TOWARDS MORE SCALABLE AND PRACTICAL PROGRAM SYNTHESIS

by

Yanjun Wang

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

School of Electrical and Computer Engineering

West Lafayette, Indiana

May 2022

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Xiaokang Qiu, Chair

School of Electrical and Computer Engineering

Dr. Milind Kulkarni

School of Electrical and Computer Engineering

Dr. Sanjay G. Rao

School of Electrical and Computer Engineering

Dr. Suresh Jagannathan

Department of Computer Science

Approved by:

Dr. Dimitrios Peroulis

To my parents.

ACKNOWLEDGMENTS

Pursuing Ph.D. is a journey full of adventures and surprises. To this day, It's still hard for me to believe that it's coming to an end. I wish to thank all the people who helped me throughout this journey.

Words are powerless to express my gratitude to my advisor, Xiaokang Qiu. I feel extremely lucky to be his student and have the privilege to work with him during the past six years. He is the best advisor I could ever ask for. I cannot thank him enough for bringing me to the world of research, which has certainly reshaped my life. He has taught me most of what I know today about programming languages and all I know of being a good researcher. He provided a research environment in which I could thrive. Whenever I need help, he never holds back his time or efforts. His enthusiasm for research has inspired and will continue inspiring me to become a better researcher in the future. He is more than an ideal advisor, he is a great friend, who has made my Ph.D. journey an enjoyable, rewarding and regretless experience. I would not be where I am today without him.

I would also like to thank Milind Kulkarni, Sanjay Rao and Suresh Jagannathan for being on my dissertation committee. I am grateful for the time they spent reading my preliminary report and this dissertation. Their many insightful comments and helpful suggestions helped me broaden and deepen my sense of research. Moreover, I greatly appreciate the guidance and assistance that Sanjay has offered, particularly in the topics of network synthesis.

I would like to thank all of my collaborators: Sanjay Rao, Zixuan Li, Chuan Jiang, Kangjing Huang, Peiyuan Shen, Dalin Zhang, Jinwei Liu and all others, for their contributions to this dissertation. I would also like to express my appreciation to my lab mates at Purdue CAP for providing invaluable input and comments in lab meetings and research discussions.

Next, there are the friends who helped me maintain my sanity through the tough times. Thank you for bringing happiness and joy to my life. Among them, I want to express my special gratitude to Mengyue Hang for her priceless friendship, encouragement and support. I owe much thanks to brothers and sisters in Greater Lafayette Chinese Alliance Church. I

am thankful for all their prayer, sharing and continuous love, making my spiritual journey in the Midwest much more abundant than I expected.

Last but not least, I am forever grateful for the unconditional love and unfailing support from my parents. I cannot be more fortunate to be always loved by you, no matter who I am and no matter what I do. Thank you for supporting me every step of the way. This dissertation is dedicated to you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Program synthesis aims to generate programs automatically from user-provided specifications and has the potential to aid users in real-world programming tasks from different domains. Although there have been great achievements of synthesis techniques in specific domains such as spreadsheet programming, computer-aided education and software engineering, there still exist huge barriers that keep us from achieving scalable and practical synthesis tools.

This dissertation presents several techniques towards more scalable and practical program synthesis from three perspectives: 1) *intention*: Writing formal specification for synthesis is a major barrier for average programmers. In particular, in some quantitative synthesis scenarios (such as network design), the first challenge faced by users is expressing their optimization targets. To address this problem, we present comparative synthesis, an interactive synthesis framework that learns near optimal programs through comparative queries, without explicitly specified optimization targets. 2) *invention*: Synthesis algorithms are key to pushing the performance limit of program synthesis. Aiming to solve syntax-guided synthesis problems efficiently, we introduce a cooperative synthesis technique that combines the merits of enumerative and deductive synthesis. 3) *adaptation*: Besides functional correctness, quality of generated code is another important aspect. Towards automated provably-correct optimization over tree traversals, we propose a stack-based representation for iterations in tree traversals and an encoding to Monadic Second-Order logic over trees, which enables reasoning about tree traversal transformations which were not possible before.

1. INTRODUCTION

Automation of programming has been a dream longed for a long time. As one of the effective approaches to automated programming, program synthesis has attracted significant attention in recent years. Program synthesis is the task that aims to generate implementations of programs automatically from user-provided intentions specified in some form of specifications. By unburdening programmers' efforts to produce programs with every detail, program synthesis has been identified as having the potential to impact software development dramatically. Thanks to the advancements in automated reasoning tools in recent years, program synthesis has a surge of practical efforts in a variety of applications ranging from spreadsheet programming, computer-aided education, software engineering.

Unfortunately, due to the inherent challenges in program synthesis, there is still a long way to go for program synthesis to be used more widely. To facilitate this goal, two central problems need to be considered.

The first challenge is *practicality*. When dealing with real-life synthesis tasks, one of the challenges is expressing the user's intent and interpreting it accurately. Program synthesis systems usually expect users tell the machine what they need via some form of specifications. One of the widely adopted specifications is logical predicate, which on it own is challenging for users who do not have any programming expertise to construct. Besides formal specifications, informal ones, such as input-output examples, execution traces and partial programs etc., often cause ambiguity on the expected behavior of the synthesized programs. In many cases, user intent can be too complicated to be captured by any existing informal specifications. How to carefully design interactions between user and the synthesis system in a way that neither need programming expertise nor cause ambiguity is a major challenge in program synthesis.

The second major problem is *scalability*. The goal of program synthesis is finding a program that satisfies a set of constraints on desired behavior, from a given search space. In that sense, program synthesis problems are often framed as search problems and are usually discharged by performing some kind of search over the spaces of candidate programs. Since the number of potential programs increases exponentially as the size of programs grows, it is

notoriously hard for program synthesizers to scale to complex programs. How to exploit the program space efficiently and find the desired implementation effectively becomes an intense research topic.

Besides the scalability of the synthesizer, the scalability of the synthesized programs is another important aspect to be considered. Generating an arbitrary program that satisfy the functional specification is sometimes not enough. There can be many different programs that satisfy the same functional specification, where user expect a high quality program in terms of program size, execution time or readability etc. How to generate programs of high quality and how to automatically adapt user-unfriendly programs to user-friendly programs are problems that have not been fully resolved yet.

The challenges in program synthesis closely relate to three technical pillars of machine programming [1]: *intention*, *invention* and *adaptation*. Each pillar focuses on one of three components of program synthesis systems: input specification, synthesizer and output program, respectively. *Intention* emphasizes on capturing user's intention. People find it easier to provide informal specifications instead of formal ones. Based on this observation, FlashFill [2] system embedded in Microsoft Excel uses input-output examples to construct string transformation programs. Similarly, the SKETCH system [3] takes a partial program template (program sketch) as input and generates the unknowns in the program sketch. While these informal specifications are effective in specific domains, they are not applicable or sufficient in other domains. Research in *invention* pillar emphasizes the design of algorithms used for accomplishing synthesis tasks. The search techniques used in program synthesis are typically based on enumeration [4], deduction [5]–[7], constraint solving [3] or machine learning [8]. It requires discovering new algorithms, in order to improve the efficiency of existing techniques. The *adaptation* pillar emphasizes on developing techniques that adapt or maintain the software to run efficiently. Program optimization techniques [9]–[11] facilitate programs running efficiently on one or different platforms. Synthesizing program optimizations automatically and proving the validity of transformations becomes another burgeoning research area.

In this thesis, we present a suite of techniques aiming *towards more scalable and practical program synthesis*. Specifically, the techniques proposed in this thesis achieves the following:

- Introduce a novel human-computer interface to learn near-optimal programs.
- Increase the scalability of syntax-guided synthesizers.
- Enable the capability of verifying more sophisticated tree traversal optimizations.

Below, we provide an overview for each direction.

1.1 Learning Near-optimal Programs Through Comparative Queries

Our work presented in Chapter 2 considers a novel way to interact with users when input-output examples are hard to obtain. We consider the quantitative synthesis problems in network design domain, where the first challenge faced by users is expressing their optimization targets in the form of either closed-form functions or input-output examples. We propose comparative synthesis [12], an interactive synthesis framework which produces near-optimal programs (network designs) through two kinds of comparative queries (VALIDATE and COMPARE), without an objective function explicitly given. The key idea is to make comparative queries to learn the user's preference over candidate programs, with which objectives can be conjectured. These objectives, which are indeterminate as they can be refined along the course of user interaction, guide the search of satisfying programs.

Within the comparative synthesis framework, we developed the first learning algorithm for comparative synthesis in which a voting-guided learner picks the most informative query in each iteration. We present theoretical analysis of the convergence rate of the algorithm and identify a class of comparative synthesis problems on which our algorithm converges faster. We implemented NET10Q, a system based on our approach, and demonstrate its effectiveness on four real-world network case studies using black-box oracles and simulation experiments, as well as a pilot user study comprising network researchers and practitioners. Experiments show that our framework can successfully synthesize satisfying solutions by making a budgeted number of queries, without a priori knowledge about the quantitative objective.

1.2 More Scalable Syntax-Guided Synthesis

In Chapter 3, we present a novel cooperative synthesis framework [13] designed for the syntax-guided synthesis problem. As one of the critical thrusts of program synthesis, Syntax-guided Synthesis (SyGuS) allows user to provide syntactic constraints along with the semantic specifications, so that the correctness of implementations are ensured as long as the software quality. Currently, there are two main synthesis techniques adopted by most synthesizers: enumerative synthesis and deductive synthesis. Enumerative synthesis repeatedly enumerates possible candidate programs following a specific order and checks those programs satisfy the synthesis constraints. Enumerative synthesis is very generally applicable but becomes inefficient as search space grows. Deductive synthesis is usually very efficient, as it tries to reduce the specification to the desired implementation. Since deductive synthesis relies on a set of pre-defined deduction rules, it is hard to extend to other grammars or applications.

In order to combine the best of the two worlds, we propose a cooperative synthesis technique for SyGuS problems with the conditional linear integer arithmetic (CLIA) background theory, as a novel integration of the two approaches. The technique exploits several novel divide-and-conquer strategies to split a large synthesis problem to smaller subproblems. The subproblems are solved separately and their solutions are combined to form a final solution. The technique integrates two synthesis engines: a pure deductive component that can efficiently solve some problems, and a height-based enumeration algorithm that can handle arbitrary grammar. We implemented the cooperative synthesis technique, and evaluated it on a wide range of benchmarks. Experiments showed that our technique can solve many challenging synthesis problems not possible before, and tends to be more scalable than state-of-the-art synthesis algorithms.

1.3 Verifying Tree Traversal Transformations

In Chapter 4, we consider the problem of reasoning about tree traversals, which paves the way to automated tree traversal transformations. Traversals are commonly seen in tree data structures, and performance-enhancing transformations between tree traversals are critical

for many applications. Existing approaches to reasoning about tree traversals and their transformations are ad hoc, with various limitations on the classes of traversals they can handle, the granularity of dependence analysis, and the types of possible transformations. We propose RETREET [14], a framework in which one can describe general recursive tree traversals, precisely represent iterations, schedules and dependences, and automatically check data-race-freeness and transformation correctness. The crux of the framework is a stack-based representation for iterations and an encoding to Monadic Second-Order (MSO) logic over trees. Experiments show that RETREET can automatically verify optimizations for complex traversals on real-world data structures, such as CSS and cycletrees, which are not possible before. Our framework is also integrated with other MSO-based analysis techniques to verify even more challenging program transformations.

1.4 Thesis Organization

The rest of this document is organized as follows.

- In Chapter 2, we propose comparative synthesis, an interactive synthesis framework, that learns user intent by making comparative queries, which produces near-optimal programs through queries, without specifying an objective function.
- In Chapter 3, We present a cooperative synthesis technique that combines enumerative and deductive synthesis for SYGUS problems with conditional linear integer arithmetic (CLIA) background theory.
- In Chapter 4, we introduce a framework in which one can describe general tree traversals and automatically reason about data-race-freeness and transformation correctness.
- Finally, Chapter 5 concludes with a summary of our contributions and looks ahead to future research directions.

2. COMPARATIVE SYNTHESIS: LEARNING NEAR-OPTIMAL NETWORK DESIGNS BY QUERY

In this chapter, we present an interactive synthesis framework that learns near optimal program for quantitative synthesis problems through comparative queries.

2.1 Introduction

Synthesizing wide-area computer network designs typically involves solving *multi-objective optimization* problems. For instance, consider the task of managing the traffic of a wide-area network — deciding the best routes and allocating bandwidth for them — the architect must consider myriad considerations. She must choose from different routing approaches — e.g., shortest path routing [15], and routing along pre-specified paths [16], [17]. The traffic may correspond to different classes of applications — e.g., latency-sensitive applications such as Web search and video conferencing, and elastic applications such as video streaming, and file transfer applications [16], [18], [19]. The architect may need to decide how much traffic to admit for each class of applications. It is desirable to make decisions that can ensure high throughput, low latency, and fairness across different applications, yet not all these goals may be simultaneously achievable. Likewise, a network must not only perform acceptably under normal conditions, but also under failures — however, providing guaranteed performance under failures may require being sacrificing normal performance [17], [20].

Traffic engineering formulates network design problems as optimization problems [15], [18]–[21], e.g., minimizing a weighted sum of link utilization and latency subject to constraints. In this context, architect must provide the objectives as well-defined mathematical functions (which we henceforth refer to as *target functions*), which is a challenging task in the first place. Even the simplest target functions may involve several knobs to capture the relative importance of different criteria (e.g., throughput, latency, and fairness, performance under normal conditions vs. failures). These knobs must be manually tuned by the architect in a "trial and error" fashion to result in a desired design. Further, many optimization problems (e.g., [19]) require architects to use abstract functions that capture the utility an

application sees if a given design is deployed. Utility functions are often non-linear (e.g., logarithmic) and may involve weights, which are not intuitive for a designer to specify in practice [22]. Finally, objectives are often chosen in a manner to ensure tractability, rather than necessarily reflecting the true intent of the architect.

This paper presents one of the first attempts to *learn near-optimal network designs with indeterminate objectives*. Our work adopts an interactive, program-synthesis-based approach based on the key insight that when a user has difficulty in providing a concrete objective, it is relatively easy and natural to give preferences between pairs of concrete candidates. The approach may be viewed as a new variant of programming-by-example (PBE), where preference pairs are used as "examples" instead of input-output pairs in traditional PBE systems.

In this paper, we make the following contributions:

• **A novel user-interaction paradigm (§2.3).** We present a rigorous formulation of an interactive synthesis framework which we refer to as **comparative synthesis**. As Fig 2.1 shows, the framework consists of two major components: a comparative learner and a teacher (a user or a black-box oracle). The learner takes as input a clearly defined qualitative synthesis problem (including a parameterized program and a specification), a metric group and a target function space, and is tasked to find a near-optimal program w.r.t. the teacher's quantitative intent through two kinds of queries — VALIDATE and COMPARE.

The notion of comparative synthesis stems from a recent position paper [23]. The preliminary work lacks formal foundation and query selection guidance, and may involve impractically many rounds of user interaction (see §2.2). In contrast, the formalism of our framework enables the design and analysis of learning algorithms that strive to minimize the number of queries, and are amenable for real user interaction.

• **The first algorithm for comparative synthesis (§2.4).** We develop the first, voting-guided learning algorithm for comparative synthesis, which provides a provable guarantee on the quality of the found program. The key insight behind the algorithm is that objective learning and program search are mutually beneficial and should be done in tandem. The idea of the algorithm is to search over a special, unified search space we call *Pareto*

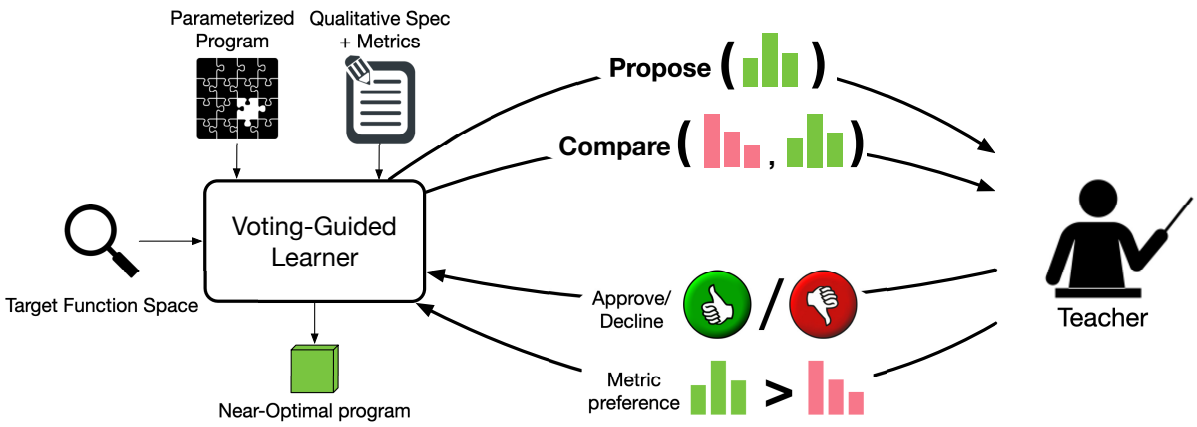


Figure 2.1. Overview of comparative synthesis.

candidate set, and to pick the *most informative* query in each iteration using a voting-guided estimation.

