_{i2}\ Y_{i1}]$. Each element of the vector T_i is encoded for the neural network in the following way:

- A value's type is encoded via one-hot encoding, and thus a vector of length 2 is sufficient (one element for `int`, and the other for `[int]`).
- Lists of integers have the limit L put on their maximum length, and in our experiments $L = 20$. A list of length n is encoded as a vector of L values, in which the first n elements are the same as the elements of the list, and the remaining elements of the vector have the `NULL` value (technically realized as an integer 256). We also require that all integers are in range $[-256, 255]$ in order to facilitate the learning process and make the problem easier.
- Integers are encoded as lists of length one in the same process as described above, and thus only information about the type differentiates them from actual lists of length 1.

If a program expects only a single input (i.e., $X_{i2} = \emptyset$), then X_{i2} is represented in T_i as L `NULL`s with the zero vector as its type information. It thus follows that each element of vector T_i is encoded using $2 + 20 = 22$ values, the whole vector T_i requires $3 \cdot 22 = 66$ values.

8.2.2.2 Architecture of the neural network

In this study we recreated the neural network used in DeepCoder [17], with only minor changes in the architecture. The diagram of the network's architecture is presented in Figure 8.1. The network is feedforward and consists of several layers:

- the input layer,
- the embedding layer,
- the three fully connected layers (256 neurons each),
- the average pooling layer (256 neurons),
- the fully connected output layer (34 neurons).

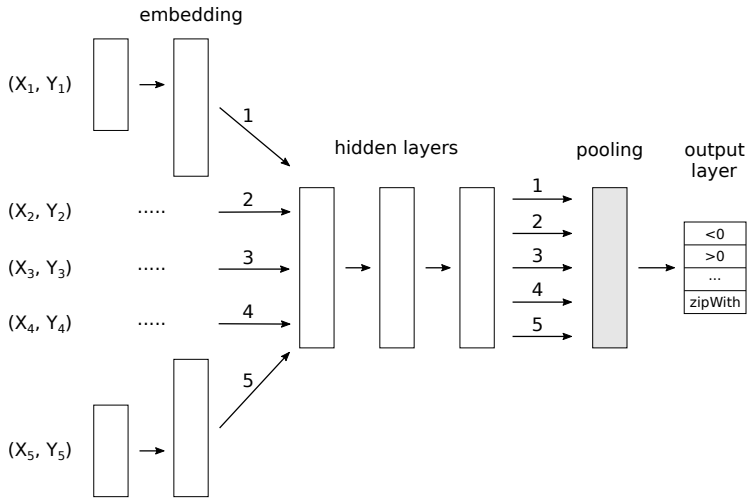


Figure 8.1: The diagram of the neural network's architecture. The hidden layers process each input-output example independently (as indicated by the numbered arrows), and then aggregate results for those five examples in the pooling layer.

Importantly, the first three stages from the above list are applied to each input-output example independently.

As the first processing step, all values representing input-output examples (including `NULL`s), with the exception of vectors representing types, are *embedded* in a 20-dimensional space, which means that each integer is mapped to a certain vector in that space. This mapping is learned in the embedding layer during the network's training, and is expected to distribute numbers "meaningfully" in this space so that the subsequent layers of the network are able to pick on patterns between different integers. As an example, Balog et al. [17, p. 16] reported that in their experiments with 2-dimensional embeddings the even and odd numbers formed parallel elongated clusters (notice the presence of isOdd and isEven predicates in the DSL).

As the next step, the three fully connected layers are independently mapping (with the same weights) each encoded and embedded input-output example of a program into a latent representation, and these five resulting representations are then averaged in the pooling layer.

Finally, the fully connected output layer with sigmoid activation function returns a vector of length 34 (the number of language elements in the DSL) with values in range $[0, 1]$ indicating network's estimation of likelihood that a given language element (function, lambda, predicate) is present in the solution.

8.2.2.3 Generation of the training set

When constructing a training set for machine learning problems, we would like for it to cover all possible observations[1]. Unfortunately, usually this is not the case, and a training set is only a small sample of the space of all possible observations. The objective of Neuro-Guided GP is, however, to efficiently synthesize programs from the a priori selected

[1] At least if we had infinite computing power.

programming language, and nothing prevents us from exhaustive generation of such a training set.

A training set $\mathcal{T}$ is thus constructed by enumerating all programs in the DSL starting from the shortest one-instruction programs and gradually increasing their length up to the assumed maximum length l_{max}. We apply various heuristics to prune redundant semantically equivalent programs and programs with dead code; their absence introduces a bias of our models towards concise programs. For the purpose of computational experiments (Section 8.3), we generated two such training sets for the DSL presented in Section 8.2.1, which we dubbed *small* ($l_{max} = 3$; $|\mathcal{T}| = 822{,}582$) and *large* ($l_{max} = 4$; $|\mathcal{T}| = 5{,}004{,}532$).

For each program $p \in \mathcal{T}$, we generate five random input-output examples. This process involves certain challenges, for example many inputs do not satisfy implicit program's preconditions, such as for example a requirement that both provided lists are of the same length. Additionally, we must ensure that all program's inputs and an output are in the proper range of $[-256, 255]$. To meet these requirements, we "back-propagate" the allowed ranges of variables starting from the program's output, and we end up with constraints on the program's inputs, which allow us to uniformly draw values satisfying the constraints. Our implementation, however, does not take into account constraints for lengths of lists, and thus their acceptable lengths are obtained by trial and error, and if that fails too many times, then the program is discarded from the training set.

8.2.2.4 Network training

During training, the neural network learns to predict from input-output examples (encoded in the way presented in Section 8.2.2.2) generated for a certain program $p \in \mathcal{T}$ which instructions occur in p. A single epoch of the network's training is a pass through all programs in $\mathcal{T}$. A gradient is computed in batches of 512 training examples, with the binary cross-entropy between a vector returned by the network and a target binary vector (with 1s indicating that a given instruction is present in the correct program, and 0s that it is not) being a measure of loss. As the optimization algorithm, we used Adam [100], and network's weights were initialized using the He method [81]. We used the implementations of these method available in the TensorFlow library [9]. We train the network for 100 epochs, with the early stopping condition terminating the optimization process when the loss on a validation set, delineated from the training set, starts deteriorating.

We examined different possible activation functions (rectified linear units (ReLUs), 'leaky' ReLUs, and exponential-linear units (ELUs)) for the three consecutive fully connected layers (Section 8.2.2.2), and decided to use ReLUs for the *small* training set, and ELU for the *large* training set. The accuracy on the test set (10000 programs not present in the training set) of these best performing activation functions was 92.48% for the *small* (ReLU), and 90.85% for the *large* training set (ELU).

While the full results of the experiment are presented in Section 8.3, we find it beneficial to present in Figure 8.2 the heatmaps visualizing predictions of the trained neural networks for, respectively, *small* and *large* training sets, and for the eight considered benchmarks P0–P7 (see Section 8.3.2). Squares with green frames indicate instructions which are part of the solution and ideally should have the likelihood of 1. For comparison, we include also the priors representing frequencies of occurrence of lexical elements of the DSL in all

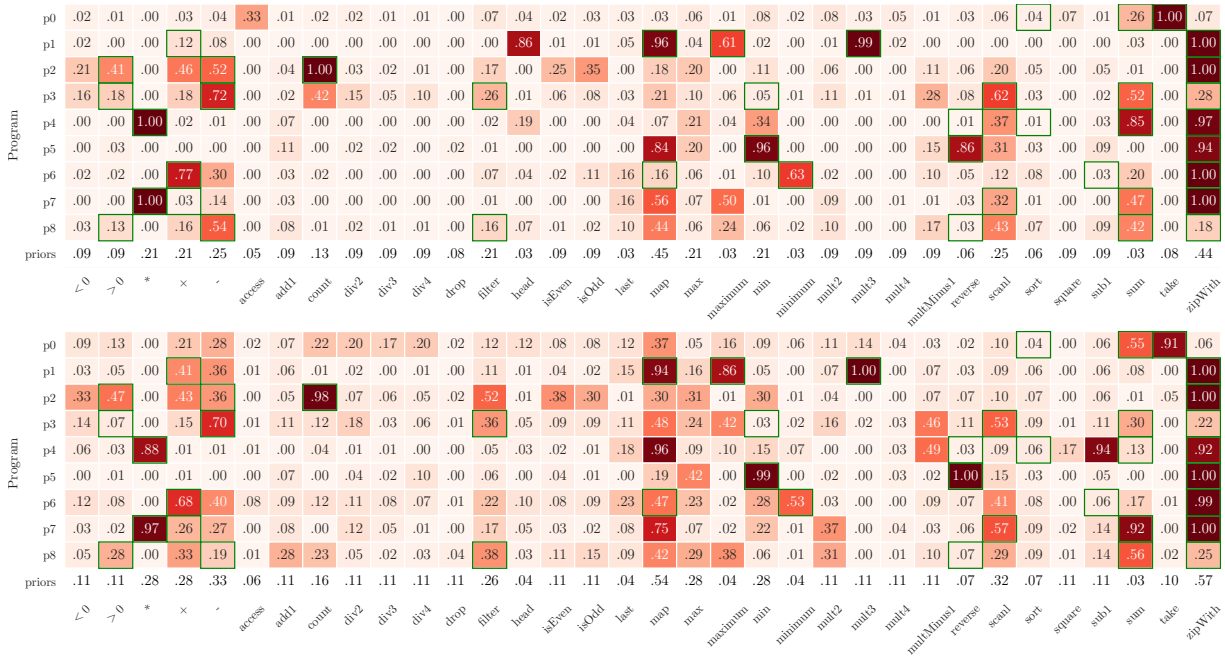


Figure 8.2: The visualization of the trained networks' outputs for individual benchmarks on, respectively, *small* and *large* training sets (reproduced from [122]). In the columns are lexical elements of the DSL, and in the rows are benchmarks used in our experiments. Included are also priors computed on the whole training sets.

$p \in \mathcal{T}$.

Based on Figure 8.2, we can observe that:

- Predictions of the networks significantly diverge from the priors and vary per benchmark.
- The networks manage to discover meaningful patterns between input-output examples and lexical elements of the DSL. For example, only programs P4 (Listing 8.2) and P7 were supposed to use the multiplication lambda (*), and both networks correctly indicated this fact, at the same time predicting probability 0 for that lambda for all other benchmarks. Statistically, this observation is backed by the high accuracy that we reported earlier in this section.

8.2.3 Online solving phase

In this section, we detail Neuro-Guided GP, the genetic programming algorithm adapted to using predictions of the trained neural network to search through the space of programs more effectively. We will emphasize that the neural network is trained only once, and then it can be repeatedly used for different synthesis tasks. We denote the vector of probabilities provided by the network as $P(\mathcal{L}|\mathcal{T})$, where $\mathcal{L}$ is the set of all relevant lexical elements of the DSL (Section 8.2.1).

A task specification for Neuro-Guided GP is a set of input-output examples. The first step of our method is querying the trained neural network on these examples in order to get a concrete vector of probabilities $P(\mathcal{L}|\mathcal{T})$. This vector is then used to bias GP's initialization and variation operators. Notice that the querying of the network takes place

only once per a run of Neuro-Guided GP.

8.2.3.1 Fitness

While the neural network accepts only five input-output examples, this number is insufficient for the fitness function to effectively guide the evolutionary search process (there would be only six unique values of fitness). Thus, in our experiments we generated 128 input-output examples in total per a single benchmark, of which only the first five were used by the neural network.

8.2.3.2 Termination condition

A GP run is terminated after a set number of generations (200 in our experiments), or earlier if it finds a program with zero error on the set of all 128 input-output examples. However, program correctness is judged based on the fitness alone, and there is no guarantee that the program is correct outside of the provided sample of input-output examples.

8.2.3.3 Population initialization

During population initialization, programs are generated instruction by instruction. A set V of variables initially contains only the input variables annotated with their types. The process of generation of a new instruction $var \coloneqq f(args)$ proceeds in the following way:

- A function f is drawn from the DSL according to the probability distribution defined by a normalized vector of the network's outputs $P(\mathcal{L}|\mathcal{T})$. If this is the last instruction of the program, we take into account only those functions that have the same output type as the program's return type.
- For each argument arg of f, we draw it randomly from the set of accessible language elements compatible with arg's type. Two situations are possible:
  - A type of arg is `int` or `[int]`, in which case we draw uniformly from the set V of variables created earlier in the program; if no variable of an appropriate type exists, we draw a function f once again[2].
  - A type of arg is a lambda or predicate, in which case we draw it randomly using the normalized vector $P(\mathcal{L}|\mathcal{T})$ and taking into account only those elements that are compatible with the expected lambda/predicate type.
- A new variable symbol var is introduced and added to V, annotated with a return type of f.

This process continues until the assumed program length is reached.

8.2.3.4 Mutation

A mutation operator picks randomly a single instruction in a program, and changes its function f in such a way that the arguments are still valid. A new function f' is selected

[2]The DSL has the property that for any set of variables, there always exists at least one function for which all arguments can be selected from these variables.

Table 8.3: Common parameter settings for all methods.

Parameter	Value
Probability of mutation p_m	0.8
Probability of crossover p_c	0.0 or 0.5
Population size	1000
Selection method	Tournament (**T**) or Lexicase (**L**)
Max program length	3 or 4
Number of fitness cases	128
Max generations	200

using a probability distribution defined by a normalized vector $P(\mathcal{L}|\mathcal{T})$, of which we consider only the functions with the same signature as f. The arguments of the function call remain the same.

We considered also alternative designs of mutation operators, but they proved inferior in our preliminary experiments, so we do not present them here.

8.2.3.5 Crossover

A crossover operator exchanges up to l_c compatible function calls between parent programs (in our experiments $l_c = 2$), but without changing their arguments. In order to find the possible crossover points, we extract from both parents the multisets M_1 and M_2 of type signatures (composed of the types of function's output and arguments) for successive l_c-sized blocks of instructions (*signature vectors*), and calculate the "intersection" multisets M_1' and M_2' by leaving in them only those elements, which are present at least once in both M_1 and M_2. Next, we randomly pick a signature vector from M_1'. It is guaranteed by construction that the same signature vector is present at least once in M_2', and we select it randomly from among these. The function calls corresponding to the selected signature vectors are then exchanged between both parent programs, and the so constructed programs are returned as offspring. If there are no matching signature vectors for both parents, we repeat the procedure with $l_c - 1$, or return both parent programs unchanged as offspring when l_c has reached zero.

As it can be seen, the crossover operator is not informed in any way by the network's predictions $P(\mathcal{L}|\mathcal{T})$.

8.3 Experiment

The goal of the experiment is to investigate the extent to which domain knowledge obtained by the neural network impacts performance of GP, and which GP configurations are well suited for using this knowledge.

8.3.1 Configurations

We compare with each other the following variants of GP, presented below in the order of increasing use of domain knowledge:

Table 8.4: The list of benchmarks used in our study and originating from [17].

Name	*Arity*	*Length of solution*	*Types*	*Description*
P0	2	3	$int, [int] \mapsto int$	Sums the requested number of the smallest elements from the list
P1	2	3	$[int], [int] \mapsto int$	Computes a score of the best team in a soccer league based on the number of wins and ties for each team
P2	2	2	$[int], [int] \mapsto int$	Counts the number of elements in the list passed as the first argument that are greater than the corresponding elements in the list passed as the second argument (Listing 8.1)
P3	1	4	$[int] \mapsto int$	Calculates the minimum number of deductions by 1 which would make the list non-increasing
P4	2	5	$[int], [int] \mapsto int$	Returns the smallest possible total area of rectangles that can be constructed by pairing dimensions provided in the input lists (Listing 8.2)
P5	1	2	$[int] \mapsto [int]$	Returns a list containing at each position i the minimum of the values on positions i in input list and the reversed input list
P6	2	4	$[int], [int] \mapsto int$	Sums the two input lists element-wise and returns the smallest element decreased by 2
P7	2	3	$[int], [int] \mapsto int$	Returns the sum of multiplications of a number at index i in the second input list by the sum of elements on positions $[i, n]$ (n is the length of the input lists) from the first input list
P8	1	4	$[int] \mapsto int$	Computes element-wise differences between the input list and reversed input list, removes negative entries, and returns a sum of the remaining elements

1. **U**: An unbiased GP search, i.e., all instructions have the same probability of being used by the initialization and mutation operators (uniform probability distribution).
2. **P**: A GP search with the initialization and mutation operators biased with *a priori* probabilities $P(\mathcal{L})$ of instructions (called *priors* in Figure 8.2), i.e., their frequency of occurrence in the training set $\mathcal{T}$.
3. **S**: The *Search-only* variant of Neuro-Guided GP, in which predictions $P(\mathcal{L}|\mathcal{T})$ of the neural network are used only during search, and initialization is unbiased (i.e., uses uniform distribution).
4. **IS**: The *Init-and-Search* variant of Neuro-Guided GP, in which predictions $P(\mathcal{L}|\mathcal{T})$ of the neural network are used both during search and initialization.

The parameters shared by the methods are presented in Table 8.3. For certain parameters we examined several different values, which effectively became a separate dimension of the experiment:

- Probability of crossover operator: 0.0 and 0.5.
- Selection method: tournament selection (**T**) and lexicase selection (**L**; Section 4.1.8.2).

The length of programs in the population is set to $1 + l_{max}$, where l_{max} is the maximum length of programs in a given training set. Shorter programs can be effectively constructed by means of redundant code, which in this case is equivalent to not using certain variables in the rest of the program after they are introduced.

8.3.2 Benchmarks

We have used the same benchmarks as in [17], and we present them together with their characteristics and a concise description in Table 8.4. The examples of a correct solution are provided for benchmarks P2 (Listing 8.1) and P4 (Listing 8.2); for the solutions of others, we refer the reader to [17, p. 12]. The arity of benchmarks vary between 1 and 2, and input arguments are of two types: *int* (integer), and [*int*] (list of integers).

8.3.3 Discussion of the results

Tables 8.5 and 8.6 present success rates (i.e., the ratio of runs in which a correct program was found) of the algorithms, respectively for the networks trained on the *small* and *large* training set. Each configuration was ran 50 times. The configurations not assisted with any model (i.e., T_U, L_U) are unaffected by the change of a training set, and thus they were ran only once on our suite of benchmarks and their results are shared between the tables for comparison. Configurations of Neuro-Guided GP are marked in the tables with light blue color, and the configurations of unbiased search with light green.

The benchmarks vary in the level of difficulty they pose to the methods, ranging from very easy ones, which are solved in all runs by all methods (P2), to difficult ones, on which even the best performing configurations barely exceed 50% probability of success (P4). Unsurprisingly, success rate seems to negatively correlate with the length of the target program (2 for P2 (Listing 8.1) and 5 for P4 (Listing 8.2), the longest target program in the benchmark suite).

Most importantly, the success rates of the configurations parameterized with networks' estimates $P(\mathcal{L}|\mathcal{T})$, i.e., S and IS, are systematically better than those of configurations that rely on the uniform distribution (U) and those parameterized by the prior probabilities calculated from the training set (P). Interestingly, the latter are usually worse than the former, which suggests that for an approach that is not informed by the network, it is better to use the uniform distribution. The possible explanation is that many benchmarks are composed of several "frequent" and at least one "niche" instructions, and it thus becomes unlikely for that rare instruction to be drawn, even when it is required for a given task. In contrast, relying on the uniform distribution makes even the "niche" instructions relatively likely to be drawn.

Of the two approaches informed by the networks, priming both initialization and search (IS) performs better. Though this effect was expected, we did not anticipate its size (note the large differences between the average ranks of IS and S configurations). We hypothesize that the observed gain stems from the fact that initialization is relatively likely to produce a program that uses all instructions appointed as most probable by the network. In contrast, the mutation operator can substitute only one instruction at a time, so if it happens to

Table 8.5: Success rates for particular configurations, for the small training set. Legend: T (tournament), L (lexicase), U (unbiased), P (priors baseline), S (search), IS (initialization and search).

Method	T$_U$		T$_P$		T$_S$		T$_{IS}$		L$_U$		L$_P$		L$_S$		L$_{IS}$	
cx	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5
P0	0.70	0.54	0.34	0.42	0.88	0.94	**1.00**	**1.00**	0.58	0.66	0.40	0.58	0.72	0.82	**1.00**	**1.00**
P1	0.18	0.16	0.26	0.24	0.24	0.20	0.54	0.58	0.16	0.08	0.20	0.12	0.60	0.44	**0.96**	**0.96**
P2	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
P3	0.14	0.16	0.12	0.12	0.46	0.48	**1.00**	0.96	0.52	0.62	0.28	0.54	0.82	0.76	**1.00**	**1.00**
P4	0.14	0.06	0.02	0.08	0.02	0.02	0.00	0.00	0.52	**0.56**	0.38	0.44	0.38	0.18	0.14	0.14
P5	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.98	0.92	**1.00**	0.98	**1.00**	0.98	**1.00**	**1.00**
P6	0.08	0.08	0.06	0.14	0.02	0.14	0.04	0.04	0.40	0.60	**0.82**	0.68	0.68	0.74	0.78	0.80
P7	0.16	0.08	0.34	0.16	0.34	0.44	0.56	0.58	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
P8	0.18	0.36	0.10	0.12	0.14	0.18	0.28	0.32	0.36	0.46	0.26	0.30	0.50	0.34	**0.82**	0.76
Mean	0.40	0.38	0.36	0.36	0.46	0.49	0.60	0.61	0.61	0.66	0.59	0.63	0.74	0.70	0.86	0.85
Rank	10.72	10.94	12.00	11.33	10.67	9.61	8.28	8.06	8.50	8.22	8.17	8.78	5.39	7.00	4.17	4.17

Table 8.6: Success rates for particular configurations, for the large training set.

Method	T$_U$		T$_P$		T$_S$		T$_{IS}$		L$_U$		L$_P$		L$_S$		L$_{IS}$	
cx	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5
P0	0.70	0.54	0.34	0.38	0.82	0.78	**1.00**	**1.00**	0.58	0.66	0.54	0.58	0.64	0.68	**1.00**	**1.00**
P1	0.18	0.16	0.20	0.20	0.18	0.24	0.58	0.62	0.16	0.08	0.16	0.16	0.48	0.32	**0.98**	0.88
P2	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
P3	0.14	0.16	0.10	0.10	0.12	0.28	0.68	0.74	0.52	0.62	0.46	0.52	0.60	0.74	**0.98**	0.94
P4	0.14	0.06	0.02	0.00	0.02	0.02	0.00	0.00	0.52	**0.56**	0.52	0.50	0.22	0.32	0.32	0.08
P5	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.98	0.92	0.98	0.96	**1.00**	**1.00**	**1.00**	**1.00**
P6	0.08	0.08	0.00	0.04	0.08	0.10	0.12	0.12	0.40	0.60	0.64	0.70	0.64	0.70	0.72	**0.74**
P7	0.16	0.08	0.28	0.24	0.48	0.42	0.78	0.90	**1.00**	**1.00**	0.98	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
P8	0.18	0.36	0.16	0.08	0.18	0.24	0.42	0.42	0.36	0.46	0.32	0.34	0.28	0.32	**0.84**	0.78
Mean	0.40	0.38	0.34	0.34	0.43	0.45	0.62	0.64	0.61	0.66	0.62	0.64	0.65	0.68	0.87	0.82
Rank	10.56	11.06	12.33	12.67	10.44	9.56	7.28	6.89	8.50	7.83	9.39	8.50	7.17	6.11	3.56	4.17

start with a program that contains no useful instructions, it needs to be applied several times to produce the desired effect, which is unlikely.

The lexicase selection operator (L setups) proves its usefulness again, systematically and significantly boosting the success rates in comparison to the tournament selection (T setups). Nevertheless, in relative terms, the informed configurations improve over the non-informed ones irrespectively of the type of selection operator, which suggests that priming is beneficial independently of this component of search algorithm. It becomes thus even less likely for the observed effects of priming to be incidental.

What comes as a bit of surprise is the not so clearly positive effect of using the *large* training set, when compared to the *small* one. For IS, moving from the latter to the former causes the success rate to improve only in 12 cases (combinations of benchmark and settings) while deteriorating on 8 cases, out of the total of $9 \cdot 4 = 36$ (of which 17 could not be improved to begin with due to perfect score). Similarly, there is no clear winner when comparing the *small* and the *large* dataset for S configurations.

For statistical evaluation of our results, we employed the Friedman's test for multiple achievements of multiple subjects [91, 154]. In Table 8.7 we present the average ranks and p-values computed for four disjoint groups of configurations:

- **small**$_N$ – small training set, methods not using crossover.
- **small**$_C$ – small training set, methods using crossover.
- **large**$_N$ – large training set, methods not using crossover.

Table 8.7: Average ranks for the tested configurations. Legend: small/large (training set used), N (no crossover), C (crossover).

small_N	$(p = 0.00877)$	Method	L_{IS}	L_S	T_{IS}	L_P	L_U	T_S	T_U	T_P
		Rank	2.50	3.06	4.28	4.28	4.56	5.50	5.67	6.17
small_C	$(p = 0.010579)$	Method	L_{IS}	L_S	T_{IS}	L_U	L_P	T_S	T_U	T_P
		Rank	2.17	3.61	4.33	4.33	4.72	5.22	5.72	5.89
large_N	$(p = 0.00093)$	Method	L_{IS}	L_S	T_{IS}	L_U	L_P	T_S	T_U	T_P
		Rank	2.06	3.61	3.72	4.44	4.83	5.44	5.50	6.39
large_C	$(p = 0.00075)$	Method	L_{IS}	L_S	T_{IS}	L_U	L_P	T_S	T_U	T_P
		Rank	2.22	3.50	3.83	4.33	4.56	5.11	5.83	6.61

- **large**$_C$ – large training set, methods using crossover.

In each group, the configurations that rely on neural guidance clearly rank at the top. The p-values indicate that some methods are significantly better than others. We conducted a post-hoc analysis with the Wilcoxon-Nemenyi-McDonald-Thompson post-hoc test (also known as the Nemenyi post-hoc test) [86, 154], which allowed us to conclude that L_{IS} is statistically significantly better than T_U and T_P in every group. Additionally, in the large_N group, L_{IS} was significantly better than T_S.

8.4 Conclusions

We have shown that statistical models learned by machine learning algorithms can be successfully used by GP to guide search. This was not an entirely unexpected result, given that several other heuristics were improved in this fashion [17, 69]. While our choice of the DSL and network's architecture was inspired by that of [17], we had no access to the author's original implementation and had to recreate it based on their paper, which confirms the soundness of the general idea.

We hypothesize that aiding search with machine learning models can be particularly beneficial for *stochastic* search. Neural estimates of probabilities are inherently noisy (not least because they are based on just a handful of examples), and thus treating them with absolute confidence bears certain risks – and this is what, at least in a certain sense, some of the deterministic search algorithms considered in [17] do. In particular, the depth-first search uses the instructions strictly in the ordering given by the estimated probabilities, so an instruction unfairly deemed as unlikely will wait very long to be used. A stochastic search, like the evolutionary algorithm considered here, is free from that shortcoming, as it treats the estimates as probabilistic guidance only.

It is worth noting that the proposed approach is largely independent from the underlying programming language (DSL), and the described training regimen could be easily used for any other language operating on integers. For real numbers, embedding becomes a problem; for text strings and generally more complicated data structures, the algorithm would require further modifications.

Chapter 9

Conclusions

9.1 Summary

Program synthesis is a computationally hard problem of growing relevance as the importance and complexity of software increases. In this thesis, we focused on the different ways in which GP, a heuristic program synthesis algorithm, can be augmented to better handle program synthesis problems. We presented several new variants of GP supported by either an SMT solver, or a neural network. The feature these approaches have in common is utilizing external sources of knowledge which are then used to assist search. Using an SMT solver for verification enabled GP to synthesize programs from formal specifications, and the counterexamples were collected from the failed verification attempts and subsequently used to guide search. Using an SMT solver for optimization, in conjunction with holes in the programs evolved by GP, allowed finding the most fitting content of holes, and assigning the corresponding fitness to the whole solution. And last but not least, a neural network was used to learn how the relationship between inputs and the expected output of tests constituting program synthesis tasks can be statistically associated with the instructions in the correct program.

9.2 Contributions

Below we summarize the most important contributions of this thesis:

- Evolutionary Program Sketching (EPS; Chapter 5), our proof-of-concept evolutionary approach to program synthesis by sketching [174, 176], which relegates the creation of sketches to GP, and the user needs only to provide a set of tests. An optimizing SMT solver [26] is then used to complete the sketch in the framework typical for memetic algorithms [140]. In our experiments, the "Baldwinian" variant of EPS proved to be better than both the "Lamarckian" variant and the GP baselines.
- Counterexample-Driven GP (CDGP; Chapter 6), a novel approach for GP to synthesize provably-correct programs from formal specifications. The correctness of programs is verified by the SMT solver, and counterexamples from the failed verification attempts are converted to tests and subsequently used to guide search. To

limit the costly calls to the SMT solver, only the solutions that pass the ratio α of the collected tests are formally verified.