We analyze the *convergence* of voting-guided algorithm, i.e., how fast the solution approaches the real optimal as more queries are made. We prove that the algorithm guarantees the median quality of solutions to converge logarithmically to the optimal. When the target function space is sortable, which covers a commonly seen class of problems, a better convergence rate can be achieved — the median quality of solutions converges linearly to the optimal.

• **Evaluations on network case studies and pilot user study (§2.5).** We developed NET10Q, an *interactive network optimization system* based on our approach. We evaluated NET10Q on four real-world scenarios using oracle-based evaluation. Experiments show that NET10Q only makes half or less queries than the baseline system to achieve the same solution quality, and robustly produces high-quality solutions with inconsistent teachers. We conducted a pilot study with NET10Q among networking researchers and practitioners. Our study shows that user policies are diverse, and NET10Q is effective in finding allocations meeting the diverse policy goals in an interactive fashion.

A Lookahead: While our motivation and evaluation are from the context of network design, the challenge of indeterminate objectives is commonly seen in many quantitative

synthesis problems beyond the networking domain. For example, the default ranker for the FlashFill synthesizer is manually designed and highly tuned by experts [24]. In quantitative syntax-guided synthesis (qSyGuS) [25], the objective should be provided as a weighted grammar, which is nontrivial for average programmers. Therefore, the problem we address in this paper can be viewed as an instance of *specification mining* [26], a long-standing problem in the formal methods community which recognizes that a precise specification may not always be available. The key contributions of the paper, including comparison-based interaction (§2.3) and the voting-guided algorithm (§2.4), are thus potentially applicable in other more traditional program synthesis domains in the future.

2.2 Motivation

In this section, we present background on network design, how it may be formulated as a program synthesis problem, and discuss challenges that we propose to tackle.

Network design background. In designing Wide-Area Networks (WANs), Internet Service Providers (ISPs) and cloud providers must decide how to provision their networks, and route traffic so their traffic engineering goals are met. Typically WANs carry multiple classes of traffic (e.g., higher priority latency sensitive traffic, and lower priority elastic traffic). Traffic is usually specified as a matrix with cell (i, j) indicating the total traffic which enters the network at router i and that exits the network at router j. We refer to each pair (i, j) as a flow, or a source-destination pair. It is typical to pre-decide a set of tunnels (paths) for each flow, with traffic split across these tunnels in a manner decided by the architect, though traffic may also be routed along a routing algorithm that determines shortest paths (§2.5.1).

Given constraints on link capacities, it may not be feasible to meet the requirements of all traffic of all flows. An architect must decide how to allocate bandwidth to different flows of different classes and how to route traffic (split each flow's traffic across it paths) so desired objectives are met. In doing so, an architect must reconcile multiple metrics including throughput, latency, and link utilizations [16], [18], [27], [28], ensure fairness across flows [19], [22], [29], and consider performance under failures [17], [20], [21], [30], [31].

Network design as program synthesis problems. Consider a variant of the classical multi-commodity flow problem used in Microsoft's Software Defined Networking Controller SWAN [16], which we refer to as MCF. MCF allocates traffic to tunnels optimally trading off the total throughput seen by all flows with the weighted average flow latency [16]. We consider a single class (see §2.5.1 for multiple classes).

Fig 2.2 shows how the demand-capacity constraints may be described as a *sketch-based synthesis* problem, in which the programmer specifies a sketch — a program that contains unknowns to be solved for, and assertions to constrain the choice of unknowns. The Topology struct represents the network topology (we use the Abilene topology [32] with 11 nodes, 14 links and 220 flows as a running example). The allocate function should determine the bandwidth allocation (denoted by **??**), which is missing and should be generated by the synthesizer. The function also serves as a test harness and checks that the synthesized allocation is valid, satisfying all demand and capacity constraints (lines 12–13). Finally, the main function takes the generated allocation, and computes and returns the total throughput and weighted latency.

Now Fig 2.2 has encoded all *hard constraints* and represented a *qualitative synthesis* problem, which can be solved by Sketch [3] easily. The bandwidth allocations generated by the synthesizer (the values of bw) is just a network design solving the MCF problem.[1] However, there are many different ways to fill the **??**, corresponding to many different ways of assigning paths and leading to different throughput-latency combinations as computed in main. Which solution is the most desirable one? Traditionally, the architect has to explicitly provide a *target function* which maps each possible solution to a numerical value indicating the preference. Given a well-specified target function, the bandwidth allocation problem becomes a *quantitative synthesis* problem and can be solved using existing techniques from both synthesis and optimization communities. E.g., in Fig 2.2, one can explicitly add a target function O_{real} and use the minimize construct (cf. Sketch manual [33]) to find the optimal solution.

[1]↑We will use network design and network program interchangeably in the paper, as network design can always be extracted from the synthesized network program.

```
struct Topology {
  int n_nodes; int n_links; int n_flows;
  bit[n_nodes][n_nodes] links;
  /* every link has a capacity and a weight, every flow consists of multiple links and
     has a demand */
  float[n_links] capacity; int[n_links] wght;
  bit[n_links][n_flows] l_in_f; float[n_flows] demand; ...
}
Topology abilene = new Topology(n_nodes=11, n_links=220, ...);
float[] allocate(Topology T) {
  float[T.n_flows] bw = ??; // generate bandwidth allocation
  /* assert that every flow's demand is satisfied and every link's bandwidth is not
     exceeded */
  assert ⋀_i bw[i] <= T.demand[i];
  assert ⋀_j ( Σ_i l_in_f[j][i] ? bw[i] : 0 ) ≤ T.capacity[j];
  return bw; }
float[] main() {
  float[] alloc = allocate(abilene);
  /*compute the throughput and weighted latency*/
  float throughput = Σ_i alloc[i];
  float latency = Σ_i ( alloc[i] · Σ_j l_in_f[j][i]? wght[j] : 0 );
  return {throughput, latency}; }
```

Figure 2.2. MCF allocation encoded as a program sketch.

Why synthesis with indeterminate objectives? Unfortunately, in practice, it is hard for network architects to precisely express their true intentions using target functions. For example, to capture the intuition that once the throughput (resp. latency) reaches a certain level, the marginal benefit (resp. penalty) may be smaller (resp. larger), an architect may need to use a target function like below:

$$O_{real}(\mathsf{throughput}, \mathsf{latency}) \stackrel{\text{def}}{=} 2 \cdot \mathsf{throughput} - 9 \cdot \mathsf{latency} - \max(\mathsf{throughput} - 350, 0) - 10 \cdot \max(\mathsf{latency} - 28, 0) \quad (2.2.1)$$

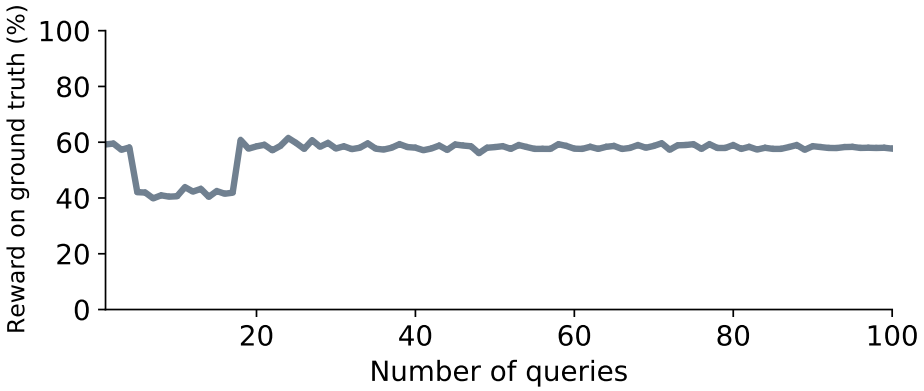


Figure 2.3. Naïve objective synthesis of O_{real}.

More generally, the marginal reward in obtaining a higher bandwidth allocation is smaller capturing which may require a target function of the form $O(\mathsf{throughput}, \mathsf{latency}) \stackrel{\text{def}}{=} 1 \cdot \log_n(\mathsf{throughput}/tmax + 1) + 5 \cdot \log_n(lmin/\mathsf{latency} + 1)$ where $tmax$ is the maximum possible throughput and $lmin$ is the minimum possible latency. Expressing such abstract target functions is not trivial, let alone the parameters associated with the functions. We present several other examples in §2.5.1.

Naïve objective synthesis is not enough. A preliminary effort [23] argued for synthesizing target functions by having the learner (synthesizer) iteratively query the teacher (user) on its preference between two concrete network designs. In each iteration, any pair of designs may be compared as long as there exist two target functions that (i) disagree on how they rate these designs, and (ii) both satisfy preferences expressed by the teacher in prior iterations. The process continues until no disagreeing target functions are found. However, this work only discusses objective learning and does not explicitly consider design synthesis. Moreover, it does not address how to generate good preference pairs to minimize queries. These limitations make this naïve approach not amenable for real user interaction. Fig 2.3 shows the performance of a design optimal for a target function synthesized using this procedure if it were terminated after a given number of queries. The resulting designs achieve a reward only 60% of the true optimal design under the ground truth (Eq 2.2.1), and there is hardly any performance improvement in the first 100 queries.

2.3 Comparative Synthesis, Formally

In this section, we provide a formal foundation for the comparative synthesis framework, based on which we design and analyze learning algorithms. The key novelty of our framework compared to past work on quantitative synthesis [25], [34]–[38] is that comparative synthesis does not require the user to explicitly specify the objective. Instead, we approach synthesis via *interaction through comparative queries* where queries simply involve the users comparing two alternative programs and indicating which is more preferable. Since a user will only be willing to answer a small number of queries and may choose to stop at any point of the interactions, finding a perfect quantitative objective can be unrealistic. Therefore, our goal is to *generate a near-optimal program within a budgeted number of queries*. As the real ground truth optimal is not accessible, we also introduce a natural notion, called *quality of solution*, to estimate how close a solution is to optimal.

Roadmap. In §2.3.1, we formally define quantitative synthesis, a necessary first step for us to formally treat comparative synthesis. Rather than restrict quantitative synthesis to objectives which are closed-form mathematical functions, we formulate quantitative synthesis more generally as we motivate and discuss in §2.3.1. We formally define comparative queries, and solution quality in §2.3.2. We conclude with a formal definition of comparative synthesis in §2.3.3.

2.3.1 Quantitative Synthesis with Metric Ranking

In this section, we present a formal definition of quantitative synthesis, as a first step towards defining comparative synthesis more precisely. Quantitative synthesis may be viewed as a goal of synthesizing a program that meets a set of "hard constraints", while performing well on a quantitative objective. Rather than restrict quantitative synthesis to objectives which are closed-form mathematical functions, we formulate quantitative synthesis when given a ranking over all possible programs (which we refer to as a *metric ranking*). This more general formulation is motivated by the fact that we wish to capture a rich set of user policies in terms of relative preferences across programs, and not restrict the user to objectives that are closed-form mathematical functions.

We start by defining qualitative synthesis (which captures the "hard constraints" that any acceptable program must meet), and then discuss quantitative synthesis with metric ranking.

Definition 2.3.1 (Qualitative synthesis problem)**.** *A qualitative synthesis problem is represented as a tuple* $(\mathcal{P}, C, \Phi)$ *where* $\mathcal{P}$ *is a parameterized program,* C *is the space of parameters for* $\mathcal{P}$*, and* Φ *is a verification condition with a single free variable ranging among* C*. The synthesis problem is to find a value* $\mathsf{ctr} \in C$ *such that* $\Phi(\mathsf{ctr})$ *is valid.*

Example 2.3.1. Our running example can be formally described as a qualitative synthesis problem $\mathscr{L}_{MCF} = (\mathcal{P}_{\mathsf{abilene}}, \mathbb{R}^{220}, \Phi_{\mathsf{abilene}})$, where $\mathcal{P}_{\mathsf{abilene}}$ is the program sketch presented in Fig 2.2, $\mathbb{R}^{220}$ is the search space of unknown hole (line 10) which includes 220 bandwidth values of the Abilene network, Φ_{abilene} is the verification condition, taking a candidate solution $c \in \mathbb{R}^{220}$ as input and checking whether $\mathcal{P}_{\mathsf{abilene}}[c]$ satisfies all assertions in Fig 2.2. Any solution that satisfies assertions in Fig 2.2 is a feasible program to the qualitative synthesis problem $\mathscr{L}_{MCF}$.

While a qualitative synthesis problem captures all hard constraints, there are potentially infinitely many solutions. Which one is the most desirable? *Quantitative synthesis* concerns itself with this question and extends a qualitative synthesis problem with a quantitative goal, which is evaluated using a metric group, as defined below.

Definition 2.3.2 (Metric)**.** *Given a parameterized program* $\mathcal{P}[c]$ *where* c *ranges from a search space* C*, a metric with respect to* $\mathcal{P}$ *is a computable function* $m_{\mathcal{P}} : C \to \mathbb{R}$*. In other words,* $m_{\mathcal{P}}$ *takes as input a concrete program in the search space and computes a real value.*

Definition 2.3.3 (Metric group)**.** *Given a parameterized program* $\mathcal{P}$*, a* d*-dimensional metric group* M *w.r.t.* $\mathcal{P}$ *is a sequence of* d *metrics w.r.t.* $\mathcal{P}$*. We write* M_{i} *for the* i*-th metric in the group and* $M(c)$ *for the value vector* $\big(M_1(c), \ldots, M_d(c)\big)$*.*

Example 2.3.2. A metric can be computed from the syntactical aspects of the program. For example, a metric $\mathsf{size}_{\mathcal{P}}(c)$ can be defined as the size of the parse tree for $\mathcal{P}[c]$.

Example 2.3.3. A metric can simply be the value of a variable on a particular input (or with no input). In our running example in Fig 2.2, the two variables throughput and latency of the main function can be used to define two metrics. As the latency as a metric is not beneficial, i.e., smaller latency is better, we can simply use its inverse -latency as a beneficial metric. The two metrics form a metric group $M_{MCF} = (\mathsf{throughput}, \mathsf{-latency})$.

Now given a metric group, the quantitative intent of a user can be captured either *syntactically* (using a target function) or *semantically* (using a metric ranking). We formally define them below and discuss their relationship.

Definition 2.3.4 (Target function)**.** *Given a metric group M, a* target function *with respect to M is a computable function $\mathbb{R}^{|M|} \to \mathbb{R}$.*

Definition 2.3.5 (Metric ranking)**.** *Given a d-dimensional metric group M, a* metric ranking *for M is a total preorder $\lesssim_M$ over $\mathbb{R}^d$. In other words, $\lesssim_M$ satisfies the following properties: for any $u \in \mathbb{R}^d$, $u \lesssim_M u$ (*reflexivity*); for any three vectors $u, v, w \in \mathbb{R}^d$, if $u \lesssim_M v$ and $v \lesssim_M w$, then $u \lesssim_M w$ (*transitivity*); for any two vectors $u, v \in \mathbb{R}^d$, $u \lesssim_M v$ or $v \lesssim_M u$ (*connexity*).*

We write $u \sim_M v$ if $u \lesssim_M v$ and $v \lesssim_M u$. In this paper, we also flexibly write $u \gtrsim_M v$, $u <_M v$, $u >_M v$ and $u \nsim_M v$ with the expected meaning. Moreover, we also abuse $\lesssim_M$ and other derived symbols we just described as relations between programs: when the metric group M is clear from the context, for any two program parameters $c_1, c_2 \in \mathrm{dom}(M)$, we write $c_1 \lesssim_M c_2$ to indicate that $M(c_1) \lesssim_M M(c_2)$.