- Definition of the Symbolic Regression with Formal Constraints problem (SRFC; Section 7.2).
- Allowing GP to generate symbolic regression models satisfying arbitrary logical constraints provided by the user (i.e., the SRFC problem). This was realized as an extension of CDGP, and to differentiate these two approaches we called it Counterexample-Driven Symbolic Regression (CDSR; Chapter 7).
- Comparison of CDSR with the state-of-the-art constraint-agnostic machine learning regression algorithms, both in terms of error on test set and the number of satisfied constraints. We discovered that while the best constraint-agnostic regression algorithms surprisingly managed to satisfy more constraints on average than CDSR, the best success rates and MSE on test set were achieved by the best configurations of CDSR using the lexicase selection (Section 4.1.8.2).
- Training a neural network to predict instructions present in the correct program based on the input-output examples, and using this information to prioritize the search (Neuro-Guided GP; Chapter 8). While the network's architecture and original inspiration came from DeepCoder [17], we were the first who tested this approach with GP and investigated the impact of various hyperparameters of GP on the effectiveness of search. Our conclusion is that Neuro-Guided GP outperforms both the 'vanilla' GP and the GP supported with the naive bias computed as the instructions' frequency of occurrence in the training set.

9.3 Future work

The work described in this dissertation can be followed with more research, the main directions of which are:

- Conducting a more comprehensive computational experiment for EPS (Chapter 5) in order to investigate how that approach would scale as the number of benchmark's input variables increases.
- The notion of approximate verification (Section 7.5.1) was devised after we have already implemented and tested CDGP and CDSR, and we used it only to compare the latter with the constraint-agnostic regression algorithms. It could be, however, integrated with the CDGP/CDSR's evaluation function (Algorithm 6.2) and used to drastically reduce the number of costly calls to the SMT solver. We speculate that such a modification would bring substantial reduction of runtime.
- Applying Neuro-Guided GP to problems in which program synthesis task is specified by means of formal specification instead of tests. This would require a vastly different approach on the side of the neural network, since instead of a sequence of numbers (program's inputs and correct output) there would be a logical expression in a certain formalism provided as input to the network.
- Combining all the approaches described in this thesis in a single algorithm. In

principle, there are no obstacles for using CDSR equipped with EPS for finding values of constants and using a trained neural network to bias the choice of instructions selected by the variation operators.

- Our general observation is that GP may not necessarily be the best heuristic for program synthesis. The fitness landscape in GP for non-trivial program synthesis problems is known to be rugged, meaning that introducing even a small difference to a program can drastically influence its fitness. Exploring different heuristic or exact algorithms, such as enumerative search, in conjunction with the techniques proposed in this thesis, can also be an interesting avenue of research.

Appendix A

Queries to SMT solver

A.1 Introduction

In this appendix, we will briefly describe the queries to SMT solver which are used in EPS (Chapter 5), CDGP (Chapter 6), and CDSR (Chapter 7). All queries, with the exception of the optimization query in EPS (Section A.7), will be presented for the `max2` synthesis task, which is defined by the following specification:

$$\begin{aligned} max(x, y) \geq x \quad \wedge \\ max(x, y) \geq y \quad \wedge \\ (max(x, y) = x \ \vee \ max(x, y) = y) \end{aligned}$$

where $x, y \in \mathbb{Z}$.

A.2 Short introduction to SMT-LIB

The established format of communication with SMT solvers is the SMT-LIB language [19, 21]. We will now briefly describe the semantics of this language so that interested readers can understand listings in this appendix without referencing external sources.

SMT-LIB was created with the goal of representing logical formulas in a uniform format, which was inspired by the LISP programming language. In SMT-LIB scripts the most important are `assert` statements, which specify logical formulas. All formulas, even when in different `assert` statements, are connected implicitly by a conjunction. Expressions inside `assert` statements often utilize *free variables*, which are declared with the `declare-fun` statement (in the case of free variables, the arity of the function is 0). A solver will try to find values for those variables in such a way that all formulas in `assert` statements are satisfied. Each free variable has assigned type; in the examples below they are of the type `Int` (i.e., integer numbers). The `define-fun` statement is used to define a function or a constant, which then can be used multiple times in `assert` statements or other functions. Functions may also make use of free variables, so strictly speaking they are more like macros than mathematical functions (unless we implicitly treat free variables as function's arguments).

The remaining commands are on the meta-level and instruct the solver itself:

- `set-logic` – Specifies a *logic* for SMT solver, which is a combination of a theory (e.g., a theory of integers which defines the semantics of arithmetical operators and integer constants) with some additional constraints. For example, as a logic can be specified `LIA` (Linear Integer Arithmetic) or `NIA` (Nonlinear Integer Arithmetic), which subsumes `LIA`. Similarly, there is `LRA` (Linear Real Arithmetic) and `NRA` (Nonlinear Real Arithmetic). There are also dedicated logics for operations on arrays, text strings, or bitvectors.
- `check-sat` – Starts the search for proof of the conjunction of formulas in `assert` statements. The result is either a `sat` answer accompanied with a logical model, i.e., valuation of free variables which make all formulas satisfied, or an `unsat` answer with the interpretation that no such valuation exists.
- `get-value` – Instructs the solver to print the value of a specified variable in the model.

A.3 Verification query (CDGP/CDSR)

```
1  (set-logic LIA)
2  (define-fun max ((x Int)(y Int)) Int (ite (>= y x) x y))
3  (declare-fun x () Int)
4  (declare-fun y () Int)
5  (assert (not (and
6      (>= (max x y) x)
7      (>= (max x y) y)
8      (or (= (max x y) x) (= (max x y) y)))))
9  (check-sat)
10 (get-value (x y))
```

The aim of this query is to determine, if a program satisfies the formal specification. In the example above, the (incorrect) program `(ite (>= y x) x y))` was selected for verification and constitutes the body of the `max` function in line 2. In lines 3-4, inputs to the function are declared as free variables. Then, in lines 5-8, there is the negated conjunction of all constraints from the formal specification. The negation is crucial here, because we are searching for inputs invalidating the program, so that they may be later used to create test cases. In general, for a formal specification given as $(Pre, Post)$, asserted will be $\neg(Pre \Rightarrow Post)$. In the `max2` problem there is no precondition, hence the lack of implication.

The interpretation of solver's results is as follows:

- `sat` – input for which specification does *not* hold was found; a program is incorrect.
- `unsat` – it was proven that no such input exists; a program is correct.

In this case, a `sat` will be returned together with a logical model (counterexample). The output of Z3 4.8.8 for this query looks like this:

```
1 sat
2 ((x 0)
3  (y (- 1)))
```

A.4 Query for evaluation of an incomplete test (CDGP/CDSR)

```
1 (set-logic LIA)
2 (define-fun max ((x Int)(y Int)) Int (ite (>= y x) x y))
3 (define-fun x () Int (- 1))
4 (define-fun y () Int 0)
5 (assert (and
6     (>= (max x y) x)
7     (>= (max x y) y)
8     (or (= (max x y) x) (= (max x y) y))))
9 (check-sat)
```

The aim of this query is to determine, if a program satisfies the formal specification for a given input. This query is very similar to the query in Section A.3, but it has two differences:

- Function's inputs in lines 3-4 are not free variables but constants.
- Specification in lines 5-8 is *not* negated. The objective of this query is to assert correctness, not search for a counterexample.

If the solver answers with `unsat` then the function is incorrect for that input; if it answers with `sat` then it is correct for that input. The output of Z3 4.8.8 for the query above looks like this:

```
1 unsat
```

meaning that program `(ite (>= x y) y x)` is not correct for the inputs $(x = -1, y = 0)$.

A.5 Query for finding output of a test case (CDGP/CDSR)

```
1 (set-logic LIA)
2 (declare-fun out () Int)
3 (define-fun max ((x Int)(y Int)) Int out)
4 (define-fun x () Int (- 1))
5 (define-fun y () Int 0)
6 (assert (and
7     (>= (max x y) x)
8     (>= (max x y) y)
9     (or (= (max x y) x) (= (max x y) y))))
10 (check-sat)
11 (get-value (out))
```

This query is used during the creation of a test case from a counterexample. The aim of this query is to find such an output to a particular input (in this case: $x = -1, y = 0$)

that will satisfy the formal specification. For consistency with other queries, free variable `out` is put inside the `max` function definition. Then, in lines 6-9 we simply assert the original specification.

The output of Z3 4.8.8 for this query looks like this:

```
1 sat
2 ((out 0))
```

The interpretation is that 0 is a correct output with respect to the formal specification. We have, however, no guarantees that it is the only correct output. If solver answers with `unsat` then it means that the specification is contradictory for this input.

A.6 Query for checking if a synthesis problem has global single-output property (CDGP)

```
1  (set-logic LIA)
2  (declare-fun out1 () Int)
3  (declare-fun out2 () Int)
4  (define-fun max2__1 ((x Int)(y Int)) Int out1)
5  (define-fun max2__2 ((x Int)(y Int)) Int out2)
6  (declare-fun x () Int)
7  (declare-fun y () Int)
8
9  (assert (>= (max2__1 x y) x))
10 (assert (>= (max2__1 x y) y))
11 (assert (or (= x (max2__1 x y)) (= y (max2__1 x y))))
12
13 (assert (>= (max2__2 x y) x))
14 (assert (>= (max2__2 x y) y))
15 (assert (or (= x (max2__2 x y)) (= y (max2__2 x y))))
16
17 (assert (distinct out1 out2))
18 (check-sat)
```

The aim of this query is to determine, if there exists such an input to the function that at least two different outputs would satisfy the specification. To this end, two free variables `out1` and `out2` are declared in lines 2 and 3 (and, for consistency with other queries, they are placed inside wrapping-functions defined in lines 4 and 5). Then, in lines 6 and 7, declared are free variables representing inputs of the program. Lines 9-11 and 13-15 contain the original specification with the target function replaced by an appropriate wrapping-function. Both wrapping-functions have the same input. At the very end of the script in line 17, we assert that the two output variables need to have different values.

The interpretation of solver's results is as follows:

- `sat` – two different correct outputs for the same input were found; the synthesis problem does not have global single-output property.

- `unsat` – it was proven that no such outputs exist; the synthesis problem has global single-output property.

For this particular example, the output of Z3 4.8.8 looks like this:

```
unsat
```

A.7 Optimization query (EPS)

```
(set-logic NIA)

; ------------------------------------------------------------------
; FORMAL GRAMMAR
; ------------------------------------------------------------------

(declare-fun H1Start0_r () Int)
(assert (<= 0 H1Start0_r))(assert (<= H1Start0_r 4))
(declare-fun H1Start0_Int () Int)
(declare-fun H1Start1_r () Int)
(assert (<= 0 H1Start1_r))(assert (<= H1Start1_r 1))
(declare-fun H1Start1_Int () Int)
(declare-fun H1Start2_r () Int)
(assert (<= 0 H1Start2_r))(assert (<= H1Start2_r 1))
(declare-fun H1Start2_Int () Int)

(define-fun H1Start2 ((x Int)) Int
    (ite (= H1Start2_r 0)
        x
        H1Start2_Int)
)
(define-fun H1Start1 ((x Int)) Int
    (ite (= H1Start1_r 0)
        x
        H1Start1_Int)
)
(define-fun H1Start0 ((x Int)) Int
    (ite (= H1Start0_r 0)
        x
        (ite (= H1Start0_r 1)
            H1Start0_Int
            (ite (= H1Start0_r 2)
                (+ (H1Start1 x)  (H1Start2 x) )
                (ite (= H1Start0_r 3)
                    (- (H1Start1 x)  (H1Start2 x) )
                    (* (H1Start1 x)  (H1Start2 x) )))))
)


```

```
41 ; -------------------------------------------------------------------
42 ; TEST CASES
43 ; -------------------------------------------------------------------
44
45 (define-fun res_T0 () Int (+ (H1Start0 -2) 1))
46 (define-fun itest0 () Int (ite (= res_T0 5) 1 0))
47 ; -------------------------------------------------------------------
48
49 (define-fun res_T1 () Int (+ (H1Start0 0) 1))
50 (define-fun itest1 () Int (ite (= res_T1 1) 1 0))
51 ; -------------------------------------------------------------------
52
53 (define-fun res_T2 () Int (+ (H1Start0 2) 1))
54 (define-fun itest2 () Int (ite (= res_T2 5) 1 0))
55 ; -------------------------------------------------------------------
56
57 (declare-fun fitness () Int)
58 (assert (= fitness (+ itest0 itest1 itest2)))
59 (maximize fitness)
60 (check-sat)
61 (get-model)
```

In this query we assume that we have a partial program (sketch) with holes, but in the example above for simplicity there is only one hole represented by a function `H1Start0`. This function is parameterized with several free variables (`H1Start0_r`, `H1Start1_r`, `H1Start2_r`, `H1Start0_Int`, `H1Start1_Int`, `H1Start2_Int`) and organized into a more manageable form with the help of auxiliary functions (`H1Start1`, `H1Start2`). The free variables define the space of all possible completions of `H1Start0`, and they do it by expressing the formal grammar of the content that can be put into the hole. In this case, the depth of expression to fill the hole is limited to be not greater than 2. After the definition of `H1Start0`, all test cases are processed by the program and the fitness is computed. Finally, the `maximize` statement instructs SMT solver about the optimization criterion, i.e., maximization of the number of passed tests.

The query can have the following outcomes:

- `sat` – the optimal hole's completion was found.
- `unsat` – no hole's completion was able to satisfy additional, unrelated to the fitness, constraints (there are no such constraints in the example above).

Appendix B

CDSR: Detailed experimental results

B.1 Introduction

In this appendix, we present the full results of the computational experiment performed in Chapter 7:

- **Table B.1** – Success rates for all benchmarks (rows) and CDSR variants (columns). A 'success' is defined as a satisfaction of all constraints at the end of the run.
- **Table B.2** – Average ratio of satisfied constraints per run for all benchmarks and CDSR variants.
- **Table B.3** – Median MSE on test set for all benchmarks and CDSR variants.
- **Table B.4** – (*noNoise benchmarks*) Information about how often a given constraint (rows) was satisfied by a solution returned by a given algorithm (columns). 1.0 means that it was satisfied in every run, and 0.0 that in no run.
- **Table B.5** – (*withNoise* benchmarks) The same as above, but for a different set of benchmarks.
- **Table B.6** – a comparison of success rates between the most performant constraint-agnostic regression algorithms and CDSR.
- **Table B.7** – a comparison of constraint satisfiability ratio between the most performant constraint-agnostic regression algorithms and CDSR.
- **Table B.8** – a comparison of MSE on test set between the most performant constraint-agnostic regression algorithms and CDSR.
- **Table B.9** – a comparison of runtimes between the most performant constraint-agnostic regression algorithms and CDSR.

Table B.1: Success rates for all benchmarks (N=Noise).

	CDSR				CDSR_p							
	Tour		*Lex*		*Tour*				*Lex*			
	0.75	1.0	0.75	1.0	0.75		1.0		0.75		1.0	
					1	5	1	5	1	5	1	5
gravity	0.00	0.02	0.06	**0.08**	0.00	0.00	0.00	0.00	0.00	0.04	0.02	0.06
keijzer12	0.02	0.04	0.08	0.10	0.02	0.00	0.00	0.00	0.04	0.10	0.04	**0.12**
keijzer14	0.66	0.58	0.04	0.00	**0.76**	0.72	0.64	0.70	0.26	0.66	0.20	0.72
keijzer15	0.00	0.00	0.04	0.06	0.00	0.02	0.00	0.02	0.12	**0.30**	0.14	0.28
keijzer5	0.02	0.02	0.00	0.04	0.00	0.02	0.02	0.00	0.04	0.04	0.02	**0.08**
nguyen1	0.86	0.90	0.46	0.42	**0.92**	0.86	0.90	0.86	0.68	0.80	0.62	0.64
nguyen3	0.28	**0.40**	0.06	0.12	**0.40**	0.20	0.36	0.26	0.26	0.38	0.22	0.38
nguyen4	0.24	0.12	0.18	0.12	0.12	0.34	0.24	0.24	0.30	0.46	0.34	**0.50**
pagie1	0.62	0.32	0.30	0.10	0.40	0.52	0.42	0.24	0.40	**0.66**	0.40	0.58
res2	0.84	**0.90**	0.82	0.84	0.84	0.76	0.86	0.80	0.74	0.78	0.66	0.76
res3	0.04	0.10	**0.46**	0.40	0.00	0.02	0.00	0.00	0.16	0.26	0.28	0.22
gravityN	0.00	0.00	**0.14**	0.00	0.00	0.00	0.00	0.00	0.02	0.04	0.00	0.04
keijzer12N	0.00	0.02	0.02	0.00	0.00	0.02	0.00	0.00	**0.04**	**0.04**	0.00	0.02
keijzer14N	0.60	0.50	0.00	0.00	0.58	0.48	0.62	0.64	0.18	0.70	0.26	**0.74**
keijzer15N	0.04	0.02	0.08	0.02	0.02	0.02	0.00	0.02	0.14	**0.34**	0.14	0.22
keijzer5N	0.00	0.00	0.02	0.06	0.04	0.04	0.02	0.02	0.00	0.08	**0.10**	0.06
nguyen1N	0.44	0.36	0.46	0.38	0.58	0.38	0.40	0.50	0.72	**0.74**	0.70	0.52
nguyen3N	0.26	0.22	0.16	0.20	0.14	0.08	0.24	0.28	**0.36**	**0.36**	0.24	**0.36**
nguyen4N	0.24	0.28	0.34	0.18	0.24	0.20	0.28	0.24	0.34	**0.44**	0.42	0.40
pagie1N	0.46	0.32	0.18	0.08	0.24	0.24	0.44	0.18	0.32	0.66	0.44	**0.74**
res2N	**0.84**	0.54	0.76	0.66	0.58	0.50	0.70	0.42	0.78	0.68	0.72	0.58
res3N	0.02	0.02	**0.32**	0.30	0.00	0.00	0.00	0.00	0.22	0.18	0.18	0.24
Mean	0.29	0.26	0.23	0.19	0.27	0.25	0.28	0.25	0.28	0.40	0.28	0.38
Rank	6.66	7.25	6.80	7.73	7.55	8.07	7.34	8.25	5.77	3.09	6.05	3.45

Table B.2: The average ratio of satisfied constraints for all benchmarks (N=Noise).

	CDSR				CDSR_p							
	Tour		*Lex*		*Tour*				*Lex*			
	0.75	1.0	0.75	1.0	0.75		1.0		0.75		1.0	
					1	5	1	5	1	5	1	5
gravity	0.28	0.40	0.41	0.46	0.35	0.42	0.41	0.39	0.49	0.49	0.48	**0.53**
keijzer12	0.12	0.10	0.22	0.12	0.43	0.42	0.46	0.40	0.39	0.54	0.43	**0.56**
keijzer14	0.69	0.62	0.05	0.03	**0.79**	0.77	0.67	0.73	0.30	0.70	0.23	0.73
keijzer15	0.00	0.03	0.04	0.07	0.03	0.03	0.03	0.03	0.19	0.33	0.17	**0.36**
keijzer5	0.18	0.23	0.10	0.19	0.26	0.23	0.23	0.16	0.25	0.27	0.20	**0.32**
nguyen1	0.86	0.90	0.47	0.42	**0.92**	0.86	0.90	0.86	0.69	0.85	0.66	0.69
nguyen3	0.32	0.40	0.10	0.12	0.43	0.27	0.37	0.26	0.36	**0.50**	0.33	**0.50**
nguyen4	0.25	0.12	0.18	0.12	0.14	0.35	0.24	0.25	0.30	0.49	0.35	**0.56**
pagie1	0.77	0.48	0.40	0.40	0.66	0.75	0.67	0.56	0.74	**0.85**	0.73	0.84
res2	0.89	**0.91**	0.89	0.88	0.89	0.85	0.87	0.86	0.82	0.85	0.77	0.83
res3	0.06	0.10	**0.49**	0.45	0.12	0.11	0.10	0.14	0.38	**0.49**	0.46	0.42
gravityN	0.34	0.38	0.44	0.38	0.42	0.43	0.41	0.42	0.48	**0.50**	0.47	0.49
keijzer12N	0.23	0.15	0.10	0.13	0.36	0.37	0.38	0.36	0.41	0.43	0.35	**0.49**
keijzer14N	0.61	0.56	0.03	0.01	0.67	0.57	0.66	0.70	0.30	0.75	0.33	**0.77**
keijzer15N	0.06	0.03	0.09	0.03	0.04	0.04	0.05	0.05	0.17	**0.37**	0.17	0.29
keijzer5N	0.24	0.21	0.19	0.23	0.27	0.21	0.24	0.19	0.29	0.34	0.33	**0.35**
nguyen1N	0.53	0.45	0.53	0.42	0.59	0.47	0.49	0.59	0.77	**0.83**	0.77	0.68
nguyen3N	0.35	0.39	0.23	0.33	0.35	0.25	0.35	0.45	0.44	0.47	0.37	**0.52**
nguyen4N	0.27	0.31	0.37	0.22	0.35	0.29	0.35	0.30	0.37	**0.54**	0.50	0.49
pagie1N	0.63	0.52	0.33	0.37	0.57	0.59	0.74	0.61	0.64	0.84	0.74	**0.88**
res2N	**0.89**	0.69	0.82	0.77	0.74	0.67	0.81	0.62	0.87	0.80	0.81	0.78
res3N	0.05	0.05	0.40	0.38	0.11	0.12	0.10	0.10	0.37	0.46	0.33	**0.50**
Mean	0.39	0.37	0.31	0.30	0.43	0.41	0.43	0.41	0.46	0.58	0.45	0.57
Rank	7.84	8.57	8.18	9.39	6.36	7.36	6.57	7.59	5.20	2.59	5.64	2.70

Table B.3: The median MSE on test set for all benchmarks (N=Noise).

	CDSR				CDSR_p							
	Tour		*Lex*		*Tour*				*Lex*			
	0.75	1.0	0.75	1.0	0.75		1.0		0.75		1.0	
					1	5	1	5	1	5	1	5
gravity	$1.2 \cdot 10^{-14}$	$7.3 \cdot 10^{-15}$	$3.0 \cdot 10^{-15}$	$\mathbf{6.7 \cdot 10^{-16}}$	$2.7 \cdot 10^{-14}$	$1.1 \cdot 10^{-14}$	$3.4 \cdot 10^{-15}$	$1.1 \cdot 10^{-14}$	$1.1 \cdot 10^{-14}$	$1.1 \cdot 10^{-14}$	$7.2 \cdot 10^{-15}$	$7.4 \cdot 10^{-15}$
keijzer12	$1.3 \cdot 10^{2}$	$\mathbf{3.7 \cdot 10^{-2}}$	$1.0 \cdot 10^{2}$	$3.0 \cdot 10^{-1}$	$7.0 \cdot 10^{2}$	$6.0 \cdot 10^{2}$	$6.5 \cdot 10^{2}$	$7.0 \cdot 10^{2}$	$5.0 \cdot 10^{2}$	$5.0 \cdot 10^{2}$	$6.0 \cdot 10^{2}$	$4.7 \cdot 10^{2}$
keijzer14	$1.2 \cdot 10^{-1}$	$1.3 \cdot 10^{-1}$	$5.6 \cdot 10^{-4}$	$\mathbf{7.7 \cdot 10^{-5}}$	$1.1 \cdot 10^{-1}$	$1.1 \cdot 10^{-1}$	$1.3 \cdot 10^{-1}$	$1.2 \cdot 10^{-1}$	$9.7 \cdot 10^{-2}$	$1.3 \cdot 10^{-1}$	$9.6 \cdot 10^{-2}$	$1.3 \cdot 10^{-1}$
keijzer15	$5.2 \cdot 10^{2}$	$1.1 \cdot 10^{2}$	$2.3 \cdot 10^{0}$	$\mathbf{5.1 \cdot 10^{-1}}$	$1.3 \cdot 10^{3}$	$1.2 \cdot 10^{3}$	$1.2 \cdot 10^{3}$	$1.3 \cdot 10^{3}$	$9.8 \cdot 10^{1}$	$2.3 \cdot 10^{2}$	$2.9 \cdot 10^{2}$	$1.5 \cdot 10^{2}$
keijzer5	$8.9 \cdot 10^{7}$	$4.9 \cdot 10^{6}$	$5.4 \cdot 10^{6}$	$1.7 \cdot 10^{6}$	$1.6 \cdot 10^{6}$	$3.5 \cdot 10^{6}$	$1.3 \cdot 10^{7}$	$\mathbf{1.4 \cdot 10^{6}}$	$1.2 \cdot 10^{7}$	$4.8 \cdot 10^{7}$	$1.3 \cdot 10^{8}$	$1.2 \cdot 10^{8}$
nguyen1	$2.9 \cdot 10^{-27}$	$2.7 \cdot 10^{-27}$	$3.7 \cdot 10^{-27}$	$3.0 \cdot 10^{-27}$	$2.7 \cdot 10^{-27}$	$\mathbf{2.6 \cdot 10^{-27}}$	$3.0 \cdot 10^{-27}$	$2.8 \cdot 10^{-27}$	$2.2 \cdot 10^{-2}$	$7.6 \cdot 10^{-2}$	$2.4 \cdot 10^{-2}$	$2.8 \cdot 10^{-2}$
nguyen3	$3.7 \cdot 10^{-3}$	$6.0 \cdot 10^{-23}$	$7.7 \cdot 10^{1}$	$1.8 \cdot 10^{1}$	$2.0 \cdot 10^{-1}$	$1.5 \cdot 10^{0}$	$6.2 \cdot 10^{-23}$	$\mathbf{5.4 \cdot 10^{-23}}$	$3.4 \cdot 10^{2}$	$1.4 \cdot 10^{3}$	$6.0 \cdot 10^{2}$	$1.5 \cdot 10^{3}$
nguyen4	$8.3 \cdot 10^{0}$	$\mathbf{7.4 \cdot 10^{-21}}$	$2.2 \cdot 10^{4}$	$3.5 \cdot 10^{3}$	$2.1 \cdot 10^{2}$	$3.6 \cdot 10^{1}$	$4.4 \cdot 10^{-2}$	$9.3 \cdot 10^{-21}$	$8.5 \cdot 10^{4}$	$6.1 \cdot 10^{5}$	$1.8 \cdot 10^{5}$	$4.0 \cdot 10^{5}$
pagie1	$1.2 \cdot 10^{-1}$	$1.1 \cdot 10^{-1}$	$1.6 \cdot 10^{-2}$	$\mathbf{4.4 \cdot 10^{-3}}$	$1.1 \cdot 10^{-1}$	$1.3 \cdot 10^{-1}$	$1.1 \cdot 10^{-1}$	$1.0 \cdot 10^{-1}$	$6.7 \cdot 10^{-2}$	$1.3 \cdot 10^{-1}$	$1.0 \cdot 10^{-1}$	$1.2 \cdot 10^{-1}$
res2	$2.7 \cdot 10^{-31}$	$2.5 \cdot 10^{-31}$	$2.7 \cdot 10^{-31}$	$2.6 \cdot 10^{-31}$	$\mathbf{2.4 \cdot 10^{-31}}$	$2.6 \cdot 10^{-31}$	$\mathbf{2.4 \cdot 10^{-31}}$	$2.5 \cdot 10^{-31}$	$2.8 \cdot 10^{-31}$	$2.6 \cdot 10^{-31}$	$3.1 \cdot 10^{-31}$	$2.6 \cdot 10^{-31}$
res3	$1.9 \cdot 10^{-1}$	$1.4 \cdot 10^{-1}$	$1.3 \cdot 10^{-3}$	$\mathbf{2.4 \cdot 10^{-4}}$	$6.3 \cdot 10^{-1}$	$7.4 \cdot 10^{-1}$	$6.3 \cdot 10^{-1}$	$6.9 \cdot 10^{-1}$	$2.6 \cdot 10^{-1}$	$1.7 \cdot 10^{-1}$	$7.8 \cdot 10^{-2}$	$2.1 \cdot 10^{-1}$
gravityN	$1.1 \cdot 10^{-14}$	$3.3 \cdot 10^{-15}$	$\mathbf{2.5 \cdot 10^{-16}}$	$1.2 \cdot 10^{-15}$	$1.1 \cdot 10^{-14}$	$3.0 \cdot 10^{-15}$	$2.1 \cdot 10^{-14}$	$1.3 \cdot 10^{-14}$	$2.1 \cdot 10^{-14}$	$1.1 \cdot 10^{-14}$	$2.1 \cdot 10^{-14}$	$1.1 \cdot 10^{-14}$
keijzer12N	$3.0 \cdot 10^{4}$	$\mathbf{2.8 \cdot 10^{4}}$	$\mathbf{2.8 \cdot 10^{4}}$	$2.9 \cdot 10^{4}$	$3.2 \cdot 10^{4}$	$3.0 \cdot 10^{4}$	$3.4 \cdot 10^{4}$	$3.7 \cdot 10^{4}$	$3.2 \cdot 10^{4}$	$3.3 \cdot 10^{4}$	$3.3 \cdot 10^{4}$	$2.9 \cdot 10^{4}$
keijzer14N	$9.7 \cdot 10^{-2}$	$1.2 \cdot 10^{-1}$	$5.6 \cdot 10^{-4}$	$\mathbf{1.1 \cdot 10^{-4}}$	$1.2 \cdot 10^{-1}$	$1.1 \cdot 10^{-1}$	$1.1 \cdot 10^{-1}$	$1.1 \cdot 10^{-1}$	$9.3 \cdot 10^{-2}$	$1.2 \cdot 10^{-1}$	$1.1 \cdot 10^{-1}$	$1.2 \cdot 10^{-1}$
keijzer15N	$8.5 \cdot 10^{2}$	$8.5 \cdot 10^{2}$	$\mathbf{6.8 \cdot 10^{1}}$	$7.0 \cdot 10^{1}$	$1.3 \cdot 10^{3}$	$1.3 \cdot 10^{3}$	$1.3 \cdot 10^{3}$	$1.4 \cdot 10^{3}$	$1.8 \cdot 10^{2}$	$2.0 \cdot 10^{2}$	$1.5 \cdot 10^{2}$	$1.5 \cdot 10^{2}$
keijzer5N	$6.2 \cdot 10^{6}$	$3.4 \cdot 10^{6}$	$4.6 \cdot 10^{6}$	$\mathbf{2.7 \cdot 10^{6}}$	$1.1 \cdot 10^{8}$	$1.1 \cdot 10^{8}$	$2.8 \cdot 10^{6}$	$6.8 \cdot 10^{6}$	$8.1 \cdot 10^{6}$	$2.9 \cdot 10^{6}$	$7.5 \cdot 10^{7}$	$8.2 \cdot 10^{6}$
nguyen1N	$2.9 \cdot 10^{2}$	$2.9 \cdot 10^{2}$	$2.5 \cdot 10^{2}$	$\mathbf{2.4 \cdot 10^{2}}$	$2.6 \cdot 10^{2}$	$2.6 \cdot 10^{2}$	$\mathbf{2.4 \cdot 10^{2}}$	$2.5 \cdot 10^{2}$	$2.7 \cdot 10^{2}$	$2.7 \cdot 10^{2}$	$2.7 \cdot 10^{2}$	$2.8 \cdot 10^{2}$
nguyen3N	$4.6 \cdot 10^{6}$	$4.9 \cdot 10^{6}$	$4.5 \cdot 10^{6}$	$\mathbf{4.3 \cdot 10^{6}}$	$5.0 \cdot 10^{6}$	$4.9 \cdot 10^{6}$	$4.9 \cdot 10^{6}$	$5.0 \cdot 10^{6}$	$4.4 \cdot 10^{6}$	$4.7 \cdot 10^{6}$	$5.2 \cdot 10^{6}$	$5.5 \cdot 10^{6}$
nguyen4N	$6.7 \cdot 10^{8}$	$6.4 \cdot 10^{8}$	$6.6 \cdot 10^{8}$	$7.4 \cdot 10^{8}$	$\mathbf{5.5 \cdot 10^{8}}$	$7.1 \cdot 10^{8}$	$6.1 \cdot 10^{8}$	$6.3 \cdot 10^{8}$	$6.4 \cdot 10^{8}$	$6.1 \cdot 10^{8}$	$7.0 \cdot 10^{8}$	$6.1 \cdot 10^{8}$
pagie1N	$1.2 \cdot 10^{-1}$	$1.1 \cdot 10^{-1}$	$1.3 \cdot 10^{-2}$	$\mathbf{6.7 \cdot 10^{-3}}$	$1.1 \cdot 10^{-1}$	$9.8 \cdot 10^{-2}$	$1.1 \cdot 10^{-1}$	$8.6 \cdot 10^{-2}$	$7.4 \cdot 10^{-2}$	$1.3 \cdot 10^{-1}$	$8.6 \cdot 10^{-2}$	$1.4 \cdot 10^{-1}$
res2N	$\mathbf{3.8 \cdot 10^{-3}}$	$3.9 \cdot 10^{-3}$	$4.0 \cdot 10^{-3}$	$4.1 \cdot 10^{-3}$	$4.4 \cdot 10^{-3}$	$4.6 \cdot 10^{-3}$	$4.1 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	$\mathbf{3.8 \cdot 10^{-3}}$	$4.4 \cdot 10^{-3}$	$4.1 \cdot 10^{-3}$	$4.2 \cdot 10^{-3}$
res3N	$1.9 \cdot 10^{-1}$	$1.3 \cdot 10^{-1}$	$5.6 \cdot 10^{-3}$	$\mathbf{1.5 \cdot 10^{-3}}$	$7.4 \cdot 10^{-1}$	$7.0 \cdot 10^{-1}$	$6.5 \cdot 10^{-1}$	$5.9 \cdot 10^{-1}$	$2.4 \cdot 10^{-1}$	$2.9 \cdot 10^{-1}$	$2.7 \cdot 10^{-1}$	$1.5 \cdot 10^{-1}$
Rank	6.77	5.11	3.93	3.18	7.57	7.59	6.75	6.70	6.34	8.16	7.82	8.07