Why metric ranking? Target function and metric ranking are closely related, but metric ranking is a more general and unique representation for quantitative intent, and can capture a richer set of user policies in terms of which of two feasible programs is preferable. First, every target function implicitly determines a metric ranking (see Def 2.3.6 below). Second, multiple target functions may have the same metric ranking. For example, any target functions O and O' such that $O'(v) = 2 \cdot O(v)$ for any $v \in \mathbb{R}^{|M|}$ have the same metric ranking. Third, some metric ranking does not correspond to any target function, e.g., one can define a metric ranking $\lesssim$ between integer metric values such that $n_1 \lesssim n_2$ if and only

if the n_1-th digit of Ω is less than or equal to the n_2-th digit of Ω, where Ω is a Chaitin's constant [39] representing the probability that a randomly generated program halts. To this end, we define quantitative synthesis problem below using metric ranking.

Definition 2.3.6. *Given a target function O w.r.t. a d-dimensional metric group M, the corresponding metric ranking $\lesssim_O \subset \mathbb{R}^d \times \mathbb{R}^d$ is defined as follows: for any two program parameters $c_1, c_2 \in dom(M)$, $c_1 \lesssim_O c_2$ if and only if $O(M(c_1)) \leq O(M(c_2))$. It can be easily verified that $\lesssim_O$ is indeed a metric ranking.*

Definition 2.3.7 (Quantitative synthesis problem)**.** *A quantitative synthesis problem is represented as a tuple $\mathscr{Q} = (\mathcal{P}, C, \Phi, M, \lesssim_M)$ where $(\mathcal{P}, C, \Phi)$ forms a qualitative synthesis problem $\mathscr{Q}^{qual}$, M is a metric group w.r.t. $\mathcal{P}$ and $\lesssim_M$ is a metric ranking for M. The synthesis problem is to find a solution ctr to $\mathscr{Q}^{qual}$ such that for any other solution $\mathsf{ctr'}$, $\mathsf{ctr'} \lesssim_M \mathsf{ctr}$.*

Example 2.3.4. With the metric group M_{MCF} defined in Example 2.3.3, the function O_{real} defined in Equation 2.2.1 is a 2-dimensional target function with the corresponding metric ranking $\lesssim_{real}$. Then the qualitative synthesis problem $\mathscr{L}_{MCF} = (\mathcal{P}_{\mathsf{abilene}}, \mathbb{R}^{220}, \Phi_{\mathsf{abilene}})$ can be extended to a quantitative synthesis problem $\mathscr{Q}_{MCF} \stackrel{\text{def}}{=} (\mathcal{P}_{\mathsf{abilene}},\ \mathbb{R}^{220},\ \Phi_{\mathsf{abilene}},\ M_{MCF},\ \lesssim_{real})$.

2.3.2 Interaction Through Comparative Queries

Quantitative synthesis problem as defined in Def 2.3.7 expects a metric ranking explicitly or implicitly (e.g., through a target function). Comparative synthesis is more challenging as it seeks to synthesize a program that is near-optimal in terms of the objective, but without being explicitly given the objective. To achieve the goal, our comparative synthesis framework is interactive between a **learner** and a **teacher** (see Fig 2.1). As the teacher may choose to stop at any point of the interactions, our comparative synthesis framework maintains the best candidate solution found through the synthesis process and recommends the best solution confirmed by the teacher when terminated.

As the quantitative objective is assumed to be very complex and not direct accessible from the teacher, the comparative learner can only make several types of queries to the

teacher, whose responses provide indirect access to the specifications. The query types serve as an interface between the learner and the teacher, and different query types lead to different synthesis power (e.g., see the query types discussed in [40]). What makes our framework special is that the learner can make queries about the *metric ranking* — queries that compare two program (based on their corresponding metric value vectors).

Let us fix a parameterized program $\mathcal{P}$ and a metric group M. The learner and the teacher interact using two types of queries: [2]

• COMPARE(c_1, c_2) **query:** The learner provides two concrete programs $\mathcal{P}[c_1]$ and $\mathcal{P}[c_2]$, and asks "Which one is better under the target metric ranking $\lesssim_M$?" The teacher responds with $<$ or $>$ if one is strictly better than the other, or $=$ if $\mathcal{P}[c_1]$ and $\mathcal{P}[c_2]$ are considered equally good.

• VALIDATE(c) **query:** [3] The learner proposes a candidate program $\mathcal{P}[c]$ and asks "Is $\mathcal{P}[c]$ better than the running best candidate *r_best*?" If the teacher finds that $\mathcal{P}[c]$ is not better than the running best, she can respond with $\bot$. Otherwise, the teacher considers that $\mathcal{P}[c]$ is better and responds with $\top$; in that case, the running best will be updated to $\mathcal{P}[c]$.

Now in comparative synthesis, the specification (a metric ranking R) is hidden to the learner. Instead, the learner can approximate/guess the specifications by making queries to the teacher. Ideally, the teacher should be *perfect*, i.e., the responses she makes to queries are always consistent and satisfiable. Formally,

Definition 2.3.8 (Perfect teacher)**.** *A teacher $\mathcal{T}$ is* perfect *if there exists a metric ranking $\lesssim_M$ such that: 1) for any query* COMPARE(v_1, v_2)*, the response is "$<$" if $v_1 \lesssim_M v_2$ and $v_2 \not\lesssim_M v_1$, or "$>$" if $v_1 \not\lesssim_M v_2$ and $v_2 \lesssim_M v_1$, or "$=$" if $v_1 \lesssim_M v_2$ and $v_2 \lesssim_M v_1$; 2) for any query* VALIDATE(c) *with the current running best r_best, the response is "$\top$" if $c >_M$ r_best; or "$\bot$" otherwise.*

We denote the perfect teacher w.r.t. $\lesssim_M$ as $\mathcal{T}_{\lesssim_M}$. We also denote the metric ranking $\lesssim_M$ represented by a perfect teacher $\mathcal{T}$ as $\lesssim_{\mathcal{T}}$. A perfect teacher guarantees that an optimal

[2]↑We believe our framework can be extended in the future to support more types of queries.

[3]↑Note that VALIDATE(c) can be viewed as a special COMPARE query between c and *r_best*, the goal of the query is slightly different: VALIDATE(c) intends to beat the running best using c, while COMPARE(c_1, c_2) intends to distinguish very close solutions c_1 and c_2 to learn the teacher's intent. COMPARE(c_1, c_2) will **not** update the running best.

solution exists among all candidates. For now, let us assume that the teacher is perfect, i.e., consistent and able to answer all queries; but in the real world, a human teacher may be inconsistent and responds incorrectly. We do consider imperfect teachers in our evaluation (see §2.5.3).

Why budgeted number of queries? Ideally, the goal of the learner is to find an objective target (in the form of target function or metric ranking) that matches the teacher's mind and the corresponding optimal program that optimizes the objective. However, finding the target function can be impossible as the objective target may have no closed-form representation and not in the target function space. As the teacher is free to terminate the synthesis process at any point, pinpointing the target function in a potentially infinitely large space can also be impossible, even if the target function is in the target function space.

Therefore, the goal of the learner is *to spend a budgeted number of queries and to produce a near-optimal program from the perspective of the teacher*. Note that the learner may use a conjectured objective to guide the search process, finding a perfect target function is not a goal. This is also a key insight for our algorithm design (cf. §2.4).

Now to determine how close a solution is to the ground truth optimal, we introduce a natural notion called *quality of solution* which is intuitively the "relative rank" of the solution among all solutions. E.g., a solution of quality 0.9 is better than or equal to 90% of possible solutions. From a probability theory point of view, the quality is just the cumulative distribution function (CDF). Below we formally define the quality of solutions.

Definition 2.3.9 (Quality of solution)**.** *Let $\mathscr{Q} = (\mathcal{P}, C, \Phi, M, \lesssim_M)$ be a quantitative synthesis problem and let ctr be a solution to $\mathscr{Q}^{qual}$. The quality of ctr is defined as*

$$Quality_{\mathscr{Q}}(\mathsf{ctr}) \stackrel{def}{=} P(\mathsf{ctr} \geq_{\mathcal{T}} X^M_{\mathscr{Q}})$$

where $X^M_{\mathscr{Q}}$ is a variable randomly sampled from the uniform distribution for Solutions$(\mathscr{Q}^{qual})$.

In particular, when $\text{Quality}_{\mathscr{Q}}(c) = 1$, c is better than or equal to all other possible solutions, i.e., is the optimal solution under the teacher's preference. Note that computing

Table 2.1. A Comparative Synthesis run for Example 2.3.5

Iter	Candidate Allocation	Query	Response	Running Best	Quality
1	$P_0(\mathsf{thrpt} = 205.2, \mathsf{ltncy} = 10.3)$	$\textsc{Validate}(P_0)$	$\top$	P_0	32.8%
2	$P_1(\mathsf{thrpt} = 470.2, \mathsf{ltncy} = 33.0)$	$\textsc{Validate}(P_1)$	$\top$	P_1	73.6%
3	$P_2(\mathsf{thrpt} = 385.2, \mathsf{ltncy} = 24.5)$	$\textsc{Validate}(P_2)$	$\top$	P_2	92.8%
...	...	...	...	...	...
6	$P_6(\mathsf{thrpt} = 405.4, \mathsf{ltncy} = 26.5)$ $P_7(\mathsf{thrpt} = 377.8, \mathsf{ltncy} = 23.8)$	$\textsc{Compare}(P_6, P_7)$	P_6	P_2	92.8%
7	$P_8(\mathsf{thrpt} = 392.9, \mathsf{ltncy} = 25.3)$	$\textsc{Validate}(P_8)$	$\top$	P_8	97.8%

the exact quality can be very expensive, if not impossible. However we can *estimate* the quality by sampling, as we do in evaluation (see §2.5.3).

Example 2.3.5. The quantitative synthesis problem $\mathscr{Q}_{MCF}$ in Example 2.3.4 involves a metric ranking $\lesssim_{real}$. Let $\mathcal{T}_{real}$ be a perfect teacher w.r.t. $\lesssim_{real}$. Table 2.1 illustrates how a voting-guided learning algorithm (which we present later in §2.4) serves as the learner and learns a near-optimal solution to $\mathscr{Q}_{MCF}$ through queries to $\lesssim_{real}$. In the first iteration, the learner solves the synthesis problem in Fig 2.2 and gets a first mediocre allocation P_0 and presents it to the architect, using query $\textsc{Validate}(P_0)$. The teacher accepts the proposal as this is the first running best. In the sixth iteration, the learner presents two programs P_6 and P_7 to the teacher and asks her to compare them. Based on the feedback that the architect prefers P_6 to P_7, the learner proposes P_8 which is confirmed by the teacher to be the best program so far. After seven queries, the running best is already very close to the optimal under the real objective (Quality of this solution has already achieved 97.8%). If the teacher wishes to answer more queries, the solution quality can be further improved.

2.3.3 The Comparative Synthesis Problem

Given the approximation nature of query-based interaction and the quality of solution defined above, the learner is tasked to solve what we call *comparative synthesis problem*, which is formally defined below.

Definition 2.3.10 (Comparative synthesis problem)**.** *A comparative synthesis problem is represented as a tuple* $\mathscr{C} = (\mathcal{P}, C, \Phi, M, \mathcal{T})$ *where* $\mathcal{P}$ *is a parameterized program,* C *is the*

space of parameters for $\mathcal{P}$, Φ is a verification condition for $\mathcal{P}$, M is a metric group w.r.t. $\mathcal{P}$ and $\mathcal{T}$ is a perfect teacher, such that $(\mathcal{P}, C, \Phi, M, \lesssim_{\mathcal{T}})$ forms a quantitative synthesis problem, which is denoted as $\mathscr{Q}$. The synthesis problem is to find, by making a sequence of COMPARE *and* VALIDATE *queries to the teacher $\mathcal{T}$, a near-optimal solution ctr to $\mathscr{Q}$ with a provable guarantee on $Quality_{\mathscr{Q}}(\mathsf{ctr})$.*

2.4 Voting-Guided Learning Algorithm

In this section, we focus on the learner side of the framework and propose a voting-guided learning algorithm that can play the role of the comparative learner and solve the comparative synthesis problem. Below we propose a novel search space combining the program search and objective learning, then present an estimation of query informativeness, based on which our voting-guided algorithm is designed. We discuss the convergence of the algorithm at last.

2.4.1 A Unified Search Space

A syntactical and natural means to describing quantitative specification is *target functions* (in contrast to the semantically defined metric ranking in Def 2.3.5). Now to solve a comparative synthesis problem efficiently, an explicit task of the learner is *program search*: the goal is to minimize human interaction (i.e., the number of queries) and maximize the quality of the solution (see Def 2.3.9) proposed through VALIDATE queries. Another implicit task of the learner is *objective learning*: to steer program search faster to the optimal and minimize the query count, the learner should conjecture target functions that fit the teacher-provided preferences, and use them to determine which programs are more likely to be optimal. Note that the conjectured target function need not (and sometimes cannot) be perfect — the goal is just to *approximates* the teacher's metric ranking $\lesssim_{\mathcal{T}}$.

Our key insight is that the two tasks are inherently tangled and better be done together. On one hand, the quantitative synthesis task needs to be guided by an appropriate objective; otherwise the search is blind and unlikely to steer to those candidates satisfying the user. On the other hand, learning a perfect target function can be extremely expensive (if not

impossible — see the "why metric ranking" discussion in §2.3) and unnecessary — even an inaccurate target function may guide the program search. We first define the target function space.

Definition 2.4.1 (Target function space)**.** *A target function space $\mathcal{O}$ is a set of target functions with respect to a d-dimensional metric group M such that for any metric ranking $\lesssim_M \subset \mathbb{R}^d \times \mathbb{R}^d$ and any finite subset $S \subset_{fin} \mathbb{R}^d$, there exists a target function $O \in \mathcal{O}$ such that for any $u, v \in S$, $u \lesssim_M v$ if and only if $O(u) \leq O(v)$.*

Example 2.4.1. The class of *conditional linear integer arithmetic* (CLIA) functions forms a target function space. A CLIA target function, intuitively, uses linear conditions over metrics to divide the domain into multiple regions, and defines in each region as a linear combination of metric values. Formally, for any d-dimensional metric group M, a target function space $\mathcal{O}^d_{CLIA}$ can be defined as the class of expressions derived from the nonterminal T of the following grammar:

$$
\begin{aligned}
T &::= E \mid \mathsf{if}\ B\ \mathsf{then}\ T\ \mathsf{else}\ T \\
B &::= E \geq 0 \mid B \wedge B \mid B \vee B \\
E &::= v_1 \mid \ldots \mid v_d \mid c \mid E + E \mid E - E
\end{aligned}
$$

where $c \in \mathbb{Z}$ is a constant integer, and v_{i} is the i-th value of the metric vector. It is not hard to see that $\mathcal{O}^d_{CLIA}$ is indeed a target function space, because with arbitrarily many conditionals, one function can be constructed to fit *any finite subset* of any metric ranking.

Example 2.4.2. While the CLIA space is general enough for arbitrary metric group, it can be too large to be efficiently searched. For many concrete metric groups, more target function spaces usually exist. For our running example, a commonly used function to quantify this trade-off is the multi-commodity flow functions used in software-driven WAN [16]. The O_{real} function (Equation 2.2.1) in our running example is an instance of the generalized, two-segment MCF function space, which can be described in the following form:

$$O(\mathsf{throughput}, \mathsf{latency}) \stackrel{\text{def}}{=} \mathsf{throughput} * ?? - \max(\mathsf{throughput} - ??, 0) * ?? - \mathsf{latency} * ?? - \max(\mathsf{latency} - ??, 0) * ??$$

where ?? can be arbitrary weights or thresholds. Note that the two-segment template is insufficient to characterize arbitrary finite metric ranking. In that case, the template may be extended to more segments. We call the whole target function space $\mathcal{O}_{MCF}$.

Now as the learner's task is to search two spaces — one for programs and one for target functions — we merge the two tasks into a single one, searching over a unified search space which we call *Pareto candidate set*:

Definition 2.4.2 (Pareto candidate set)**.** *Let $\mathscr{C} = (\mathcal{P}, C, \Phi, M, \mathcal{T})$ be a comparative synthesis problem and $\mathcal{O}$ be a target function space w.r.t. M. A* Pareto candidate set *(PCS) with respect to $\mathscr{C}$ and $\mathcal{O}$ is a finite partial mapping $\mathscr{G} : \mathcal{O} \twoheadrightarrow C$ from a space of target functions $\mathcal{O}$ to a space of program parameters C, such that for any $O \in \mathcal{O}$, $\mathscr{G}(O)$ is the effectively optimal solution under target function O, i.e., a solution $c \in C$ such that $\mathscr{G}(O) \lesssim_O c$, if exists, cannot be effectively found. Specifically, for any other $O' \in \mathcal{O}$, $\mathscr{G}(O') \lesssim_O \mathscr{G}(O)$.*

Intuitively, a Pareto candidate set (PCS) $\mathscr{G}$ maintains a set of candidate target functions, a set of candidate programs, and a mapping between the two sets, and guarantees that every candidate target function O is mapped to the best candidate program under O.[4]

2.4.2 Query Informativeness

Now with the unified search space — PCS in Def 2.4.2 — comparative synthesis becomes a game between the learner and the teacher: a PCS $\mathscr{G}$ is maintained as the current search space; and in each iteration, the learner makes a query and the teacher gives her response, based on which $\mathscr{G}$ is shrunken. The learner's goal should be, in each iteration, to pick the most informative query in the sense that it can reduce the size of $\mathscr{G}$ as fast as possible. The key question is how to evaluate the informativeness of a query.

[4]↑Note that in general, the best candidate program under O is not necessarily unique. To break the tie and make $\mathscr{G}$ uniquely determined by the component sets of target functions and programs, when two candidate programs c_1 and c_2 both get the highest reward under O, we assume $\mathscr{G}(O)$ is c_1 if $M(c_1)$ is smaller than $M(c_2)$ in lexicographical order, or c_2 otherwise.

$$
\begin{aligned}
\text{Quality}(\textsc{Compare}(c_1, c_2)) &\stackrel{\text{def}}{=} \min\Big(\big|c_1 \lesssim_O c_2\big|,\ \big|c_1 \gtrsim_O c_2\big|,\ \big|c_1 \nsim_O c_2\big|\Big) \\
\text{Quality}(\textsc{Validate}(c)) &\stackrel{\text{def}}{=} \min\Big(\big|\mathit{NewBest}(c)\big|,\ \big|c >_O r\_best\big|\Big) \\
|\varphi| &\stackrel{\text{def}}{=} \Big|\{x \mid \forall O \in \text{dom}(\mathscr{G}) : \mathscr{G}(O) = x \ \Rightarrow\ \varphi\}\Big| \\
\big|\mathit{NewBest}(c)\big| &\stackrel{\text{def}}{=} r\_best <_O c \ \vee\ \forall y \in \text{image}(\mathscr{G}).y \lesssim_O c
\end{aligned}
$$

Figure 2.4. Informativeness of queries (with $\mathscr{G}$ the current PCS, and r_best the running best program).

In this paper, we develop a greedy strategy which evaluates the informativeness by computing *how many candidate programs from $\mathscr{G}$ can be removed immediately with the teacher's response*. As the teacher's response can be arbitrary, our evaluation considers all possible responses and take the minimum number among all cases. The formulation is shown in Fig 2.4 and explained below.

• Compare **query:** For Compare(c_1, c_2), recall the teacher may prefer $\mathcal{P}[c_1]$ to $\mathcal{P}[c_2]$, or vice versa, or consider the two programs equally good (corresponding to the three responses: $<$, $>$ and $=$). In each case, we can remove all the candidate target functions that have a different relative ordering of $\mathcal{P}[c_1]$ and $\mathcal{P}[c_2]$ than the teacher's preference. We denote the number of candidates that can be removed when the teacher prefers $\mathcal{P}[c_1]$ (resp. $\mathcal{P}[c_2]$) as $\big|c_1 \lesssim_O c_2\big|$ (resp. $\big|c_1 \gtrsim_O c_2\big|$). Further let $\big|c_2 \nsim_O c_1\big|$ denote the candidates that can be removed if the user indicates both programs are equally good. The overall informativeness is just the minimum of the three cases.

• Validate **query:** For Validate(c), recall the teacher may confirm that $\mathcal{P}[c]$ is indeed better than the running best $\mathcal{P}[r\_best]$ (response $\top$), or keep the current running best (response $\bot$). Like the compare query, in each case, we can eliminate all candidate target functions that do not satisfy this relative preference between $\mathcal{P}[c]$ and $\mathcal{P}[r\_best]$ expressed by the user. However, in the case that the user prefers $\mathcal{P}[c]$, we can additionally remove all candidate target functions for which $\mathcal{P}[c]$ is already the best choice, i.e., more queries are not needed for further improvement — we use $\big|\mathit{NewBest}(c)\big|$ in Fig 2.4 to denote the total

Table 2.2. Example PCS $\mathscr{G}_{ex}$.

Trgt func O	Optimal solution $\mathscr{G}_{ex}(O)$	Ranking of all candidate programs under O
O_0	$\boldsymbol{P_2}$	$P_0 < r\_best < P_1 < \boldsymbol{P_2}$
O_1	$\boldsymbol{P_2}$	$P_1 < P_0 < r\_best < \boldsymbol{P_2}$
O_2	$\boldsymbol{P_0}$	$P_1 = P_2 < r\_best < \boldsymbol{P_0}$
O_3	$\boldsymbol{P_1}$	$r\_best < P_0 < P_2 < \boldsymbol{P_1}$
O_4	$\boldsymbol{P_2}$	$r\_best < P_0 < P_1 < \boldsymbol{P_2}$

Table 2.3. Informativeness of queries $\textsc{Compare}(c_1, c_2)$.

c_1	c_2	$c_1 \lesssim_O c_2$	$c_1 \gtrsim_O c_2$	$c_1 \nsim_O c_2$	Quality
P_0	P_1	1	1	3	1
P_1	P_2	2	2	2	2
P_0	P_2	2	1	3	1
P_0	r_best	0	2	3	0
P_1	r_best	1	1	3	1
P_2	r_best	1	2	3	1

Table 2.4. Informativeness of queries $\textsc{Validate}(c)$.

c	$NewBest(c)$	$c >_O r\_best$	Quality
P_0	1	2	1
P_1	2	1	1
P_2	2	2	2

number of eliminated candidate programs in this case. The overall informativeness is just the minimum of the two cases.

Example 2.4.3. Table 2.2 shows a PCS $\mathscr{G}_{ex}$ that consists of 5 candidate target functions, namely O_{i} for $0 \leq \mathrm{i} \leq 4$, and 3 candidate programs, namely P_0, P_1 and P_2. The rankings of all candidate programs and the current running best r_best under target functions are also shown in Table 2.2. The informativeness of Compare and Validate queries for PCS $\mathscr{G}_{ex}$ is presented in Tables 2.3 and 2.4, respectively. Take $\textsc{Compare}(P_1, P_2)$ as an example, $\left|P_1 \lesssim_O P_2\right|$ is 2 since all candidate target function except O_3 believes $P_1 \lesssim_O P_2$ and removing these four target functions essentially removes P_0 and P_2 from PCS $\mathscr{G}_{ex}$. As per Fig 2.4, $\left|P_1 \gtrsim_O P_2\right|$ and $\left|P_1 \nsim_O P_2\right|$ are also 2. The informativeness of $\textsc{Compare}(P_1, P_2)$ is therefore 2, the minimum of the three cases above, which means at least 2 candidate programs will

be removed from $\mathscr{G}_{ex}$ no matter which the user prefers. Consider VALIDATE(P_2) as another example, $\left|NewBest(P_2)\right|$ is 2, since O_2 prefers r_best over P_2 and P_2 is the best choice under O_0, O_1 and O_4. Candidate program P_1 and P_2 will be removed from $\mathscr{G}_{ex}$ as their target functions either do not satisfy user preference or can not improve the running best further. Given PCS $\mathscr{G}_{ex}$, both VALIDATE(P_2) and COMPARE(P_1, P_2) share highest informativeness, which is 2. In this case, VALIDATE(P_2) will be presented to the user.

2.4.3 The Algorithm

Our learning algorithm is almost straightforward: for each iteration, compute the informativeness of every possible query and make the most informative query. The remaining issue is that it is not realistic to keep a PCS that contains all possible candidates, because the number of candidates is usually very large, if not infinite. For example, $\mathscr{L}_{MCF}$ in Example 2.3.1 has infinitely many Pareto optimal solutions, ranging in the continuous spectrum from maximizing throughput to minimizing latency. To this end, our voting-guided algorithm maintains a moderate-sized PCS, from which queries are generated and selected based on their informativeness.

Algorithm 1 illustrates the voting-guided algorithm. The algorithm takes as input a comparative synthesis problem $\mathscr{C}$ and a target function space $\mathcal{O}$, and maintains a PCS $\mathscr{G}$ w.r.t. $\mathscr{C}$ and $\mathcal{O}$ and set of preferences R, both empty initially. In each iteration, the algorithm computes the informativeness of all possible queries that can be made about the current candidates image($\mathscr{G}$), and picks the highest-informativeness query according to the computation presented in Fig 2.4 (line 7). After the query is made and the response is received, an **compare-update** subroutine is invoked to update $\mathscr{G}$ and remove all candidates violating the preference (lines 11–15). Moreover, the algorithm also checks at the beginning of every iteration the size of $\mathscr{G}$; if image($\mathscr{G}$) is below a fixed threshold THRESH, the algorithm attempts to extend $\mathscr{G}$ using a **generate-more** subroutine. The algorithm terminates and returns the current running best when $\mathscr{G}$ becomes 0 or NQUERY queries have been made, where NQUERY is the number of queries that the teacher promises to answer (line 17). Table 2.1 shows an example run of this algorithm.

input : A comparative synthesis problem $\mathscr{C} = (\mathcal{P}, C, \Phi, M, \mathcal{T})$ and a target function space $\mathcal{O}$ with respect to M
output: A quasi-optimal solution to $\mathscr{C}$

def voting-guided-learn$(\mathcal{P}, C, \Phi, M, \mathcal{T}, \mathcal{O})$**:**
  $R \leftarrow \top$, $count \leftarrow 0$, $\mathscr{G} \leftarrow \emptyset$ `// collected preferences, query count and the PCS`
  $r\_best \leftarrow$ SynProg$(\mathcal{P}, C, \phi_P)$ `// get the first solution and initialize the running best`
  repeat
    if $|image(\mathscr{G})| <$ Thresh **:**
      generate-more $(\mathcal{P}, M, R, r\_best, \mathscr{G})$ `// generate more candidates`
    `/* pick and make the most informative query */`
    $q\_type, c_1, c_2 \leftarrow$ BestQuery $(\mathscr{G})$
    $response \leftarrow$ MakeQuery (q_type, c_1, c_2)
    if $q\_type =$ *Validate* `// if` Validate(c_1) **:**
      update $(M(\mathcal{P}[c_1]), M(\mathcal{P}[r\_best]), response)$ `// update` R `and` $\mathscr{G}$
      if $response = (\top)$ **:**
        `// if` $\mathcal{P}[c_1]$ `is better than running best`
        $\mathscr{G} \leftarrow \mathscr{G}\,|_{\{O \mid \mathscr{G}(O) >_O \mathcal{P}[c_1]\}}$
        $r\_best \leftarrow c_1$
    elif $q\_type =$ *Compare* `// if` Compare(c_1, c_2) **:**
      update $(M(\mathcal{P}[c_1]), M(\mathcal{P}[c_2]), response)$ `// update` R `and` $\mathscr{G}$
    $count++$
  until $\mathscr{G} = \emptyset \vee count \geq$ NQuery ;
  return r_best

Algorithm 1: The voting-guided learning algorithm.

The subroutines involved in the algorithm are shown as Algorithm 2. The **compare-update** subroutine is straightforward, taking a new preference pair and shrinking $\mathscr{G}$ accordingly. The **generate-more** subroutine is tasked to expand $\mathscr{G}$ as much as possible within a time limit. Each time, it tries to find a pair (O, c) such that O satisfies all existing preferences and prefers $\mathcal{P}[c]$ to $\mathcal{P}[r\_best]$, and $\mathcal{P}[c]$ is effectively optimal under O. Note that this subroutine delegates several heavy-lifting tasks to off-the-shelf, domain-specific procedures: SynProg for qualitative synthesis, SynObj for objective synthesis, and Improve for optimization under a known objective. For example, the qualitative synthesis and objective synthesis problem of our running example can be encoded to a logical query and discharged by any SMT solvers, such as Z3 [41]. The optimization problem under known objectives can also be solved by a linear programming solver, such as Gurobi [42].

input : Two program metric vectors m, n and their comparison result *response*
modifies: The current metric vector preferences R and the current PCS $\mathscr{G}$

def update *(m, n, response)*:
  if $response = (>)$:
    $R \leftarrow R \wedge m > n$
  elif $response = (<)$:
    $R \leftarrow R \wedge n > m$
  else:
    $R \leftarrow R \wedge m = n$
  $\mathscr{G} \leftarrow \mathscr{G}\,|_{\{O|O \models R\}}$
  return

input : A parameterized program $\mathcal{P}$, a metric group M, current metric vector preferences R and current running best *r_best*
modifies: The Pareto candidate set $\mathscr{G}$

def generate-more *(P, M, R, r_best, G)*:
  repeat
    $c \leftarrow$ SynProg$(\mathcal{P}, M)$; `// synthesize an arbitrary (Pareto optimal) program`
    $O \leftarrow$ SynObj$(R \wedge M(\mathcal{P}[c]) > M(\mathcal{P}[r_best]))$ `// synthesize an objective that prefers the new` c `over` *r_best*
    if $O \neq \bot$:
      $c \leftarrow$ Improve$(O, \mathcal{P}, M, c)$ `// this is optional: try to improve` $\mathcal{P}[c]$
    else:
      $O \leftarrow$ SynObj(R) `// synthesize an arbitrary objective satisfying` R
      $c \leftarrow$ Improve$(O, \mathcal{P}, M, r_best)$ `// synthesize a best possible program under` O`, but at least better than` *r_best*
    $\mathscr{G} \leftarrow \mathscr{G} \uplus (O, c)$
  until *timeout*;

Algorithm 2: The subroutines involved in the voting-guided learning algorithm.

2.4.4 Convergence

In the rest of the section, we discuss the convergence of the algorithm. Recall that our algorithm only produces quasi-optimal programs as the ground-truth target function is not present. Therefore, the algorithm should be evaluated on the rate of convergence [43], i.e.,

how fast the median quality [5] of solutions (see Def 2.3.9) approaches 1 as more queries are made. Our first result is that the algorithm guarantees a logarithmic rate of convergence.

Theorem 2.4.1. *Given a comparative synthesis problem $\mathscr{C}$ and a target function space $\mathcal{O}$ as input, if Algorithm 1 terminates after n queries, the median quality of the output solutions is at least $2^{\frac{-1}{n+1}}$.*

Proof. Note that every query will discard at least one candidate program from the PCS, regardless of the query type. In other words, the final output c must be the optimal among at least $(n+1)$ randomly selected candidates from the uniform distribution. Therefore, the quality of c is at least the $(n+1)$-th order statistic of the uniform distribution, which is a beta distribution $\text{Beta}(n+1, 1)$, whose median is $2^{\frac{-1}{n+1}}$. □

The proved lower-bound in the theorem above is tight only when each query only removes one candidate from the PCS $\mathscr{G}$. Unfortunately, the following lemma shows that in general, this scenario is always realizable:

Theorem 2.4.2. *The bound in Theorem 2.4.1 is tight.*

The PCS constructed for the following lemma serves as a witness of the bound tightness:

Lemma 2.4.1. Let $\mathscr{S} = (\mathcal{P}, C, \Phi)$ be a qualitative synthesis problem with infinitely many solutions and $\mathcal{O}$ be a target function space. For any integer $n > 0$, there exist a PCS $\mathscr{G} : \mathcal{O} \nrightarrow C$ and a parameter $r\_best \in C$ such that: **(1)** $|\text{image}(\mathscr{G})| = n$; **(2)** for any $c_1, c_2 \in \text{image}(\mathscr{G})$, $\text{Quality}(\textsc{Compare}(c_1, c_2)) = 1$; **(3)** for any $c \in \text{image}(\mathscr{G})$, $\text{Quality}(\textsc{Validate}(c)) = 1$.

Proof. As $\mathscr{C}$ has infinitely many solution, we can pick arbitrary n solutions, say $c_1, \ldots, c_n$. For each $1 \leq \mathrm{i} \leq n$, one can construct a total order $\lesssim_{\mathrm{i}}$ such that $c_n \lesssim_{\mathrm{i}} \ldots c_{\mathrm{i}+1} \lesssim_{\mathrm{i}} c_{\mathrm{i}-1} \ldots c_1 \lesssim_{\mathrm{i}} c_{\mathrm{i}}$. According to the definition of target function space (Def 2.4.1), there exists a target function O_{i} that fits $\lesssim_{\mathrm{i}}$. Now we can construct $\mathscr{G}$ such that $\text{dom}(\mathscr{G}) = \{O_1, \ldots, O_n\}$, and $\mathscr{G}(O_{\mathrm{i}}) = c_{\mathrm{i}}$ for each i. It can be verified $\mathscr{G}$ is a Pareto candidate set satisfying the required conditions. □

[5] ↑The algorithm involves random sampling and results we prove below are for the median quality of output solutions; the proofs can be easily adapted to get similar results for the mean quality of solutions.