Table B.4: Satisfiability ratio of individual constraints for the *noNoise* benchmarks. An empty cell means that the ratio was zero.
Legend: equality, constant output bound, variable output bound, Symmetry w.r.t. arguments, monotonicity.

	Ada-Boost	Gradient-Boosting	Kernel-Ridge	Lasso-Lars	Linear	Linear-SVR	MLP	Random-Forest	SGD	XG-Boost	CDSR				$CDSR_p$							
											Tour		*Lex*		*Tour*				*Lex*			
											0.75	1.0	0.75	1.0	0.75		1.0		0.75		1.0	
															1	5	1	5	1	5	1	5
keijzer12-2											0.08	0.06	0.18	0.10	0.64	0.54	0.60	0.54	0.46	**0.68**	0.60	0.62
keijzer15-0													0.04	0.06		0.02		0.02	0.20	0.32	0.14	**0.34**
keijzer5-0											0.02	0.14	0.02	0.14	0.04	0.08	0.06	0.04	0.12	**0.16**	0.12	0.14
gravity-1	**1.00**	**1.00**		**1.00**				**1.00**	**1.00**	**1.00**	0.52	0.70	0.66	0.64	0.60	0.80	0.74	0.66	0.94	0.86	0.92	0.96
keijzer14-0	**1.00**		**1.00**	**1.00**		**1.00**	**1.00**	**1.00**		**1.00**	0.72	0.64	0.06	0.06	0.80	0.80	0.68	0.74	0.34	0.72	0.24	0.74
keijzer14-1	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**		**1.00**	**1.00**	**1.00**	0.70	0.64	0.04		0.80	0.76	0.68	0.74	0.30	0.74	0.22	0.74
keijzer15-1	**1.00**	**1.00**		**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**			0.02	0.04	0.06	0.04	0.02	0.06	0.04	0.18	0.34	0.16	0.38
keijzer15-2			**1.00**							**1.00**		0.06	0.04	0.08	0.06	0.04	0.02	0.02	0.18	0.34	0.20	0.36
keijzer5-1	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.42	0.46	0.22	0.32	0.60	0.50	0.54	0.40	0.50	0.54	0.38	0.58
keijzer5-2		**1.00**	**1.00**	**1.00**				**1.00**	**1.00**		0.10	0.08	0.06	0.10	0.14	0.10	0.10	0.04	0.12	0.10	0.10	0.24
nguyen1-0	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.86	0.90	0.48	0.42	0.92	0.86	0.90	0.86	0.70	0.88	0.68	0.72
nguyen1-1	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.86	0.90	0.46	0.42	0.92	0.86	0.90	0.86	0.68	0.80	0.62	0.64
nguyen3-0	**1.00**	**1.00**		**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.34	0.40	0.12	0.12	0.42	0.30	0.38	0.26	0.42	0.58	0.38	0.56
nguyen3-1	**1.00**	**1.00**		**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.28	0.40	0.06	0.12	0.42	0.22	0.36	0.26	0.26	0.38	0.24	0.40
nguyen4-0	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.26	0.12	0.18	0.12	0.12	0.34	0.24	0.26	0.30	0.48	0.34	0.58
nguyen4-1	**1.00**	**1.00**	**1.00**				**1.00**	**1.00**		**1.00**	0.24	0.12	0.18	0.12	0.18	0.36	0.24	0.24	0.30	0.52	0.34	0.52
pagie1-0	**1.00**	**1.00**	**1.00**			**1.00**		**1.00**	**1.00**	**1.00**	0.88	0.68	0.48	0.70	0.84	0.92	0.82	0.76	0.94	0.98	0.92	0.98
pagie1-1	**1.00**							**1.00**		**1.00**	0.80	0.44	0.34	0.40	0.74	0.80	0.76	0.68	0.88	0.90	0.88	0.96
res2-2	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.90	0.92	0.96	0.94	0.94	0.92	0.88	0.92	0.94	0.92	0.90	0.92
res3-4	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**		**1.00**	**1.00**	**1.00**	0.08	0.12	0.60	0.62	0.44	0.36	0.40	0.40	0.78	0.84	0.88	0.74
keijzer12-0			**1.00**								0.16	0.12	0.28	0.12	0.62	0.62	0.70	0.62	0.56	0.74	0.62	0.78
keijzer12-1	**1.00**		**1.00**							**1.00**	0.12	0.12	0.28	0.12	0.54	0.54	0.64	0.46	0.56	0.70	0.58	0.74
keijzer14-2	**1.00**	**1.00**	**1.00**	**1.00**				**1.00**		**1.00**	0.70	0.64	0.06	0.06	0.78	0.80	0.66	0.74	0.32	0.70	0.24	0.74
nguyen1-2	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.86	0.90	0.48	0.42	0.92	0.86	0.90	0.86	0.70	0.88	0.68	0.72
nguyen3-2	**1.00**	**1.00**		**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.34	0.40	0.12	0.12	0.44	0.30	0.38	0.26	0.40	0.54	0.36	0.54
nguyen4-2	**1.00**	**1.00**		**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.24	0.12	0.18	0.12	0.12	0.34	0.24	0.26	0.30	0.46	0.36	0.58
res2-1											0.88	**0.90**	0.86	0.86	**0.90**	0.78	0.86	0.84	0.76	0.80	0.72	0.76
res3-3											0.04	0.10	**0.48**	0.42	0.02	0.02			0.28	0.30	0.32	0.24
gravity-0	**1.00**	**1.00**		**1.00**				**1.00**	**1.00**	**1.00**	0.60	0.86	0.48	0.40	0.80	0.90	0.90	0.90	0.98	0.90	0.96	0.94
keijzer14-3			**1.00**	**1.00**							0.66	0.58	0.04		0.76	0.72	0.64	0.70	0.26	0.66	0.22	0.72
pagie1-2			**1.00**								0.62	0.32	0.38	0.10	0.40	0.52	0.44	0.24	0.40	0.66	0.40	0.58
res2-0	**1.00**	**1.00**	**1.00**					**1.00**		**1.00**	0.88	0.90	0.86	0.84	0.84	0.84	0.88	0.82	0.76	0.82	0.70	0.82
res3-0	**1.00**	**1.00**	**1.00**					**1.00**		**1.00**	0.06	0.10	0.46	0.40	0.04	0.06	0.06	0.08	0.30	0.38	0.34	0.38
res3-1	**1.00**	**1.00**	**1.00**					**1.00**		**1.00**	0.06	0.10	0.46	0.40	0.04	0.08	0.04	0.12	0.26	0.46	0.40	0.36
res3-2	**1.00**	**1.00**	**1.00**					**1.00**		**1.00**	0.06	0.10	0.46	0.40	0.04	0.04	0.02	0.12	0.26	0.48	0.36	0.36
gravity-2					**1.00**	**1.00**						0.02	0.24	0.40					0.02	0.12	0.02	0.10
gravity-3					**1.00**	**1.00**						0.02	0.24	0.38					0.02	0.08	0.04	0.10
keijzer12-3											0.16	0.12	0.28	0.12	0.68	0.70	0.72	0.64	0.56	0.74	0.62	**0.78**
keijzer12-4				**1.00**	**1.00**	**1.00**			**1.00**		0.10	0.08	0.16	0.12	0.06	0.06	0.06	0.12	0.08	0.18	0.08	0.22
keijzer12-5											0.08	0.08	0.16	0.12	0.02	0.04	0.04	0.04	0.10	**0.22**	0.06	0.20
Mean	0.62	0.57	0.55	0.50	0.40	0.45	0.33	0.62	0.45	0.62	0.37	0.36	0.30	0.29	0.44	0.45	0.44	0.41	0.44	0.57	0.43	0.57
Rank	9.21	10.06	10.50	11.35	13.16	12.31	14.30	9.21	12.19	9.21	13.53	12.80	13.06	13.38	11.70	12.00	12.55	13.05	11.07	8.41	11.51	8.43

Table B.5: Satisfiability ratio of individual constraints for the *withNoise* benchmarks. An empty cell means that the ratio was zero.
Legend: equality, constant output bound, variable output bound, Symmetry w.r.t. arguments, monotonicity.