2.4.5 Better Convergence Rate with Sortability

We have shown that our voting-guided algorithm guarantees a logarithmic rate of convergence in general, but are there scenarios for which the algorithm guarantees faster convergence? We next show that when the comparative synthesis problem is convex and the target function space is concave with two metrics, our algorithm guarantees a faster, *linear* convergence. The conditions are commonly seen in practice — satisfied by half of optimization scenarios studied in §2.5 — and intuitively, capture the assumption that there are two competing metrics (e.g., throughput and latency) such that for each metric continued improvement leads to diminishing marginal utility (e.g., increasing throughput from 1Gbps to 2Gbps is more favorable than increasing throughput from 2Gbps to 3Gbps).

The idea of the proof bears a similarity to the convergence guarantee for many algorithms in traditional convex optimization [44]; but the key difference is that the objective is indeterminate for our algorithm. We first introduce a key enabling notion for the proof called *sortability*, which makes sure that the candidates in the PCS can be ordered appropriately such that every target function with corresponding candidate c always prefers its nearer neighbors to farther neighbors.

Definition 2.4.3 (Sortability)**.** *A PCS $\mathscr{G}$ is sortable if there exists a total order $\prec$ over image($\mathscr{G}$) such that for any target functions $O, P, Q \in dom(\mathscr{G})$ such that $\mathscr{G}(O) \prec \mathscr{G}(P) \prec \mathscr{G}(Q)$, the following two conditions hold: $\mathscr{G}(P) >_O \mathscr{G}(Q)$, and $\mathscr{G}(P) >_Q \mathscr{G}(O)$ A target function space $\mathcal{O}$ is sortable with respect to a comparative synthesis problem $\mathscr{C}$ if any PCS $\mathscr{G}$ w.r.t. $\mathscr{C}$ and $\mathcal{O}$ is sortable.*

The following lemma shows that if a PCS is sortable, one can make a query to cut at least half of the candidates, no matter what the teacher's response is.

Lemma 2.4.2. If a Pareto candidate set $\mathscr{G}$ is finite and sortable, then there exists a query whose quality for $\mathscr{G}$ as computed in Fig 2.4 is $\lfloor \frac{|\text{image}(\mathscr{G})|}{2} \rfloor$.

Proof. Let $n = |\text{image}(\mathscr{G})|$ and $m = \lfloor \frac{|\text{image}(\mathscr{G})|}{2} \rfloor$. As $\mathscr{G}$ is sortable, by Def 2.4.3, there exists a total order $\mathscr{G}(O_1) \prec \cdots \prec \mathscr{G}(O_n)$. Now we claim that $\text{Quality}\Big(\textsc{Compare}\big(\mathscr{G}(O_m), \mathscr{G}(O_{m+1})\big)\Big) =$

m. By Def 2.4.3, for any $1 \le \mathrm{i} \le m$, $\mathscr{G}(O_m) >_{O_\mathrm{i}} \mathscr{G}(O_{m+1})$, and for any $m+1 \le \mathrm{j} \le n$, $\mathscr{G}(O_m) <_{O_\mathrm{j}} \mathscr{G}(O_{m+1})$. Then according to the query quality estimation described in Fig 2.4, both $\#\mathrm{RemNEQ}\big(\mathscr{G}(O_m), \mathscr{G}(O_{m+1})\big)$ and $\#\mathrm{RemNEQ}\big(\mathscr{G}(O_{m+1}), \mathscr{G}(O_m)\big)$ are at least m. Therefore, $\mathrm{Quality}\big(\textsc{Compare}(\mathscr{G}(O_m), \mathscr{G}(O_{m+1}))\big) = m$, which is $\lfloor \frac{|\mathrm{image}(\mathscr{G})|}{2} \rfloor$. □

With the lower bound of removed candidates guaranteed by Lemma 2.4.2, our voting-guided synthesis algorithm guarantees to produce a unique best candidate after a logarithmic amount of queries:

Theorem 2.4.3. *Given a comparative synthesis problem $\mathscr{C}$ with metric group M and a sortable target function space $\mathcal{O}$ w.r.t. M as input, if Algorithm 1 terminates after n queries, the median quality of the output solutions is at least $\big(1 - \frac{1}{\Omega(1.5^n)}\big)$.*

Proof. Note that Algorithm 1 generates candidates for the Pareto candidate set $\mathscr{G}$ (through the **generate-more** subroutine) through random sampling. Therefore, if a query cuts the size of current candidate pool ($\mathscr{G}$ and the running best) by a ratio of r, the search space (those candidates satisfying all preferences in R) is cut by an equal or higher ratio in that iteration (extra candidates may be discarded by **generate-more**, before the query). Now as $\mathscr{G}$ is sortable, by Lemma 2.4.2, after the highest-informativeness query, the number of candidates remaining in $\mathscr{G}$ is at most $\lceil \frac{|\mathrm{image}(\mathscr{G})|}{2} \rceil$. In other words, the query reduces the size of $\mathscr{G}$ by a ratio of at least $\frac{2}{3}$ (when $|\mathrm{image}(\mathscr{G})| = 2$, the total number of candidates including the running best, reduces from 3 to 2), except for the last query. Therefore, the output is the best among $\mathcal{O}(1.5^n)$ randomly selected candidates, which is $\mathrm{Beta}(1.5^n, 1)$-distributed. Hence by Def 2.3.9, the median of the quality of the output is $\frac{1}{2^{\left(\frac{1}{1.5^n}\right)}}$, which is asymptotically equivalent to $\big(1 - \frac{1}{\Omega(1.5^n)}\big)$. □

We now formally define the convexity of the comparative synthesis problem and the concavity of the target function space, and build the main convergence result by proving the sortability.

Definition 2.4.4 (Convexity of comparative synthesis problem)**.** *A comparative synthesis problem $\mathscr{C}$ with metric group M is convex if for any two solutions c_1, c_2 to $\mathscr{C}^{qual}$ and any*

$\alpha \in [0,1]$, *a solution* c_3 *to* $\mathscr{C}^{qual}$ *can be effectively found such that* $M(c_3) \gtrsim \alpha \cdot M(c_1) + (1-\alpha) \cdot M(c_2)$.

Definition 2.4.5 (Concavity of target function space)**.** *Let* $\mathcal{O}$ *be a target function space w.r.t. a d-dimensional metric group* M*.* $\mathcal{O}$ *is concave if for any* $O \in \mathcal{O}$*, for any* $v_1, v_2 \in \mathbb{R}^d$ *and any* $\alpha \in [0,1]$*,* $O(\alpha \cdot v_1 + (1-\alpha) \cdot v_2) \geq \max(O(v_1), O(v_2))$.

Example 2.4.4. Our running example falls in this subclass. The comparative synthesis problem $\mathscr{C}_{MCF}$ in Example 2.3.5 is convex. As shown in Fig 2.2, both throughput and latency are weighted sum of allocations to every link. Therefore given any two solutions c_1 and c_2, their convex combination is still feasible, and the metric vector is also the corresponding convex combination of $M(c_1)$ and $M(c_2)$. Moreover, it is not hard to verify that the target function space $\mathcal{O}_{MCF}$ in Example 2.4.2 is concave, as both the weights of throughput and latency decrease when their values are good enough and exceed a threshold.

Theorem 2.4.4. *Let* $\mathscr{C}$ *be a convex comparative synthesis problem with a 2-dimensional metric group* M *and* $\mathcal{O}$ *be a concave target function space w.r.t.* M*, then* $\mathcal{O}$ *is sortable w.r.t.* $\mathscr{C}^{qual}$.

Proof. We shall show the sortability of any Pareto candidate set $\mathscr{G}$ w.r.t. $\mathscr{C}^{\text{qual}}$ and $\mathcal{O}$. We claim that the lexicographic order $\prec_{lex}$ over $\mathbb{R}^2$ (i.e., $(a_1, a_2) \prec_{lex} (b_1, b_2)$ if and only if $a_1 < b_1$ or $a_1 = b_1 \wedge a_2 < b_2$) witnesses the sortability. Per Def 2.4.3, for any target functions $O, P, Q \in \text{dom}(\mathscr{G})$ such that $\mathscr{G}(O) \prec_{lex} \mathscr{G}(P) \prec_{lex} \mathscr{G}(Q)$, we shall show $\mathscr{G}(P) >_O \mathscr{G}(Q)$ below. It can be similarly proved that $\mathscr{G}(P) >_Q \mathscr{G}(O)$.

Let $M(\mathscr{G}(O)) = (o_1, o_2)$, $M(\mathscr{G}(P)) = (p_1, p_2)$, and $M(\mathscr{G}(Q)) = (q_1, q_2)$. Note that by Def 2.4.2, each of $\mathscr{G}(O)$, $\mathscr{G}(P)$ and $\mathscr{G}(Q)$ is optimal under a distinct target function, therefore $M(\mathscr{G}(O))$, $M(\mathscr{G}(P))$, and $M(\mathscr{G}(Q))$ are pairwise incomparable, i.e., $\{o_1, p_1, q_1\}$ and $\{o_2, p_2, q_2\}$ are all distinct values. Due to the lexicographic order $\prec_{lex}$, we have $o_1 < p_1 < q_1$ and $o_2 > p_2 > q_2$. Now by Def 2.4.4, one can effectively find a solution c such that

$$M(c) \geq \left(\frac{(q_1 - p_1) \cdot o_1 + (p_1 - o_1) \cdot q_1}{q_1 - o_1}, \frac{(q_1 - p_1) \cdot o_2 + (p_1 - o_1) \cdot q_2}{q_1 - o_1}\right) = \left(p_1, \frac{(q_1 - p_1) \cdot o_2 + (p_1 - o_1) \cdot q_2}{q_1 - o_1}\right)$$

Table 2.5. Summary of topologies.

Topology	Abilene	B4	CWIX	BTNorthAmerica	Tinet	Deltacom	Ion
#nodes	11	12	21	36	48	103	114
#links	14	19	26	76	84	151	135

Then by Def 2.4.2, $\mathscr{G}(P)$ is at least as good as c and $p_2 \geq \frac{(q_1 - p_1) \cdot o_2 + (p_1 - o_1) \cdot q_2}{q_1 - o_1}$. Finally, by Def 2.4.5, we have $O((p_1, p_2)) \geq O((p_1, \frac{(q_1 - p_1) \cdot o_2 + (p_1 - o_1) \cdot q_2}{q_1 - o_1})) \geq O((q_1, q_2))$. In other words, $\mathscr{G}(P) >_O \mathscr{G}(Q)$. □

2.5 Evaluation

We have prototyped the comparative synthesis framework and the voting-guided learning algorithm as NET10Q — an interactive system that produces near-optimal network design by asking 10 questions to the user — through which we evaluate the effectiveness and efficiency of our approach. We selected four real-world network design scenarios and conducted experiments with both oracles and human users. Our evaluations were conducted on seven real-world, large-scale internet backbone **topologies** obtained from [18], [32] (sizes summarized in Table 2.5). Note that the size of our largest topologies, namely Deltacom and Ion, are already beyond the ones typically considered in the traffic engineering community.

2.5.1 Network Optimization Problems

We summarize the four optimization scenarios in Table 2.6, including their metric groups, target function spaces and sortability. We present some details below.

Balancing throughput and latency (MCF). This is our running example based on [16] described throughout the paper. This bandwidth allocation problem focuses on a single traffic class and considers balancing the throughput and latency in the network.

Utility maximization with multiple traffic classes (BW). A well-studied optimization problem is maximizing utility when allocating bandwidth to traffic of different classes [19], [22], [45]. Many applications such as file transfer have concave utility functions which indicate that as more traffic is received, the marginal utility in obtaining a higher al-

Table 2.6. Summary of optimization scenarios.

Scenario	Metric group	Target function space	Sortable?
MCF	(throughput, -latency)	$\mathsf{throughput} * ?? - \max(\mathsf{throughput} - ??, 0) * ??$ $-\mathsf{latency} * ?? - \max(\mathsf{latency} - ??, 0) * ??$	Yes
BW	$\left(\right.$ avg_k: average allocation to the flows in the k-th class $\left.\right)$	$\sum_{1 \le k \le K} w_k \log\left(avg_k\right) \quad (w_k > 0)$	No
NF	$\left(\right.$ $zn_{\mathrm{i}}, zf_{\mathrm{i}}$: guaranteed fraction of the traffic demand of group i under normal conditions and failures respectively $\left.\right)$	$\sum_{\mathrm{i}} wn_{\mathrm{i}} * zn_{\mathrm{i}} + wf_{\mathrm{i}} * zf_{\mathrm{i}} \quad (wn_{\mathrm{i}}, wf_{\mathrm{i}} > 0)$	No
OSPF	$(-\mathsf{latency}, -\mathsf{utilization})$	$\begin{cases} \mathsf{utilization} & \mathsf{utilization} > ?? \\ \mathsf{latency} * ?? + \mathsf{utilization} * ?? & ?? < \mathsf{utilization} < ?? \\ \mathsf{latency} & otherwise \end{cases}$	Yes

location is smaller. A common concave utility function which is widely used is a logarithmic utility function, where a flow that receives a bandwidth allocation of x gets a utility of $\log x$. Consider N flows, and K classes. Each flow belongs to one of the classes with F^k denoting the set of flows belonging to class k. The weight of class k is denoted by w_k and is a knob manually tuned today to control the priority of the class, which we treat as an unknown in our framework.

Performance with and without failures (NF). Resilient routing mechanisms guarantee the network does not experience congestion on failures [17], [20], [21], [30], [31] by solving optimization problems that conservatively allocate bandwidth to flows while planning for a desired set of failures. We consider the model used in [17] to determine how to allocating bandwidth to flows while being robust to single link failure scenarios. We consider an objective with (unknown) knobs $w_{n\mathrm{i}}$ and $w_{f\mathrm{i}}$ that trade off performance under normal conditions and failures tuned differently for each group of flows i.

Balancing latency and link utilization (OSPF). Open Shortest Path First (OSPF) is a widely used link-state routing protocol for intra-domain internet and the traffic flows are routed on shortest paths [15]. A variant of OSPF routing protocol assigns a weight to each link in the network topology and traffic is sent on paths with the shortest weight and equally split if multiple shortest paths with same weight exist. By configuring the link weights, network architect can tune the traffic routes to meet network demands and optimize the network on different metrics [15]. We consider a version of the OSPF problem where link

weights must be tuned to ensure link utilizations are small while still ensuring low latency paths [46]. Intuitively, when utilization is higher than a threshold, it becomes the primary metric to optimize, and when lower than a threshold, minimizing latency is the primary goal. In between the thresholds, both latency and utilization are important, and can be scaled in a manner chosen by the network architect. We treat the thresholds and the scale factors as unknowns in the objective.

2.5.2 Implementation

Note that in NET10Q, once the scenario and the topology are fixed, we can pre-compute a large pool of objective-program pairs, from which the PCS is generated. For each scenario-topology combination, we used the templates shown in Table 2.6 to generate a pool of random target functions. Then for all scenarios except for **OSPF**, we generate their corresponding optimal allocations using Gurobi [42], a state-of-the-art solver for linear and mixed-integer programming problems. For **OSPF**, as we are not aware of any existing tools that can symbolically solve the optimization problem, we used traditional synthesis approaches (cf. Fig 2.2) to generate numerous feasible link weight assignments. The pre-computed target functions and allocations are paired to form a large PCS serving as the candidate pool.

When the teacher is an imperfect oracle or a human user, inconsistent answers may potentially result in the algorithm unable to determine objectives that meet all user preferences. To ensure NET10Q robust to an imperfect oracle, inspired by the ensemble methods [47], we implemented NET10Q as a multi-threaded application where a primary thread accepts all inputs and the backup threads run the same algorithm but randomly discard some user inputs. In case no objective could satisfy all user preferences, a backup thread with the largest satisfiable subset of user inputs would take over.