	Ada-Boost	Gradient-Boosting	Kernel-Ridge	Lasso-Lars	Linear	Linear-SVR	MLP	Random-Forest	SGD	XG-Boost	CDSR				$CDSR_p$							
											Tour		*Lex*		*Tour*				*Lex*			
											0.75	1.0	0.75	1.0	0.75		1.0		0.75		1.0	
															1	5	1	5	1	5	1	5
keijzer12N-2											0.32	0.20	0.14	0.20	0.54	0.50	0.60	0.54	0.58	0.56	0.54	**0.64**
keijzer15N-0											0.04	0.02	0.08	0.02	0.04	0.02		0.02	0.18	**0.36**	0.16	0.28
keijzer5N-0											0.12	0.08	0.10	0.12	0.12	0.08	0.08	0.06	0.10	0.18	**0.26**	0.24
gravityN-1	**1.00**	**1.00**	**1.00**	**1.00**				**1.00**	**1.00**	**1.00**	0.64	0.70	0.58	0.60	0.74	0.78	0.76	0.76	0.88	0.94	0.90	0.92
keijzer14N-0	**1.00**		**1.00**	**1.00**		**1.00**		**1.00**		**1.00**	0.60	0.62	0.04	0.02	0.70	0.64	0.66	0.74	0.34	0.76	0.36	0.78
keijzer14N-1	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.62	0.62			0.72	0.62	0.68	0.76	0.34	0.76	0.30	0.78
keijzer15N-1	**1.00**		**1.00**	**1.00**	**1.00**	**1.00**		**1.00**	**1.00**		0.10	0.04	0.08	0.06	0.04	0.08	0.08	0.06	0.16	0.36	0.14	0.26
keijzer15N-2		**1.00**					**1.00**			**1.00**	0.04	0.02	0.10	0.02	0.04	0.02	0.06	0.08	0.18	0.38	0.22	0.32
keijzer5N-1	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.52	0.46	0.38	0.42	0.50	0.44	0.52	0.44	0.62	0.62	0.54	0.64
keijzer5N-2		**1.00**	**1.00**	**1.00**				**1.00**	**1.00**		0.08	0.08	0.08	0.14	0.18	0.10	0.12	0.08	0.16	0.22	0.18	0.16
nguyen1N-0	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.56	0.50	0.56	0.44	0.60	0.50	0.52	0.62	0.78	0.86	0.78	0.74
nguyen1N-1	**1.00**	**1.00**		**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.46	0.36	0.46	0.38	0.58	0.40	0.42	0.52	0.74	0.76	0.72	0.56
nguyen3N-0	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.40	0.44	0.26	0.38	0.44	0.36	0.44	0.52	0.50	0.52	0.40	0.60
nguyen3N-1	**1.00**	**1.00**		**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.26	0.26	0.18	0.22	0.16	0.10	0.28	0.32	0.36	0.36	0.28	0.36
nguyen4N-0	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.26	0.30	0.34	0.22	0.34	0.28	0.34	0.32	0.38	0.54	0.50	0.50
nguyen4N-1	**1.00**	**1.00**	**1.00**				**1.00**	**1.00**		**1.00**	0.30	0.34	0.40	0.24	0.38	0.36	0.34	0.30	0.36	0.54	0.46	0.46
pagie1N-0	**1.00**	**1.00**	**1.00**	**1.00**		**1.00**	**1.00**	**1.00**		**1.00**	0.76	0.62	0.44	0.60	0.78	0.80	0.96	0.86	0.90	0.96	0.94	0.96
pagie1N-1	**1.00**		**1.00**			**1.00**					0.66	0.60	0.28	0.44	0.68	0.72	0.82	0.80	0.68	0.90	0.82	0.92
res2N-2	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.92	0.76	0.86	0.86	0.84	0.82	0.90	0.72	0.94	0.92	0.88	0.86
res3N-4	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**		**1.00**	**1.00**	**1.00**	0.12	0.12	0.52	0.56	0.36	0.46	0.40	0.40	0.68	0.90	0.72	0.96
keijzer12N-0			**1.00**								0.34	0.18	0.14	0.18	0.46	0.56	0.48	0.50	0.58	0.58	0.48	0.70
keijzer12N-1	**1.00**			**1.00**					**1.00**	**1.00**	0.24	0.16	0.14	0.16	0.44	0.50	0.48	0.46	0.54	0.50	0.44	0.66
keijzer14N-2	**1.00**	**1.00**	**1.00**	**1.00**				**1.00**		**1.00**	0.62	0.52	0.04	0.02	0.66	0.56	0.68	0.66	0.34	0.76	0.36	0.78
nguyen1N-2	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.58	0.50	0.56	0.44	0.60	0.50	0.54	0.62	0.78	0.88	0.80	0.74
nguyen3N-2	**1.00**	**1.00**		**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.40	0.46	0.26	0.38	0.46	0.30	0.34	0.50	0.46	0.52	0.42	0.60
nguyen4N-2	**1.00**	**1.00**		**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.26	0.28	0.36	0.20	0.32	0.22	0.38	0.28	0.38	0.54	0.54	0.50
res2N-1											**0.84**	0.58	0.78	0.74	0.72	0.66	0.74	0.46	0.82	0.72	0.78	0.72
res3N-3											0.02	0.06	0.38	**0.40**	0.02	0.02			0.32	0.22	0.18	0.28
gravityN-0	**1.00**	**1.00**	**1.00**	**1.00**				**1.00**	**1.00**	**1.00**	0.70	0.80	0.42	0.26	0.94	0.94	0.86	0.92	0.98	0.98	0.94	0.92
keijzer14N-3			**1.00**	**1.00**							0.60	0.50	0.02	0.02	0.58	0.48	0.62	0.64	0.18	0.72	0.28	0.74
pagie1N-2			**1.00**								0.48	0.34	0.26	0.08	0.24	0.24	0.44	0.18	0.34	0.66	0.46	0.76
res2N-0	**1.00**	**1.00**	**1.00**					**1.00**		**1.00**	0.90	0.74	0.82	0.72	0.66	0.54	0.78	0.68	0.84	0.76	0.78	0.76
res3N-0	**1.00**	**1.00**	**1.00**					**1.00**		**1.00**	0.02	0.02	0.38	0.32	0.10	0.06	0.08		0.28	0.42	0.26	0.46
res3N-1	**1.00**	**1.00**	**1.00**					**1.00**		**1.00**	0.02	0.02	0.36	0.30	0.02	0.02	0.02	0.08	0.22	0.38	0.22	0.38
res3N-2	**1.00**	**1.00**	**1.00**					**1.00**		**1.00**	0.06	0.02	0.38	0.34	0.04	0.02		0.02	0.36	0.38	0.28	0.42
gravityN-2					**1.00**	**1.00**							0.38	0.34					0.04	0.04	0.04	0.08
gravityN-3						**1.00**					0.02		0.38	0.32					0.02	0.04	0.02	0.06
keijzer12N-3											0.40	0.30	0.14	0.22	0.64	0.58	0.64	0.60	0.62	0.70	0.58	**0.76**
keijzer12N-4					**1.00**	**1.00**					0.08	0.04	0.02		0.04	0.04	0.02	0.02	0.08	0.10	0.02	0.10
keijzer12N-5												0.04	0.02		0.02	0.04	0.04	0.04	0.08	**0.12**	0.02	0.06
Mean	0.62	0.57	0.60	0.53	0.38	0.47	0.35	0.60	0.42	0.60	0.36	0.32	0.30	0.28	0.39	0.36	0.40	0.39	0.45	0.56	0.45	0.56
Rank	9.24	10.09	9.66	10.91	13.55	11.91	13.89	9.66	12.61	9.66	12.99	14.44	13.11	14.51	12.46	13.81	12.51	13.00	9.57	7.47	10.29	7.64

Table B.6: Success rates for all benchmarks (N=Noise).

	Ada-Boost	Kernel-Ridge	$CDSR_{MSE}$	$CDSR_{sat}$
gravity	0.00	0.00	**0.08**	0.06
keijzer12	0.00	0.00	0.10	**0.12**
keijzer14	0.00	**1.00**	0.00	0.72
keijzer15	0.00	0.00	0.06	**0.28**
keijzer5	0.00	0.00	0.04	**0.08**
nguyen1	**1.00**	**1.00**	0.42	0.64
nguyen3	**1.00**	0.00	0.12	0.38
nguyen4	**1.00**	0.00	0.12	0.50
pagie1	0.00	0.00	0.10	**0.58**
res2	0.00	0.00	**0.84**	0.76
res3	0.00	0.00	**0.40**	0.22
gravityN	0.00	0.00	0.00	**0.04**
keijzer12N	0.00	0.00	0.00	**0.02**
keijzer14N	0.00	**1.00**	0.00	0.74
keijzer15N	0.00	0.00	0.02	**0.22**
keijzer5N	0.00	0.00	**0.06**	**0.06**
nguyen1N	**1.00**	0.00	0.38	0.52
nguyen3N	**1.00**	0.00	0.20	0.36
nguyen4N	**1.00**	0.00	0.18	0.40
pagie1N	0.00	**1.00**	0.08	0.74
res2N	0.00	0.00	**0.66**	0.58
res3N	0.00	0.00	**0.30**	0.24
Mean	0.27	0.18	0.19	0.38
Rank	2.82	3.14	2.34	1.70

Table B.7: The average ratio of satisfied constraints for all benchmarks (N=Noise).

	Ada-Boost	Kernel-Ridge	$CDSR_{MSE}$	$CDSR_{sat}$
gravity	0.50	0.00	0.46	**0.53**
keijzer12	0.17	0.33	0.12	**0.56**
keijzer14	0.75	**1.00**	0.03	0.73
keijzer15	0.33	0.33	0.07	**0.36**
keijzer5	0.33	**0.67**	0.19	0.32
nguyen1	**1.00**	**1.00**	0.42	0.69
nguyen3	**1.00**	0.00	0.12	0.50
nguyen4	**1.00**	0.67	0.12	0.56
pagie1	0.67	0.67	0.40	**0.84**
res2	0.67	0.67	**0.88**	0.83
res3	**0.80**	**0.80**	0.45	0.42
gravityN	**0.50**	**0.50**	0.38	0.49
keijzer12N	0.17	0.17	0.13	**0.49**
keijzer14N	0.75	**1.00**	0.01	0.77
keijzer15N	**0.33**	**0.33**	0.03	0.29
keijzer5N	0.33	**0.67**	0.23	0.35
nguyen1N	**1.00**	0.67	0.42	0.68
nguyen3N	**1.00**	0.33	0.33	0.52
nguyen4N	**1.00**	0.67	0.22	0.49
pagie1N	0.67	**1.00**	0.37	0.88
res2N	0.67	0.67	0.77	**0.78**
res3N	**0.80**	**0.80**	0.38	0.50
Mean	0.66	0.59	0.30	0.57
Rank	2.05	2.16	3.61	2.18

Table B.8: The median MSE on test set for all benchmarks (N=Noise).

	Ada-Boost	Kernel-Ridge	$CDSR_{MSE}$	$CDSR_{sat}$
gravity	$2.6 \cdot 10^{-14}$	$5.5 \cdot 10^{-14}$	$\mathbf{6.7 \cdot 10^{-16}}$	$7.4 \cdot 10^{-15}$
keijzer12	$4.6 \cdot 10^{5}$	$3.9 \cdot 10^{2}$	$\mathbf{3.0 \cdot 10^{-1}}$	$4.7 \cdot 10^{2}$
keijzer14	$1.1 \cdot 10^{-2}$	$2.6 \cdot 10^{-4}$	$\mathbf{7.7 \cdot 10^{-5}}$	$1.3 \cdot 10^{-1}$
keijzer15	$1.3 \cdot 10^{3}$	$\mathbf{1.4 \cdot 10^{-10}}$	$5.1 \cdot 10^{-1}$	$1.5 \cdot 10^{2}$
keijzer5	$\mathbf{7.5 \cdot 10^{4}}$	$2.8 \cdot 10^{9}$	$1.7 \cdot 10^{6}$	$1.2 \cdot 10^{8}$
nguyen1	$6.5 \cdot 10^{2}$	$2.1 \cdot 10^{-9}$	$\mathbf{3.0 \cdot 10^{-27}}$	$2.8 \cdot 10^{-2}$
nguyen3	$2.8 \cdot 10^{7}$	$2.1 \cdot 10^{3}$	$\mathbf{1.8 \cdot 10^{1}}$	$1.5 \cdot 10^{3}$
nguyen4	$5.1 \cdot 10^{9}$	$3.2 \cdot 10^{9}$	$\mathbf{3.5 \cdot 10^{3}}$	$4.0 \cdot 10^{5}$
pagie1	$1.3 \cdot 10^{-2}$	$\mathbf{3.0 \cdot 10^{-3}}$	$4.4 \cdot 10^{-3}$	$1.2 \cdot 10^{-1}$
res2	$1.0 \cdot 10^{-1}$	$3.1 \cdot 10^{-5}$	$\mathbf{2.6 \cdot 10^{-31}}$	$\mathbf{2.6 \cdot 10^{-31}}$
res3	$1.5 \cdot 10^{-1}$	$3.1 \cdot 10^{-3}$	$\mathbf{2.4 \cdot 10^{-4}}$	$2.1 \cdot 10^{-1}$
gravityN	$2.1 \cdot 10^{-14}$	$1.2 \cdot 10^{-14}$	$\mathbf{1.2 \cdot 10^{-15}}$	$1.1 \cdot 10^{-14}$
keijzer12N	$4.8 \cdot 10^{5}$	$\mathbf{2.4 \cdot 10^{4}}$	$2.9 \cdot 10^{4}$	$2.9 \cdot 10^{4}$
keijzer14N	$1.1 \cdot 10^{-1}$	$5.9 \cdot 10^{-4}$	$\mathbf{1.1 \cdot 10^{-4}}$	$1.2 \cdot 10^{-1}$
keijzer15N	$1.4 \cdot 10^{3}$	$\mathbf{6.7 \cdot 10^{1}}$	$7.0 \cdot 10^{1}$	$1.5 \cdot 10^{2}$
keijzer5N	$4.5 \cdot 10^{6}$	$4.3 \cdot 10^{7}$	$\mathbf{2.7 \cdot 10^{6}}$	$8.2 \cdot 10^{6}$
nguyen1N	$9.8 \cdot 10^{2}$	$2.6 \cdot 10^{2}$	$\mathbf{2.4 \cdot 10^{2}}$	$2.8 \cdot 10^{2}$
nguyen3N	$1.4 \cdot 10^{7}$	$7.5 \cdot 10^{6}$	$\mathbf{4.3 \cdot 10^{6}}$	$5.5 \cdot 10^{6}$
nguyen4N	$4.9 \cdot 10^{9}$	$\mathbf{4.4 \cdot 10^{8}}$	$7.4 \cdot 10^{8}$	$6.1 \cdot 10^{8}$
pagie1N	$2.5 \cdot 10^{-2}$	$\mathbf{3.0 \cdot 10^{-3}}$	$6.7 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$
res2N	$1.3 \cdot 10^{-1}$	$5.5 \cdot 10^{-3}$	$\mathbf{4.1 \cdot 10^{-3}}$	$4.2 \cdot 10^{-3}$
res3N	$1.7 \cdot 10^{-1}$	$4.8 \cdot 10^{-3}$	$\mathbf{1.5 \cdot 10^{-3}}$	$1.5 \cdot 10^{-1}$
Rank	2.50	2.50	2.50	2.50

Table B.9: The average wall-clock runtime (in seconds) for all benchmarks (N=Noise).