2.5.3 Oracle-Based Evaluation

We used NET10Q to solve all scenario-topology combinations described above, through interaction with (both perfect and imperfect) oracles who answer queries based on their internal objectives. As a first-of-its-kind system, NET10Q does not have any similar systems

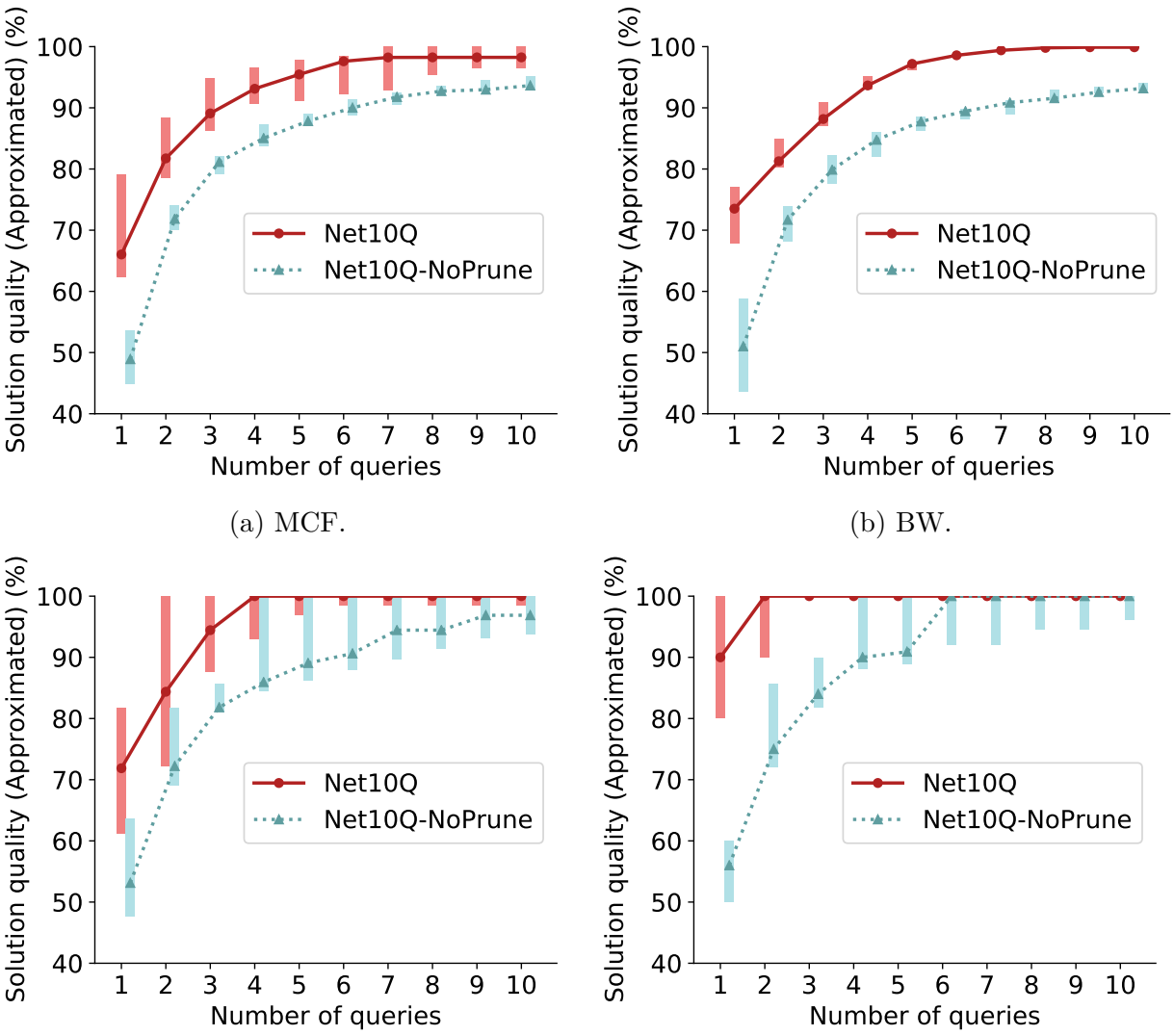
to compare with. Therefore, we developed a variant of NET10Q which adopts a simple but aggressive strategy: repeatedly proposing optimal candidates generated from randomly picked target functions. We call this baseline algorithm NET10Q-NoPrune, as the teacher's preference is not used to prune extra candidates from the search space. As a solution's real quality (per Def 2.3.9) is not practically computable, we approximate its quality using its rank in our pre-computed candidate pool.[6] Moreover, as NET10Q involves random sampling, we ran each synthesis task 301 times and reported the median of the (approximated) solution quality achieved after every query.

Evaluation on perfect teacher

We built an oracle to play the role of a perfect teacher who answers all queries correctly based on a ground truth objective. For each scenario, we as experts manually crafted a target function that fits the template and reflects practical domain knowledge.[7]

We presents the performance of NET10Q and NET10Q-NoPrune on solving four network optimization problems (cf. Table 2.6) on seven different topologies (cf. Table 2.5). Our key observation is that NET10Q *performed constantly better than* NET10Q-NoPrune *in every scenario-topology combination.* In the interest of space, we collected the quality of solutions achieved over all seven topologies for each optimization scenario and presented the median (shown as dots) and the range from max to min (shown as bars). As Fig 2.5 shows, our voting-guided algorithm is very effective. NET10Q always only needs 5 or fewer queries to obtain a solution quality achieved by NET10Q-NoPrune in 10 queries. We note that although the all-topology range for NET10Q sometimes overlaps with the corresponding range for NET10Q-NoPrune (primarily for the **NF** scenario), NET10Q still outperformed NET10Q-NoPrune for every topology. We leave the topology-wise results for **NF** in Appendix 2.8.1. Further, in all the cases where we could compute the optimal under the ground truth objective, we confirmed that programs recommended by NET10Q achieved at least 99% of the optimal.

[6]↑The quality is computed among Pareto optimal solutions only. In other words, the solution quality as per Definition 2.3.9 should be higher than what we report here.
[7]↑The ground truth does not have to match the template; see §2.5.4 for human teacher who is oblivious to the template.



(a) MCF.

(b) BW.

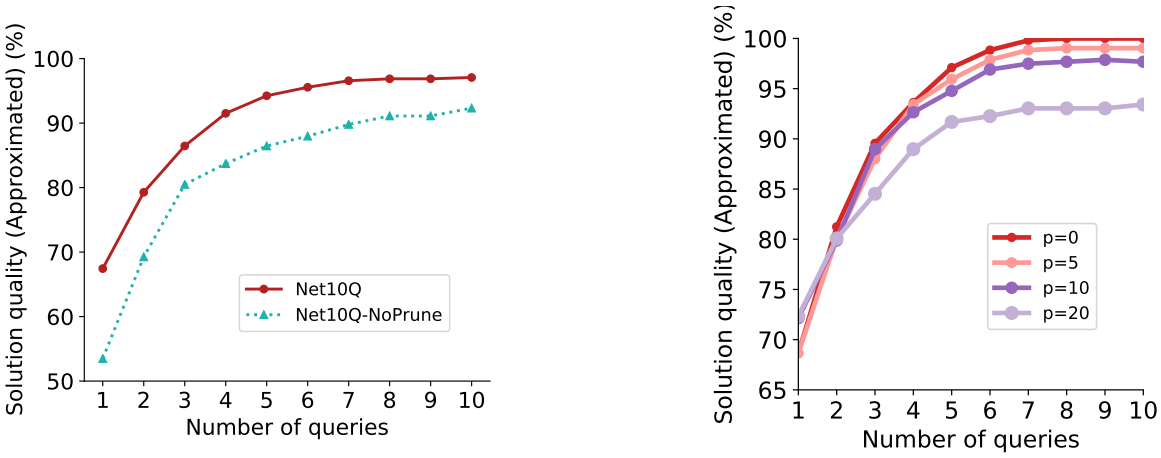
(c) NF.

(d) OSPF.

Figure 2.5. Comparing NET10Q and NET10Q-NoPrune with perfect oracle (across all seven topologies). Curves to the left are better. (More detailed, per-topology results for **NF** is available in Appendix 2.8.1)

Evaluation on imperfect teacher

We also adapt the oracle to simulate imperfect teachers whose responses are potentially inconsistent, based on an error model described below. When an allocation candidate is presented, the imperfect oracle assigns a random reward that is sampled from a normal distribution, whose expectation is the true reward under corresponding ground truth objective.



(a) NET10Q vs. NET10Q-NoPrune ($p = 10$).

(b) Sensitivity to p.

Figure 2.6. Performance of NET10Q with imperfect oracle (**BW** on CWIX) of different levels of inconsistency evaluated in different case studies on CWIX. Curves to the left are better.

The standard deviation is p percentage of the distance between average reward and optimal reward under ground truth objective.

Fig 2.6 shows our experimental results with imperfect teacher on **BW** with the CWIX topology. In the interest of space, we defer results of other scenarios to Appendix 2.8.2, from which we see similar trend. Fig 2.6a compares NET10Q with NET10Q-NoPrune under the inconsistency level $p = 10$. NET10Q continues to outperform NET10Q-NoPrune. Fig 2.6b presents the sensitivity of NET10Q on the inconsistency level ($p = 0, 5, 10, 20$). Although the solution quality degrades with higher inconsistency p, NET10Q achieves relatively high solution quality even when p is as high as 20. The results show that NET10Q tends to be able to handle moderate feedback inconsistency from an imperfect teacher, although investigating ways to achieve even higher robustness is an interesting area for future work.

Runtime and Scaling

We first discuss the online query time experienced by users. For every synthesis task mentioned above, and across all topologies, the average running time spent by NET10Q for each interactive user query is less than 0.15 seconds. The approach scales well with topology size since a pool of objective-program pairs is created offline. When creating a pool, the

solving time for a single optimization problem is under a second for most topologies on all scenarios on a 2.6 GHz 6-Core Intel Core i7 laptop with 16 GB memory, and we used a pool size of 1000 objectives. The only exception was the **NF** on the two largest topologies, Deltacom and Ion, which took 11.8 and 15.5 seconds respectively, and we used a smaller pool size in these cases to limit the pool generation time. Note that the pool creation occurs offline. Further, it involves solving multiple distinct optimization problems, and is trivially parallelizable.

To examine sensitivity to pool size, we first generated 5000 objective-program pairs and then randomly sampled a given number of objective-program pairs to form a candidate pool. Evaluating on candidate pools of size ranging from 10 to 5000, we found that pools with 300 objective-program pairs are sufficient for NET10Q to achieve over 99% optimal after 10 iterations. Please find details in Appendix 2.8.3.

2.5.4 Pilot User Study

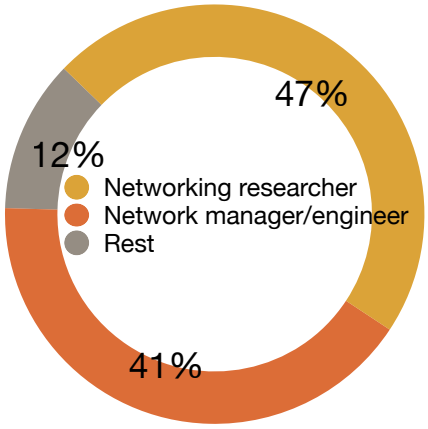
We next report on a small-scale study involving 17 users. The primary goal of the user study is to evaluate NET10Q when the real objective is arbitrarily chosen by the user, and even the actual shape is unknown to NET10Q. This is in contrast to the oracle experiments where the ground truth objectives are drawn from a template (with only the parameters unknown to NET10Q). Further, like with imperfect oracles, users may not always correctly express relative preferences.

The user study was conducted online using an IRB-approved protocol. Participants were recruited with a minimal qualifying requirement being they have taken a university course in computer networking. Figs 2.7a and 2.7b show the background of users. 88% of them are computer networking researchers or practitioners. 53% of users have more than 2 years of experience managing networks.

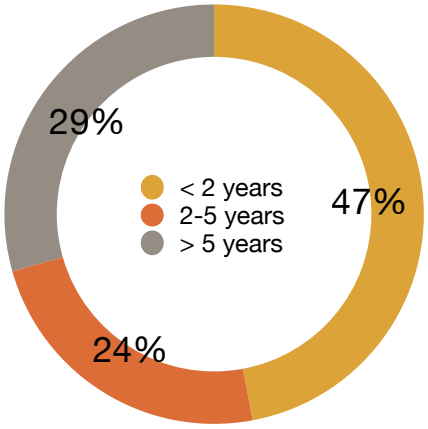
Our user study used an earlier version of Algorithm 1 implemented as an online web application. Specifically, the PCSes were generated on the fly, rather than pre-generated. To ensure responsiveness, the deployed algorithm set the threshold THRESH $= 2$. We note that the cloud application for our user study was developed and tested over multiple months, and

in parallel to refinements we developed to the algorithm. We were conservative in deploying the latest version given the need for a robust user-facing system, and to ensure all participants saw the same version of the algorithm.

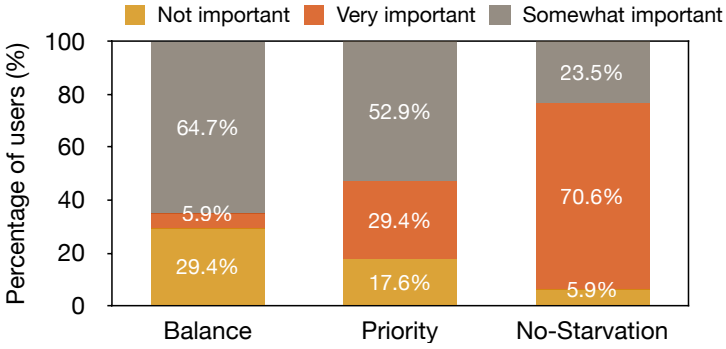
The study focused on the **BW** scenario (with four classes of traffic) and the Abilene topology. The user was free to choose any policy on how bandwidth allocations were to be made, and answer queries based on their policy. In each iteration, the user was asked to choose between two different bandwidth allocations generated by NET10Q. The user could either pick which allocation was better, or indicate it was too hard to call if the decisions were close. The study terminated after the user answered 10 questions, or when the user indicated she was ready to terminate the study.



(a) User background.

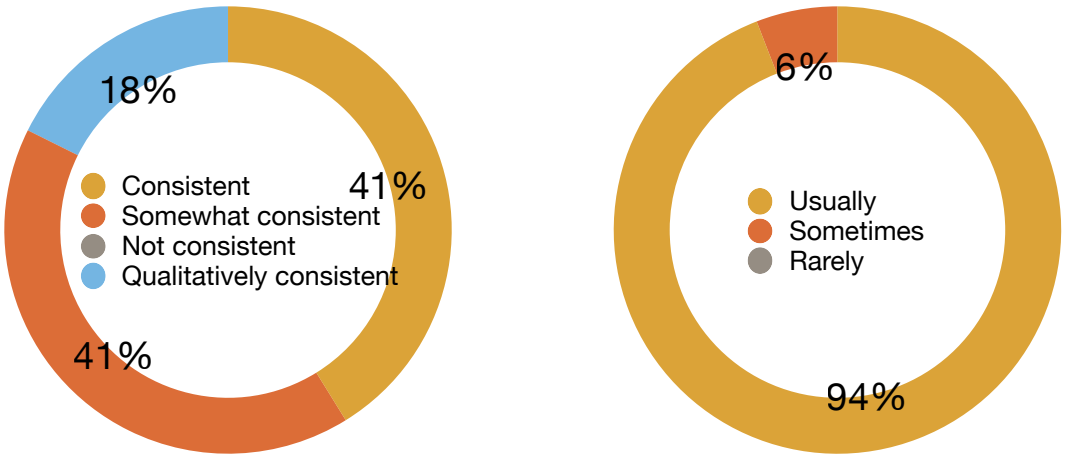


(b) User experience.



(c) Diversity of user policies.

Figure 2.7. User background and diversity of chosen policies.



(a) Quality of recommendations. (b) Was time taken acceptable?

Figure 2.8. Feedback on NET10Q from real users.

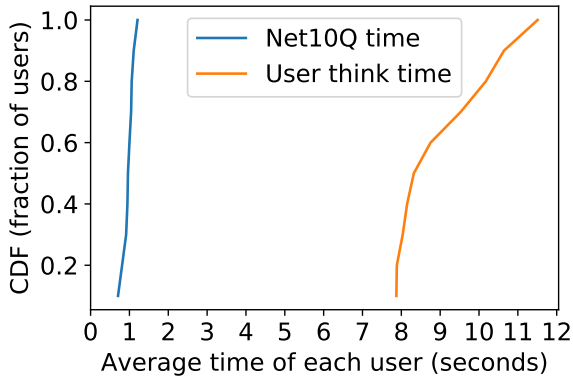


Figure 2.9. Avg. time per query across users.