	Ada-Boost	Kernel-Ridge	$CDSR_{MSE}$	$CDSR_{sat}$
gravity	889	**322**	1335	1800
keijzer12	62	**48**	1719	1800
keijzer14	36	**4.8**	1518	1800
keijzer15	39	**7.0**	1583	1800
keijzer5	24	**13**	658	1800
nguyen1	13	**3.3**	1158	1447
nguyen3	12	**4.5**	1640	1773
nguyen4	12	**3.5**	1659	1794
pagie1	37	**17**	876	1800
res2	32	**4.2**	313	752
res3	167	**22**	953	1678
gravityN	73	**61**	1285	1800
keijzer12N	29	**17**	388	1800
keijzer14N	21	**4.7**	1209	1800
keijzer15N	39	**6.3**	983	1800
keijzer5N	23	**14**	397	1800
nguyen1N	12	**3.5**	545	1800
nguyen3N	12	**3.5**	403	1800
nguyen4N	12	**4.5**	516	1800
pagie1N	38	**16**	953	1800
res2N	31	**3.9**	1731	1800
res3N	50	**21**	1475	1800
Mean	75.59	27.49	1058.95	1729.27
Rank	2.00	1.00	3.00	4.00

Index

Bibliography

[1] CompCert. https://compcert.org/. Accessed: 2022-01-14.

[2] Evolutionary Computing with Push. https://faculty.hampshire.edu/lspector/push.html. Accessed: 2022-01-14.

[3] Flex: The fast lexical analyzer. https://github.com/westes/flex. Accessed: 2022-01-14.

[4] Helena: Web Automation for End Users. https://helena-lang.org/. Accessed: 2022-01-14.

[5] Human-Competitive Awards 2004 — Present. https://www.human-competitive.org/awards. Accessed: 2022-01-14.

[6] Null httpd. https://sourceforge.net/projects/nullhttpd/. Accessed: 2022-01-14.

[7] Online Etymology Dictionary. https://www.etymonline.com/search?q=program. Accessed: 2022-01-14.

[8] Zune — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Zune. Accessed: 2022-01-14.

[9] Martín Abadi and et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Accessed: 2022-01-14.

[10] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit Seshia, Rajdeep Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-Guided Synthesis. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 1–8. IEEE, October 2013.

[11] 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*, pages 319–336, 2017.

[12] Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. Natural language interfaces to databases – an introduction. *Natural Language Engineering*, 1(1):29–81, 1995.

[13] Andrea Arcuri and Xin Yao. Coevolving Programs and Unit Tests from Their Specification. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 397–400, New York, NY, USA, 2007. ACM.

[14] Andrea Arcuri and Xin Yao. Co-evolutionary Automatic Programming for Software Development. *Information Sciences*, 259:412–432, February 2014.

[15] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[16] James Mark Baldwin. A New Factor in Evolution. *The American Naturalist*, 30(354):441–451, 1896.

[17] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to Write Programs. In *Proceedings International Conference on Learning Representations 2017*. OpenReviews.net, April 2017.

[18] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, page 171–177, Berlin, Heidelberg, 2011. Springer.

[19] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). https://smtlib.cs.uiowa.edu/, 2016. Accessed: 2022-01-14.

[20] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In C. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability, volume 185 of Frontiers in Artificial Intelligence and Applications*, chapter 12, pages 825–885. IOS Press, 2009.

[21] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[22] Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer, Cham, 2018.

[23] Rodrigo C. Barros, Márcio P. Basgalupp, André C.P.L.F. de Carvalho, and Alex A. Freitas. A Hyper-Heuristic Evolutionary Algorithm for Automatically Designing Decision-Tree Algorithms. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, page 1237–1244, New York, NY, USA, 2012. Association for Computing Machinery.

[24] Julie Beaulieu, Christian Gagné, and Marc Parizeau. Lens System Design And Re-engineering With Evolutionary Algorithms. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 155–162, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[25] Jakub Bednarek, Karol Piaskowski, and Krzysztof Krawiec. Ain't Nobody Got Time For Coding: Structure-Aware Program Synthesis From Natural Language. *arXiv preprint*, abs/1810.09717, 2018.

[26] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νZ - An Optimizing SMT Solver. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[27] Iwo Błądek, Maciej Komosiński, and Konrad Miazga. Mappism: formalizing classical and artificial life views on mind and consciousness. *Foundations of Computing and Decision Sciences*, 44(1):55–99, 2019.

[28] Daniel Blasco, Jaime Font, Mar Zamorano, and Carlos Cetina. An Evolutionary Approach for Generating Software Models: The case of Kromaia in Game Software Engineering. *Journal of Systems and Software*, 171:110804, 2021.

[29] Iwo Błądek, Maciej Drozdowski, Frédéric Guinand, and Xavier Schepler. On Contiguous and Non-Contiguous Parallel Task Scheduling. *Journal of Scheduling*, 18(5):487–495, oct 2015.

[30] Iwo Błądek and Krzysztof Krawiec. Evolutionary Program Sketching. In Mauro Castelli, James McDermott, and Lukas Sekanina, editors, *EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming*, volume 10196 of *LNCS*, pages 3–18, Amsterdam, 19-21 April 2017. Springer Verlag.

[31] Iwo Błądek and Krzysztof Krawiec. Solving Symbolic Regression Problems with Formal Constraints. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pages 977–984, New York, NY, USA, 2019. ACM.

[32] Iwo Błądek, Krzysztof Krawiec, and Jerry Swan. Counterexample-Driven Genetic Programming: Heuristic Program Synthesis from Formal Specifications. *Evolutionary Computation*, 26(3):441–469, Fall 2018.

[33] Josh C. Bongard. Evolutionary Robotics. *Communications of the ACM*, 56(8):74–83, August 2013.

[34] Markus F. Brameier and Wolfgang Banzhaf. *Linear Genetic Programming*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[35] Cameron Browne and Frederic Maire. Evolutionary Game Design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):1–16, 2010.

[36] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and Lain-Jinn Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[37] Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: a survey of the state of the art. *The Journal of the Operational Research Society*, 64(12):1695–1724, 2013.

[38] Edmund K. Burke, James P. Newall, and Rupert F. Weare. Initialization Strategies and Diversity in Evolutionary Timetabling. *Evolutionary Computation*, 6(1):81–103, March 1998.

[39] Cristiano Calcagno and Dino Distefano. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 459–465, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[40] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, page 963–975, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, page 6, USA, 2011. USENIX Association.

[42] Salman Cheema, Sarah Buchanan, Sumit Gulwani, and Joseph J. LaViola Jr. A Practical Framework for Constructing Structured Drawings. In *IUI'14 19th International Conference on Intelligent User Interfaces*, Haifa, Israel, February 2014.

[43] Alonzo Church. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[44] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[45] Edmund M. Clarke. *The Birth of Model Checking*, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[46] Clare B. Congdon. *An Evolutionary Algorithms Approach to Phylogenetic Tree Construction*, pages 99–116. Springer London, London, 2005.

[47] Carlos Cotta and Pablo Moscato. Inferring Phylogenetic Trees Using Evolutionary Algorithms. In Juan Julián Merelo Guervós, Panagiotis Adamidis, Hans-Georg Beyer, Hans-Paul Schwefel, and José-Luis Fernández-Villacañas, editors, *Parallel Problem Solving from Nature — PPSN VII*, pages 720–729, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[48] Nichael Lynn Cramer. A Representation for the Adaptive Generation of Simple Sequential Programs. In John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, 24-26 July 1985.

[49] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[50] George B. Dantzig. *Linear Programming and Extensions*. RAND Corporation, Santa Monica, CA, 1963.

[51] Charles Darwin. *On the origin of species by means of natural selection, or, The preservation of favoured races in the struggle for life*. John Murray, 1859.

[52] Ashraf Darwish, Aboul Ella Hassanien, and Swagatam Das. A survey of swarm and evolutionary computing approaches for deep learning. *Artificial Intelligence Review*, 53(3):1767–1812, 2020.

[53] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, chapter 24, pages 337–340. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.

[54] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo Theories: Introduction and Applications. *Communications of the ACM*, 54(9):69–77, September 2011.

[55] Leonardo de Moura and Nikolaj Bjørner. Z3 Theorem Prover: github repository. https://github.com/Z3Prover/z3, 2020.

[56] Edsger W. Dijkstra. Notes on Structured Programming. EWD249 – circulated privately – http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF, April 1970.

[57] Stephane Doncieux, Nicolas Bredeche, Jean-Baptiste Mouret, and Agoston E. Eiben. Evolutionary Robotics: What, Why, and Where to. *Frontiers in Robotics and AI*, 2:4, 2015.

[58] Finale Doshi-Velez and Been Kim. Towards A Rigorous Science of Interpretable Machine Learning, 2017.

[59] John H. Drake, Ahmed Kheiri, Ender Özcan, and Edmund K. Burke. Recent advances in selection hyper-heuristics. *European Journal of Operational Research*, 285(2):405–428, 2020.

[60] Alan Edelman. The Mathematics of the Pentium Division Bug. *SIAM Review*, 39:54–67, 1997.

[61] Agoston E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, 2003.

[62] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to Infer Graphics Programs from Hand-Drawn Images. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[63] Ernest Allen Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 955–1072, 1990.

[64] Ernest Allen Emerson. *The Beginning of Model Checking: A Personal Perspective*, pages 27–45. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[65] Kathleen Fisher, John Launchbury, and Raymond Richards. The HACMS program: using formal methods to eliminate exploitable bugs. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20150401, 2017.

[66] Lawrence J. Fogel, Alvin J. Owens, and Michael J. Walsh. *Artificial Intelligence through Simulated Evolution*. Wiley, Chichester, WS, UK, 1966.

[67] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A Genetic Programming Approach to Automated Software Repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, page 947–954, New York, NY, USA, 2009. Association for Computing Machinery.

[68] Edgar Galván and Peter Mooney. Neuroevolution in Deep Neural Networks: Current Trends and Future Challenges. *CoRR*, abs/2006.05415, 2020.

[69] Lei Gao, Mingxiang Chen, Qichang Chen, Ganzhong Luo, Nuoyi Zhu, and Zhixin Liu. Learn to Design the Heuristics for Vehicle Routing Problem. *arXiv preprint*, 2020.

[70] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[71] Uli Grasemann and Risto Miikkulainen. Effective Image Compression Using Evolved Wavelets. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, page 1961–1968, New York, NY, USA, 2005. Association for Computing Machinery.

[72] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. *arXiv preprint*, 2014.

[73] Sumit Gulwani. Dimensions in Program Synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 13–24, Hagenberg, Austria, January 2010.

[74] Sumit Gulwani. Automating String Processing in Spreadsheets using Input-Output Examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA*, January 2011.

[75] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet Data Manipulation Using Examples. *Communications of the ACM*, 55(8):97–105, August 2012.

[76] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Component Based Synthesis Applied to Bitvector Programs. Technical Report MSR-TR-2010-12, February 2010.

[77] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program Synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[78] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833 – 839, 2001.

[79] Mark Harman, Afshin Mansouri, and Yuanyuan Zhang. Search-Based Software Engineering: Trends, Techniques and Applications. *ACM Comput. Surv.*, 45(1), December 2012.

[80] Wilfred K. Hastings. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika*, 57(1):97–109, 1970.

[81] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, 2015.

[82] Pei He, Lishan Kang, Colin G. Johnson, and Shi Ying. Hoare logic-based genetic programming. *SCIENCE CHINA Information Sciences*, 54(3):623–637, March 2011.

[83] Thomas Helmuth, Lee Spector, and James Matheson. Solving Uncompromising Problems with Lexicase Selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643, 2015.

[84] Charles A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[85] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* University of Michigan Press, 1975.

[86] Myles Hollander and Douglas A. Wolfe. *Nonparametric Statistical Methods.* John Wiley & Sons, New York, 2nd edition, 1999.

[87] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-Guided Component-Based Program Synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224, 2010.

[88] Colin Johnson. Genetic Programming with Fitness based on Model Checking. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 114–124, Valencia, Spain, 11-13 April 2007. Springer.

[89] Michael I. Jordan. Constrained Supervised Learning. *Journal of Mathematical Psychology*, 36(3):396–425, 1992.

[90] Armand Joulin and Tomas Mikolov. Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 190–198, Cambridge, MA, USA, 2015. MIT Press.

[91] Gopal K. Kanji. *100 Statistical Tests*. SAGE Publications, 2006.

[92] Susumu Katayama. Systematic Search for Lambda Expressions. In *Trends in Functional Programming*, pages 111–126, 2005.

[93] Susumu Katayama. Efficient Exhaustive Generation of Functional Programs Using Monte-Carlo Search with Iterative Deepening. In Tu-Bao Ho and Zhi-Hua Zhou, editors, *PRICAI 2008: Trends in Artificial Intelligence*, pages 199–210, Berlin, Heidelberg, 2008. Springer.

[94] Gal Katz and Doron Peled. *Genetic Programming and Model Checking: Synthesizing New Mutual Exclusion Algorithms*, pages 33–47. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[95] Gal Katz and Doron Peled. MCGP: A Software Synthesis Tool Based on Model Checking and Genetic Programming. In Ahmed Bouajjani and Wei-Ngan Chin, editors, *8th International Symposium on Automated Technology for Verification and Analysis, ATVA 2010*, volume 6252 of *Lecture Notes in Computer Science*, pages 359–364, Singapore, September 21-24 2010. Springer.

[96] Gal Katz and Doron Peled. Synthesis of Parametric Programs using Genetic Programming and Model Checking. In Lukas Holik and Lorenzo Clemente, editors, Proceedings 15th International Workshop on *Verification of Infinite-State Systems,* Hanoi, Vietnam, 14th October 2013, volume 140 of *Electronic Proceedings in Theoretical Computer Science*, pages 70–84. Open Publishing Association, 2014.

[97] Gal Katz and Doron Peled. Synthesizing, correcting and improving code, using model checking-based genetic programming. *International Journal on Software Tools for Technology Transfer*, 19:449–464, 2016.

[98] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 97–117, Cham, 2017. Springer International Publishing.