In the post-study questionnaire, users were asked to characterize their policy by choosing how important it was to achieve each of three criteria below: (i) *Balance*, indicating allocation across classes is balanced; (ii) *Priority*, indicating how important it is to achieve a solution with more allocation to a higher priority class if a lower priority class does poorly; and (iii) *No-Starvation*, indicating how important it is to ensure lower priority classes get at least some allocation. Fig 2.7c presents a breakdown of user policies. 70.6% of users indicated *Balance* were somewhat or very important. 82.3% of users indicated *Priority* were somewhat or very important, while 94.1% of users indicated *No-Starvation* was somewhat or very important. The results were consistent with the qualitative description each user provided

regarding his/her policy. Overall, almost all users were seeking to achieve *Balance* and *Priority* avoiding the extremes (starvation) - however, they differed considerably in terms of where they lay in the spectrum based on their qualitative comments.

Results

Fig 2.8a summarizes how well the recommended allocations generated by NET10Q met user policy goals. 82% of the users indicated that the final recommendation is consistent, or somewhat consistent with their policies. The remaining 18% of the users took the study before we explicitly added the objective question to ask users to rate how well the recommended policy met their goals. However, the qualitative feedback provided by these users indicated NET10Q produced allocations consistent with user goals. For instance, one expert user said: *"The study was well done in my opinion. It put the engineer/architect in a position to make a qualified decision to try and chose the most reasonable outcome."*

Fig 2.8b shows that 94% of users indicated the response time with NET10Q was usually acceptable, while 6% indicated the time was sometimes acceptable. Fig 2.9 shows a cumulative distribution across users of the average NET10Q time (i.e., the average time taken by NET10Q between receiving the user's choice and presenting the next set of allocations). For comparison, the figure also shows a distribution of the average user think time (i.e., the average time taken by a user between when NET10Q presents the options and when the user submits her/his choice). The time taken by NET10Q was hardly a second, and much smaller than the user think time which varied from 8 to 12 seconds, indicating NET10Q can be used interactively.

Across all users, NET10Q was always able to find a satisfiable objective that met all of the user's preferences (it never needed to invoke the fallback approach (§2.5.2) of only considering a subset of user preferences). This indicates users express their preferences in a relatively consistent fashion in practice. Inconsistent responses may still allow for satisfiable objectives, however we are unable to characterize this in the absence of the exact ground truth objective.

Overall, the results show the promise of a comparative synthesis approach even when dealing with complex user chosen objectives of unknown shape. We believe there is potential for further improvements with all the optimizations in Algorithm 1, and other future enhancements.

2.6 Related Work

Network verification/synthesis. As we discussed in §2.2, the naïve approach to comparative synthesis proposed in [23] is preliminary and may involve prohibitively many queries. In contrast, we generalize and formalize the framework, design the first synthesis algorithm with the explicit goal of minimizing queries, present formal convergence results, and conduct extensive evaluations including a user study.

Much recent work applies program languages techniques to networking. Several works focus on synthesizing forwarding tables or router configurations based on predefined rules [48]–[54], or synthesizing provably-correct network updates [55], [56]. Much research focuses on verifying network configurations and dataplanes [27], [57], [58], and does not consider synthesis. Recent works mine network specification from configurations [59], generate code for programmable switches from program sketches [60], [61], or focus on generating network classification programs from raw network traces [62]. In contrast to these works, we focus on synthesizing network designs to meet quantitative objectives, with the objectives themselves not fully specified.

Optimal synthesis. There is a rich literature on synthesizing optimal programs with respect to a fixed or user-provided quantitative objective. Some of these techniques aims to solve optimal syntax-guided synthesis problems by minimizing given cost functions [25], [35]. Other approaches either generate optimal parameter holes in a program through probabilistic reasoning [38] or solve SMT-based optimization problems [63], under specific target functions.

In example-based synthesis, the examples as a specification can be insufficient or incompatible. Hence quantitative objectives can be used to determine to which extent a program satisfies the specification or whether some extra properties hold. Gulwani *et al.* [24] and Drosos *et al.* [64] defined the problem of quantitative PBE (qPBE) for synthesizing string-

manipulating programs that satisfy input-output examples as well as minimizing a given quantitative cost function.

Our work is different from all optimal synthesis work mentioned above as in our setting, the objective is unknown and automatically learnt/approximated from queries.

Human interaction. Many novel human interaction techniques have been developed for synthesizing string-manipulating programs. A line of work focuses on proposing user interaction models to help resolve ambiguity in the examples [65] and/or accelerate program synthesis [66], [67]. Using interactive approaches to solve multiobjective optimization problems has been studied by the optimization community for decades (as surveyed by Miettinen *et al.* [68]). Morpheus [69] is a routing control platform that allows users to flexibly specify their policy preferences. Morpheus requires pairwise comparisons on relative weights of metrics as input, which can be viewed as a special form of target functions.

Our novelty on the interaction method is to proactively ask comparative queries on concrete network designs, with the aim of minimizing the number of queries and maximizing the desirability of the found solution. The comparison of concrete candidates is easier than asking the user to provide rank scale, marginal rates of substitution or aspiration level, which is done by most existing approaches. The objectives we target to learn for network design also involve guard conditions, which is beyond what most existing methods can handle.

Oracle-guided synthesis. The learner-teacher interaction paradigm we use in the paper has been studied in the context of programming-by-example (PBE), aiming at minimizing the sequence of queries. Jha *et al.* [70] presented an oracle-based learning approach to automatic synthesis of bit-manipulating programs and malware deobfuscation over a given set of components. Their synthesizer generates inputs that distinguishes between candidate programs and then queries to the oracle for corresponding outputs. Ji *et al.* [71] followed up and studied how to minimize the sequence of queries. This line of work allows input-output queries only ("what is the output for this input?") to distinguish different programs. If two programs are distinguishable, they consider them equivalent or a ranking function is given. In invariant synthesis, Garg *et al.* [72] followed this paradigm and synthesized inductive invariants by checking hypotheses (equivalent to VALIDATE queries in our setting) with the teacher. Jha and Seshia [40] proposed a theoretical framework, called oracle-guided inductive

synthesis (OGIS) for inductive synthesis. The framework OGIS captures a class of synthesis techniques that operate via a set of queries to an oracle. Our comparative synthesis can be viewed as a new instantiation of the OGIS framework.

Active learning. Our algorithm for comparative synthesis has parallels to active learning [73], [74] in the machine learning community, which interactively queries a user to label data in settings where labeling is expensive. Query-by-committee (QBC) [75] is a general query strategy framework that chooses the most informative query based on the disagreements among a committee of models. How to construct the committee space and how to measure the disagreements among committee members are questions must be answered when instantiating the QBC framework. In contrast, we interactively query users to learn objectives using a carefully designed search space PCS and propose ways to estimate query informativeness specific to our setting.

2.7 Conclusions

In this paper, we have presented comparative synthesis for learning near-optimal programs with indeterminate objectives, and applied it to network design. First, we have developed a formal framework for comparative synthesis through queries with users. Second, we have developed the first learning algorithm for our framework that combines program search and objective learning, and seeks to achieve high solution quality with relatively few queries. We proved that the algorithm guarantees the median quality of solutions converges logarithmically to the optimal, or even linearly when the target function space is sortable (a property satisfied by two of our case studies). Third, we have developed NET10Q, a system implementing our approach. Experiments show that NET10Q only makes half or less queries than the baseline system to achieve the same solution quality, and robustly produces high-quality solutions with inconsistent teachers. A pilot user study with network practitioners and researchers shows NET10Q is effective in finding allocations that meet diverse user policy goals in an interactive fashion. Overall, the results show the promise of our framework, which we believe can help in domains beyond networking in the future.

2.8 Appendix: Additional Experimental Results

2.8.1 Evaluation on Perfect Oracle

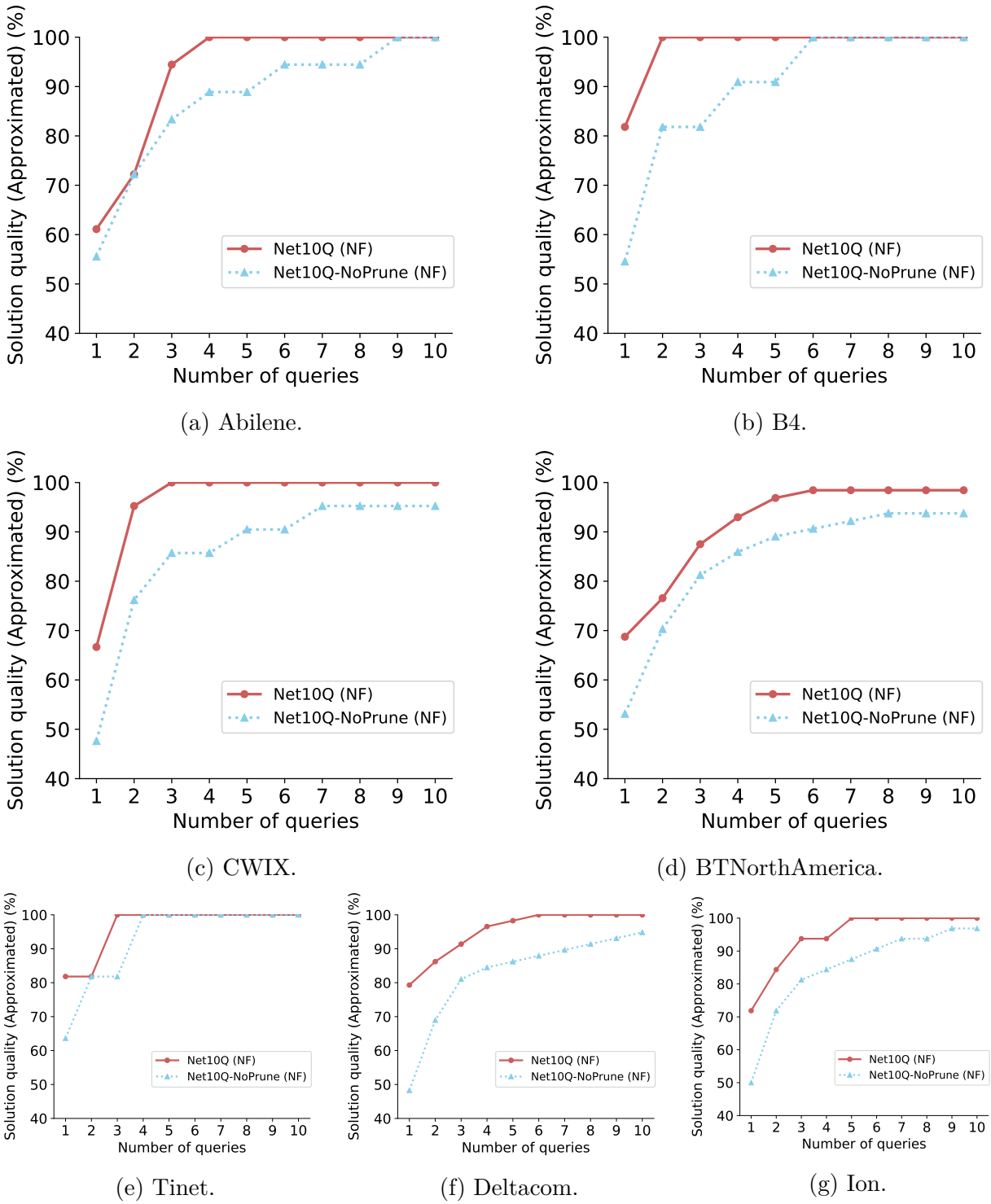
The range of solution quality achieved by NET10Q sometimes overlaps with NET10Q-NoPrune in Fig 2.5c – however NET10Q outperformed NET10Q-NoPrune in every scenario-topology combination. To demonstrate this for the **NF** scenario which saw the most overlap in Fig 2.5c, consider Fig 2.10 which presents a detailed breakdown of results by topology, and clearly illustrates NET10Q's out-performance.

2.8.2 Evaluation on Imperfect Oracle

Fig 2.6a showed NET10Q outperforms NET10Q-NoPrune for **BW** on CWIX. Fig 2.11 shows the performance of NET10Q and NET10Q-NoPrune when $p = 10$ for the other three scenarios, namely, **MCF**, **NF** and **OSPF**. NET10Q outperforms in these scenarios as well. Fig 2.12 presents the performance of NET10Q on CWIX when $p = 0, 5, 10, 20$ for **MCF**, **NF** and **OSPF**. While NET10Q shows some performance degradation at higher inconsistency levels ($p = 20$), it still achieved good solution quality.

2.8.3 Sensitivity to Size of Pre-Computed Pool

To examine how sensitive NET10Q is to the size of the pre-computed pool, we evaluated NET10Q with pools of different size for **BW** on CWIX, sampled from a larger pool of size 5000. Fig 2.13 shows the solution quality achieved by NET10Q after 10 queries. The solution quality in all cases was based on the rank computed on the entire pool of size 5000. The results indicate that performance does not substantially improve beyond a pool size of 300.



(a) Abilene.

(b) B4.

(c) CWIX.

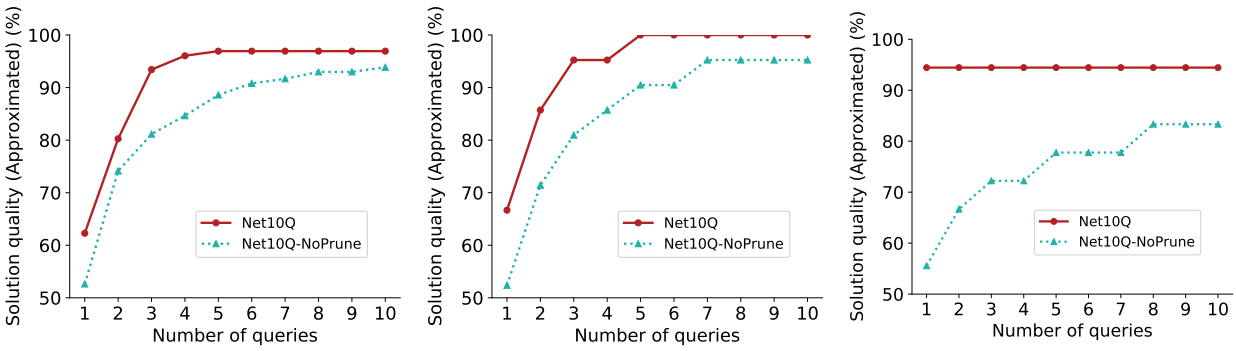
(d) BTNorthAmerica.

(e) Tinet.

(f) Deltacom.

(g) Ion.

Figure 2.10. Comparing NET10Q and NET10Q-NoPrune with perfect oracle for **NF** (each subfigure for a topology). Curves to the left are better. NET10Q outperforms in all topologies.



(a) MCF. (b) NF. (c) OSPF.

Figure 2.11. Performance of NET10Q with imperfect oracle ($p = 10$) for **MCF**, **NF** and **OSPF** on CWIX.

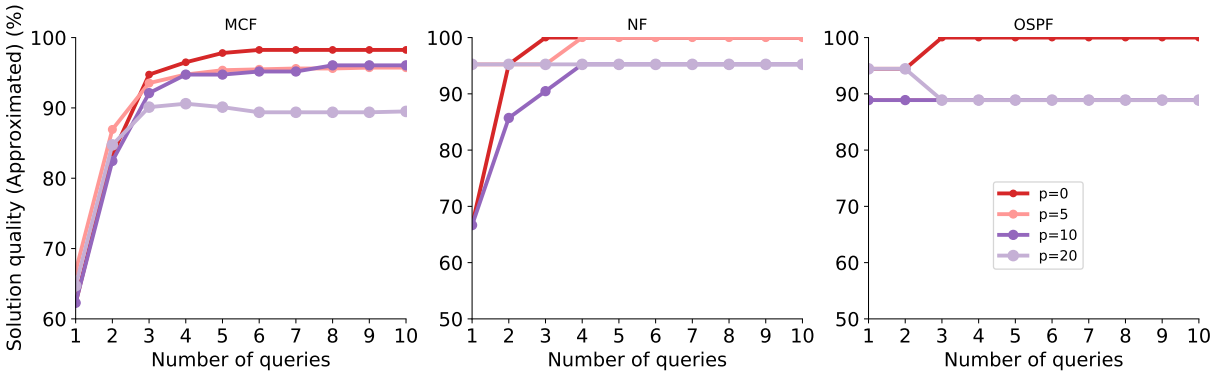


Figure 2.12. Performance of NET10Q under different level of inconsistency ($p = 0, 5, 10, 20$) on CWIX.