[99] Andy J. Keane. The Design of a Satellite Boom with Enhanced Vibration Performance using Genetic Algorithm Techniques. *Journal of the Acoustical Society of America*, 99(4):2599–2603, 1996.

[100] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint*, 2017.

[101] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. SeL4: Formal Verification of an Operating-System Kernel. *Communications of the ACM*, 53(6):107–115, jun 2010.

[102] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive Program Repair. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 217–233, Cham, 2015. Springer International Publishing.

[103] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[104] Natalio Krasnogor and Jim Smith. A Tutorial for Competent Memetic Algorithms: Model, Taxonomy, and Design Issues. *EEE Transactions on Evolutionary Computation*, 9(5):474–488, October 2005.

[105] Krzysztof Krawiec. *Behavioral Program Synthesis with Genetic Programming*, volume 618 of *Studies in Computational Intelligence*. Springer International Publishing, 2016. http://www.cs.put.poznan.pl/kkrawiec/bps.

[106] Krzysztof Krawiec, Iwo Błądek, and Jerry Swan. Counterexample-Driven Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 953–960, New York, NY, USA, 2017. ACM.

[107] Krzysztof Krawiec, Iwo Błądek, Jerry Swan, and John H. Drake. Counterexample-Driven Genetic Programming: Stochastic Synthesis of Provably Correct Programs. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 5304–5308. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[108] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural Random-Access Machines. *arXiv preprint*, 2016.

[109] William La Cava, Thomas Helmuth, Lee Spector, and Jason H. Moore. A Probabilistic and Multi-Objective Analysis of Lexicase Selection and ϵ-Lexicase Selection. *Evolutionary Computation*, 27(3):377–402, 09 2019.

[110] William La Cava, Lee Spector, and Kourosh Danai. Epsilon-Lexicase Selection for Regression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, page 741–748, New York, NY, USA, 2016. Association for Computing Machinery.

[111] Jean-Baptiste Lamarck. *Philosophie zoologique*. Paris, Duminil-Lesueur, 1809.

[112] Jean-Baptiste Lamarck. *Zoological philosophy; an exposition with regard to the natural history of animals*. New York,Hafner Pub. Co., 1963. https://www.biodiversitylibrary.org/bibliography/6432.

[113] Leslie Lamport. The TLA+ Home Page. http://lamport.azurewebsites.net/tla/tla.html. Accessed: 2022-01-14.

[114] William B. Langdon, Brian Yee Hong Lam, Marc Modat, Justyna Petke, and Mark Harman. Genetic improvement of GPU software. *Genetic Programming and Evolvable Machines*, 18(1):5–44, March 2017.

[115] Tessa Lau. *Programming by Demonstration: a Machine Learning Approach*. PhD thesis, University of Washington, Seattle, 2001.

[116] Tessa Lau, Pedro Domingos, and Daniel S. Weld. Learning Programs from Traces Using Version Space Algebra. In *Proceedings of the 2nd International Conference on Knowledge Capture*, K-CAP '03, page 36–43, New York, NY, USA, 2003. Association for Computing Machinery.

[117] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. Deep Learning for Source Code Modeling and Generation. *ACM Computing Surveys*, 53(3):1–38, Jul 2020.

[118] Gérard Le Lann. An Analysis of the Ariane 5 Flight 501 Failure - a System Engineering Perspective. In *Proceedings of the 1997 International Conference on Engineering of Computer-Based Systems*, ECBS'97, page 339–346, USA, 1997. IEEE Computer Society.

[119] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, jul 2009.

[120] Yunyao Li, Huahai Yang, and Hosagrahar V. Jagadish. NaLIX: An Interactive Natural Language Interface for Querying XML. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 900–902, New York, NY, USA, 2005. Association for Computing Machinery.

[121] Hod Lipson and Jordan Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, 406:974–978, 2000.

[122] Paweł Liskowski, Iwo Błądek, and Krzysztof Krawiec. Neuro-Guided Genetic Programming: Prioritizing Evolutionary Search with Neural Networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, page 1143–1150, New York, NY, USA, 2018. Association for Computing Machinery.

[123] Jason Lohn and Silvano Colombano. A Circuit Representation Technique for Automated Circuit Design. *IEEE Transactions on Evolutionary Computation*, 3(3):205–219, 1999.

[124] Jason Lohn, Gregory Hornby, and Derek Linden. An Evolved Antenna for Deployment on Nasa's Space Technology 5 Mission. In Una-May O'Reilly, Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 18, pages 301–315. Springer, Ann Arbor, 13-15 May 2004.

[125] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[126] Sean Luke and Liviu Panait. Lexicographic Parsimony Pressure. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 829–836, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[127] Zohar Manna and Richard Waldinger. Synthesis: Dreams $\Longrightarrow$ Programs. *IEEE Transactions on Software Engineering*, SE-5(4):294–328, 1979.

[128] Zohar Manna and Richard Waldinger. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, January 1980.

[129] Zohar Manna and Richard Waldinger. Fundamentals of Deductive Program Synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, 1992.

[130] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, 1999.

[131] Henry Massalin. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, page 122–126, Washington, DC, USA, 1987. IEEE Computer Society Press.

[132] Paul Massey, John A. Clark, and Susan Stepney. Evolution of a Human-Competitive Quantum Fourier Transform Algorithm Using Genetic Programming. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, page 1657–1663, New York, NY, USA, 2005. Association for Computing Machinery.

[133] Paul Massey, John A. Clark, and Susan A. Stepney. Human-Competitive Evolution of Quantum Computing Artefacts by Genetic Programming. *Evolutionary Computation*, 14(1):21–40, March 2006.

[134] James McDermott, David R. White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaśkowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, and Una-May O'Reilly. Genetic Programming Needs Better Benchmarks. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, page 791–798, New York, NY, USA, 2012. Association for Computing Machinery.

[135] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[136] Bertrand Meyer. Design by Contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.

[137] Risto Miikkulainen, Bobby D. Bryant, Ryan Cornelius, Igor V. Karpov, Kenneth O. Stanley, and Chern Han Yong. Computational Intelligence in Games. In Gary Y. Yen and David B. Fogel, editors, *Computational Intelligence: Principles and Practice*. IEEE Computational Intelligence Society, Piscataway, NJ, 2006.

[138] Julian F. Miller. An Empirical Study of the Efficiency of Learning Boolean Functions Using a Cartesian Genetic Programming Approach. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*, GECCO'99, page 1135–1142, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[139] David J. Montana. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3(2):199–230, June 1995.

[140] Pablo Moscato. On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts - Towards Memetic Algorithms. Technical report, California Institute of Technology, 1989.

[141] Wiliam J. Murdoch, Chandan Singh, Karl Kumbier, Reza Abbasi-Asl, and Bin Yu. Definitions, methods, and applications in interpretable machine learning. *Proceedings of the National Academy of Sciences*, 116(44):22071–22080, 2019.

[142] Elon Musk. An Integrated Brain-Machine Interface Platform With Thousands of Channels. *Journal of Medical Internet Research*, 21(10):e16194, Oct 2019.

[143] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58(4):66–73, March 2015.

[144] Michael A. Nielsen. Neural Networks and Deep Learning, 2018.

[145] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[146] Michael O'Neill and Conor Ryan. Grammatical Evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[147] Patryk Orzechowski, William La Cava, and Jason H. Moore. Where Are We Now? A Large Benchmark Study of Recent Symbolic Regression Methods. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, page 1183–1190, New York, NY, USA, 2018. Association for Computing Machinery.

[148] Ben Paechter, R. C. Rankin, Andrew Cumming, and Terence C. Fogarty. Timetabling the Classes of an Entire University with an Evolutionary Algorithm. In Agoston E. Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature — PPSN V*, pages 865–874, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[149] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 1–11, New York, NY, USA, 2015. Association for Computing Machinery.

[150] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[151] Emilio Parisotto, Abdel rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-Symbolic Program Synthesis. *arXiv preprint*, 2016.

[152] David Lorge Parnas. Software Aspects of Strategic Defense Systems. *Communications of the ACM*, 28(12):1326–1335, December 1985.

[153] Rodolfo A. Pazos R., Juan J. González B., Marco A. Aguirre L., José A. Martínez F., and Héctor J. Fraire H. *Natural Language Interfaces to Databases: An Analysis of the State of the Art*, pages 463–480. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[154] Dulce G. Pereira, Anabela Afonso, and Fátima Melo Medeiros. Overview of Friedman's Test and Post-hoc Analysis. *Communications in Statistics - Simulation and Computation*, 44(10):2636–2653, 2015.

[155] Brenden K. Petersen, Mikel Landajuela Larma, Terrell N. Mundhenk, Claudio Prata Santiago, Soo Kyung Kim, and Joanne Taery Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In *International Conference on Learning Representations*, 2021.

[156] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation*, 22(3):415–432, June 2018.

[157] Thu Pham-Gia and Tran Loc Hung. The Mean and Median Absolute Deviations. *Mathematical and Computer Modelling*, 34(7):921–936, 2001.

[158] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 297–310, New York, NY, USA, 2016. Association for Computing Machinery.

[159] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Published via `http://lulu.com`, 2008.

[160] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 522–538, New York, NY, USA, 2016. Association for Computing Machinery.

[161] Oleksandr Polozov and Sumit Gulwani. FlashMeta: A Framework for Inductive Program Synthesis. In *OOPSLA 2015 Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, October 2015.

[162] Steffen Priesterjahn, Oliver Kramer, Alexander Weimer, and Andreas Goebels. Evolution of Human-Competitive Agents in Modern Computer Games. In *2006 IEEE International Conference on Evolutionary Computation*, pages 777–784, 2006.

[163] John Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[164] Jerome Radcliffe. Hacking medical devices for fun and insulin: breaking the human SCADA system, 2011. https://infocondb.org/con/black-hat/black-hat-usa-2011/hacking-medical-devices-for-fun-and-insulin-breaking-the-human-scada-system.

[165] Mukund Raghothaman and Abhishek Udupa. Language to Specify Syntax-Guided Synthesis Problems, 2014.

[166] Esteban Real, Chen Liang, David R. So, and Quoc V. Le. AutoML-Zero: Evolving Machine Learning Algorithms From Scratch. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 8007–8019. PMLR, 2020.

[167] Ingo Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Number 15 in Problemata. Frommann-Holzboog, Stuttgart-Bad Cannstatt, 1973.

[168] Scott Reed and Nando de Freitas. Neural Programmer-Interpreters. *arXiv preprint*, 2016.

[169] Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark Barrett, and Morgan Deters. Refutation-Based Synthesis in SMT. *Formal Methods in System Design*, Feb 2017.

[170] John C. Reynolds. Separation Logic: a Logic for Shared Mutable Data Structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[171] Charles Rich and Richard C. Waters. Automatic Programming: Myths and Prospects. *Computer*, 21(8):40–51, August 1988.

[172] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009.

[173] Michael Schmidt and Hod Lipson. Distilling Free-Form Natural Laws from Experimental Data. *Science*, 324(5923):81–85, 2009.

[174] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, USA, 2008.

[175] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching Concurrent Data Structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 136–148, Tucson, AZ, USA, 2008.

[176] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415, San Jose, CA, USA, 2006.

[177] Kenneth Sörensen. Metaheuristics - the metaphor exposed. *International Transactions in Operational Research*, 22(1):3–18, 2015.

[178] Lee Spector. Assessment of Problem Modality by Differential Performance of Lexicase Selection in Genetic Programming: A Preliminary Report. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '12, page 401–408, New York, NY, USA, 2012. Association for Computing Machinery.

[179] Lee Spector, Howard Barnum, and Herbert Bernstein. Genetic Programming for Quantum Computers. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 365–374, 1998.

[180] Lee Spector, Howard Barnum, Herbert Bernstein, and Nikhil Swamy. Finding a Better-than-Classical Quantum AND/OR Algorithm using Genetic Programming. In *Proceedings of the 1999 Congress on Evolutionary Computation*, volume 3, pages 2239–2246, 1999.

[181] Lee Spector and Alan Robinson. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, March 2002.

[182] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. Path-based Inductive Synthesis for Program Inversion. In *PLDI'11, June 4-8, 2011, San Jose, California, USA*, June 2011.

[183] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From Program Verification to Program Synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 313–326, New York, NY, USA, 2010. ACM.

[184] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Template-Based Program Verification and Program Synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5):497–518, January 2012.

[185] Kenneth O. Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1:24–35, 2019.

[186] Marcin Szubert, Wojciech Jaśkowski, and Krzysztof Krawiec. Coevolutionary Temporal Difference Learning for Othello. In *IEEE Symposium on Computational Intelligence and Games*, pages 104–111, Milano, Italy, 2009.

[187] Simon Thibault, Christian Gagné, Julie Beaulieu, and Marc Parizeau. Evolutionary Algorithms Applied to Lens Design: Case Study and Analysis. In Laurent Mazuray and Rolf Wartmann, editors, *Optical Design and Engineering II*, volume 5962, pages 66 – 76. International Society for Optics and Photonics, SPIE, 2005.

[188] Emina Torlak and Rastislav Bodik. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 135–152, New York, NY, USA, 2013. Association for Computing Machinery.

[189] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 01 1937.

[190] Silviu-Marian Udrescu and Max Tegmark. AI Feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16), 2020.

[191] Renu Vyas, Purva Goel, and Sanjeev S. Tambe. *Genetic Programming Applications in Chemical Sciences and Engineering*, pages 99–140. Springer International Publishing, Cham, 2015.

[192] Yiqun Wang, Nicholas Wagner, and James M. Rondinelli. Symbolic regression in materials science. *MRS Communications*, 9(3):793–805, 2019.

[193] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically Finding Patches Using Genetic Programming. ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[194] Darrell Whitley. The *GENITOR* Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best. In *Proceedings of the Third International Conference on Genetic Algorithms*, page 116–121, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[195] Frank Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[196] Sewall Wright. The roles of mutation, inbreeding, cross-breeding, and selection in evolution. In *Proceedings of 6th International Congress on Genetics*, volume 1, pages 356–366, Ithaca, NY, 1932.

[197] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[198] Jie Yao, Nawwaf Kharma, and Peter Grogono. A Multi-Population Genetic Algorithm for Robust and Fast Ellipse Detection. *Pattern Analysis & Applications*, 8(1–2):149–162, September 2005.

[199] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, pages 54–66, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[200] Dennis Yurichev. SAT/SMT by example. https://sat-smt.codes/. Accessed: 2022-01-14.