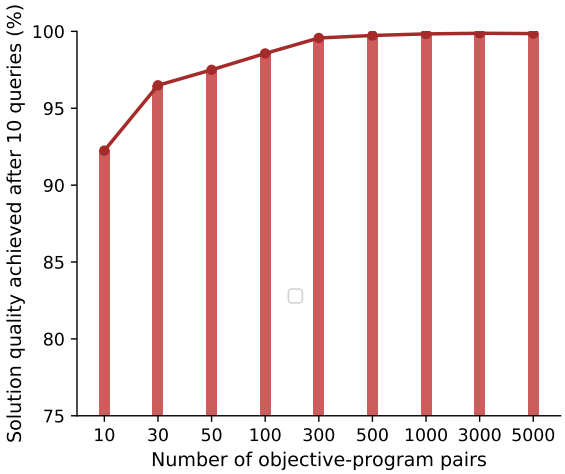


Figure 2.13. Performance of NET10Q with different size of pre-computed pool for **BW** on CWIX.

3. RECONCILING ENUMERATIVE AND DEDUCTIVE PROGRAM SYNTHESIS

In this chapter, we present a cooperative synthesis framework that combines enumerative and deductive synthesis with the aim of pushing the performance limit of syntax-guided synthesis.

3.1 Introduction

Syntax-guided synthesis (SyGuS) is a common theme underlying many program synthesis systems. The insight behind SyGuS is that to synthesize a large-scale program automatically, the user needs to provide not only a semantic specification but also a syntactic specification, i.e., a grammar of candidate programs as the search space. SyGuS has seen great success in the last decade, including the Sketch [3], [33], [76] synthesizer and the FlashFill feature of Microsoft Excel [2], [77]. The research community has also developed a standard interchange format for SyGuS problems and organized an annual competition, which encourages a plethora of syntax-guided synthesizers [78], [79].

The community usually categorizes synthesis techniques into two classes[1]: *enumerative synthesis* — which systematically enumerates possible implementations of the function to be synthesized and checks if it satisfies the desired specification; and *deductive synthesis* — which tries to reduce the specification to a desired program, purely symbolically by applying a series of deductive rules. Neither strategy clearly outperforms the other.

Enumerative synthesis traverses the search space following a specific strategy. The simplest strategy begins the search from smaller-sized candidates and moves toward larger-sized candidates. This naïve strategy guarantees to produce the smallest possible program, and is proven efficient for a wide spectrum of syntax-guided synthesis tasks. For example, EUSolver [4] adopts this strategy and has been the winner of the general track in 2016 and 2017 SyGuS competition [82], [83]. Other search strategies may perform better for different classes of problems. Stimulated by earlier success stories and the community's effort of standardization and competition [78], [79], researchers have proposed many novel

[1]↑E.g., see Sec 1.1 of [80], Lectures 2 and 17 of [81], and Table 2 of [1].

search strategies, including abstraction-based [66], [84]–[88], stochastic enumeration [36], [37], constraint-based [89], [90] and learning-based [8], [91]. Note that the appealing programming-by-example (PBE) and counterexample-guided inductive synthesis (CEGIS) techniques can also be viewed as a class of enumerative synthesis: the synthesizer is given a set of input-output examples and the search will be restricted to the programs whose behavior matches the given examples. As an example, LoopInvGen [92] leverages a learning-based variant of the CEGIS framework and won the invariant track in 2017 and 2018 SyGuS competition [83], [93]. Despite these algorithmic innovation, enumerative search is difficult to scale to large programs, because the search space grows exponentially with the size of the program.

Deductive synthesis is the oldest form of synthesis, dating back to Manna and Waldinger's field-defining paper [94] and even earlier work of Burstall and Darlington [95]. The deduction process essentially accepts a specification S and builds a constructive proof for the theorem "there exists a program satisfying S." Representative examples include Spiral [5], Paraglide [7], Fiat [6] and SusLik [96]. In general, the commonly known challenge for this paradigm of synthesis is the degree of automation, because critical steps of rule applications for synthesizing sophisticated programs, e.g., those involving loops, still rely on some guidance from the user. In recent years, researchers have automated deductive synthesis to symbolic synthesis procedures for several classes of synthesis problems, which run very efficiently. For example, CVC4 [97] embodies a symbolic synthesis algorithm called CEGQI to handle a class of integer arithmetic synthesis problems with the so called single invocation properties, and won the CLIA track of SyGuS competition four years in a row [79]. However, these procedures usually focus on synthesis problems in special domains with fixed grammars, and not applicable to more general synthesis tasks with arbitrary, user-provided grammars.

In recent years, the community has recognized the power of combining enumeration and deduction for synthesis [88], [98]–[100]. Nonetheless, existing techniques are not directly applicable to SyGuS for arbitrary grammars. In this paper, we focus on the class of SyGuS problems with the *CLIA* background theory but arbitrary grammar, and seek novel and amenable synergies of enumeration and deduction. We present a *cooperative synthesis* technique which switches between the two synthesis strategies to push the scalability. We

develop several divide-and-conquer strategies to split a large synthesis problem to smaller subproblems. The subproblems are solved separately and their solutions are combined to form a final solution. The technique integrates two synthesis engines: a pure deductive component for efficiently solving/simplifying the current problem whenever possible, and a height-based enumeration algorithm, as the last resort for handling arbitrary problem instances.

In this paper, we show our technique performs better than existing algorithms, and successfully solves many challenging problems not possible before. We summarize the contributions of this paper as below:

1. a **cooperative synthesis framework** that splits a synthesis problem into subproblems which are solved by deduction or enumeration separately (Section 3.3);

2. three novel **divide-and-conquer strategies** which allow splitting a wide variety of sophisticated synthesis problems (Section 3.4);

3. a **height-based enumeration** algorithm that splits the search space based on the height of the tree representation of the program and searches for each height symbolically (Section 3.5);

4. a set of general **deductive rules** that are powerful enough to solve/simplify many synthesis problems (Section 3.6);

5. the cooperative synthesis technique has been embodied in a SyGuS **solver** called DryadSynth, which solved more benchmarks than state-of-the-art solvers in every class of benchmarks, and tended to be more scalable for sophisticated benchmarks. 58 out of 715 benchmarks were solved uniquely by DryadSynth (Section 3.7).

3.2 Preliminaries

3.2.1 Syntax-Guided Synthesis

Definition 3.2.1 (Language)**.** *A language $\mathcal{L}$ is a tuple (Σ, τ) where Σ is an alphabet and τ maps every n-ary function name $f \in \Sigma$ to its signature $\tau(f) \in \{Bool, U\}^{n+1}$ (where U*

represents a universe). An $\mathcal{L}$-expression is an expression over symbols from Σ that conforms to their signatures τ. An $\mathcal{L}$-term is an $\mathcal{L}$-expression of type U. An $\mathcal{L}$-formula is a boolean $\mathcal{L}$-expression.

Definition 3.2.2 (Background Theory)**.** *A decidable background theory $\mathcal{T}$ with respect to a language $\mathcal{L}$ is a set of $\mathcal{L}$-formulae such that there is a decision procedure that takes a quantifier-free Σ-formula $\varphi(\boldsymbol{x})$ as input, and determines if $\mathcal{T} \models \varphi$, generates a counterexample vector $\boldsymbol{C}$ such that $\mathcal{T} \not\models \varphi(\boldsymbol{C})$, if such an $\boldsymbol{C}$ exists.*

Example 3.2.1. Consider a language $(\{0, 1, +, -, \geq, \texttt{ite}\}, \tau)$, where $\tau(0) = \tau(1) = (\mathbb{Z})$ as both 0 and 1 are constants, $\tau(+) = \tau(-) = (U, U, U)$ as they are binary functions over U, $\tau(\geq) = (U, U, \mathit{Bool})$ as $\geq$ is a binary relation. And finally, `ite` represents the *if-then-else* combination of a formula and two terms; therefore $\tau(\texttt{ite}) = (\mathit{Bool}, U, U, U)$. Then we let *CLIA* denote the standard theory for this language interpreted over $\mathbb{Z}$.

Definition 3.2.3 (Interpreted Function)**.** *An interpreted function for a language $\mathcal{L}$ is a tuple $(f,\ \Phi(x_1, \ldots, x_n))$, where f is the function name, and $\Phi(x_1, \ldots, x_n)$ is a well-typed $\mathcal{L}$-expression, i.e., each x_{i} is of type U or Bool, and the whole expression can be typed U or Bool.*

Example 3.2.2. Consider a binary function *qm* in the *CLIA* theory that returns the first non-negative argument. We declare this interpreted function as $(qm, \texttt{ite}(x_1 < 0), x_2, x_1)$.

Now we define the expression grammar, which essentially describes syntactic constraints for the expected program using a context-free grammar.

Definition 3.2.4 (Expression Grammar)**.** *An expression grammar $\mathcal{G}$ is a tuple $(\mathcal{T}, \mathcal{R}, \mathcal{N}, S, \mathcal{P})$, where $\mathcal{T}$ is a background theory with alphabet Σ, $\mathcal{R}$ is a set of interpreted functions for $\mathcal{L}$, $\mathcal{N}$ a set of non-terminal symbols (to be typed Bool or U, denoted as $\mathcal{N}_b$ and $\mathcal{N}_u$), $S \in \mathcal{N}$ is the start symbol, and $\mathcal{P} \subseteq \mathcal{N} \times \mathit{Exprs}(\Sigma, \mathcal{R}, \mathcal{N})$ is a set of production rules of form $T \to \epsilon$ or $T \to r(a_1, \ldots, a_n)$, where $T \in \mathcal{N}$, $r \in \mathcal{R} \cup \Sigma$, a_{i} is a free variable or a non-terminal in $\mathcal{N}$. Let $[\![\mathcal{G}]\!]$ denote the set of all expressions generated by $\mathcal{G}$: $\{\mathrm{e} \mid S \xrightarrow[\mathcal{P}]{}{}^{*} \mathrm{e}\}$.*

$$
\begin{array}{lcl}
S & \to & 0 \mid 1 \mid S + S \mid S - S \mid qm(S, S) \\
S' & \to & 0 \mid 1 \mid S' + S' \mid S' - S' \mid qm(S', S') \mid aux(S', S', S')
\end{array}
$$

(a) Grammar $\mathcal{G}_{qm}$

$$
\begin{array}{lcl}
S & \to & 0 \mid 1 \mid S + S \mid S - S \mid qm(S, S) \\
S' & \to & 0 \mid 1 \mid S' + S' \mid S' - S' \mid qm(S', S') \mid aux(S', S', S')
\end{array}
$$

(b) Grammar $\mathcal{G}^{+}_{qm}$

$$
\begin{array}{lcl}
qm(x_1, x_2) & \stackrel{\text{def}}{=} & \texttt{ite}(x_1 < 0, x_2, x_1) \\
aux(x_1, x_2) & \stackrel{\text{def}}{=} & \texttt{ite}(x_1 \geq x_2, x_1, x_2)
\end{array}
$$

(c) Interpreted functions

Figure 3.1. Production rules for Examples 3.2.3 and 3.3.1.

Example 3.2.3. Consider all possible expressions built using the *qm* function defined in Example 3.2.2, as well as variables x, y, z, and arbitrary constants. We call these expressions *qm-normal form* (*QNF*). Then formally *QNF* can be defined as an expression grammar

$$\mathcal{G}_{qm} \stackrel{\text{def}}{=} (CLIA\ , \{qm\}, \{S\}, S, \{x, y, z\}, \mathcal{P})$$

where $\mathcal{P}$ is the set of production rules presented in Figure 3.1a.

Example 3.2.4. We define $\mathcal{G}_{CLIA}$ as a special grammar. It takes *CLIA* as the background theory and allows all standard *CLIA* expressions.

Definition 3.2.5 (Uninterpreted Function)**.** *An n-ary uninterpreted function f is a sequence $\Big((x_1, t_1), \ldots, (x_n, t_1), rt\Big)$ where every pair $(x_{\mathrm{i}}, t_{\mathrm{i}})$ represents that the name of the* i*-th argument is x_{i} with type $t_{\mathrm{i}} \in \{Bool, U\}$, and $rt \in \{Bool, U\}$ is the return type of the function.*

Example 3.2.5. To synthesize a ternary function *max3*, we declare it as an uninterpreted function

$$((x, U), (y, U), (z, U), U)$$

.

Now we are ready to define the syntax-guided synthesis (SyGuS) problem we address in this paper.

Definition 3.2.6 (SyGuS Problem)**.** *An instance of the* SyGuS *problem is given by a tuple $(\mathcal{T}, f, \Phi, \mathcal{G})$ where $\mathcal{T}$ is a background theory with alphabet Σ, f is an n-ary uninterpreted function to be synthesized, Φ is a formula over $\Sigma \cup \{f\}$, and $\mathcal{G} = (\mathcal{T}, \mathcal{R}, \mathcal{N}, \{x_1, \ldots, x_n\}, \mathcal{P})$*

is an expression grammar. A solution to the SyGuS *problem is an expression* $E \equiv \lambda x_1, \ldots, x_n.\mathrm{e}(x_1, \ldots, x_n)$ *such that: a)* $\mathrm{e}(x_1, \ldots, x_n) \in [\![\mathcal{G}]\!]$*; b)* $\Phi[E/f]$ *is valid, i.e., instantiating* f *with* E *makes* Φ *valid. We use* $(\mathcal{T}, f, \Phi, \mathcal{G}) \rightsquigarrow E$ *to denote that* E *is a solution to the* SyGuS *problem* $(\mathcal{T}, f, \Phi, \mathcal{G})$.

Example 3.2.6. Recall the *max3* function declared in Example 3.2.5. Now we want to find an implementation of *max3* in $\mathcal{G}_{qm}$ that matches the semantics of the authentic implementation in *CLIA* . This is a SyGuS problem $(\mathit{CLIA}\ , \mathit{max3}, \Phi, \mathcal{G}_{qm})$ where Φ is the specification for the synthesis task:

$$\mathit{max3}(x, y, z) = \texttt{ite}(x \geq y \wedge x \geq z, x, \texttt{ite}(y \geq z, y, z)) \tag{3.2.1}$$

One solution to this problem is the following expression

$$\mathit{max3\_sol} \stackrel{\text{def}}{=} \lambda x, y, z.\Big(z + \mathit{qm}(x - z + \mathit{qm}(y - x, 0), 0)\Big) \tag{3.2.2}$$

Remark: To simplify the presentation, we assume there is a single function to be synthesized. However, the SyGuS definition can be easily extended to synthesize multiple functions. In the rest of the paper, we omit the background theory $\mathcal{T}$ when it is *CLIA* from the context. Unless stated otherwise, we also assume the function to be synthesized is always f. Then these components can be omitted from the tuples.

3.2.2 Counterexample-Guided Inductive Synthesis

The SyGuS problem is typically very challenging. Let $(\mathcal{T}, f, \Phi, \mathcal{G})$ be a SyGuS problem. Note that the specification Φ involves a vector of variables $\boldsymbol{x}$, and the synthesizer needs to find an implementation of f such that $\forall x.\Phi(\boldsymbol{x})$ holds. Checking this quantified formula is already undecidable for most background theories.

A common approach to addressing this problem is the Counterexample Guided Inductive Synthesis (CEGIS) framework [3], [90]. The basic idea is that a set of representative value

assignments C is usually sufficient to find a solution that works for all inputs. So the synthesis problem can be reduced to a constraint of the following form:

$$\exists f.\ \bigwedge_{\boldsymbol{c}\in C} \Phi(\boldsymbol{c}) \tag{3.2.3}$$

The set C is usually initialized to contain a random value. The synthesizer tries to solve (3.2.3) and find a candidate expression q. Then a verifier can check if the candidate works for all inputs, i.e.,

$$\mathcal{T} \models \Phi[q/f](\boldsymbol{x}) \tag{3.2.4}$$

Note that this query can be solved by the background decision procedure. If true, then q is a valid solution and the algorithm terminates; otherwise, a counterexample can be found, and the algorithm continues by adding the counterexample to C and the synthesizer tries to solve the inductive constraint again. The loop repeats until it finds a valid solution or hits a timeout.

3.2.3 Invariant Synthesis

In this paper we also address invariant synthesis, a special class of synthesis problems.

Definition 3.2.7 (Invariant synthesis problem)**.** *An invariant synthesis problem can be represented as* $\exists inv \forall \boldsymbol{x} \varphi(inv; \boldsymbol{x})$*, where inv is the predicate to be synthesized and* $\varphi(inv; \boldsymbol{x})$ *is of the form*

$$\begin{aligned}\varphi(inv;\boldsymbol{x}) \equiv \Big(pre(\boldsymbol{x}) \rightarrow inv(\boldsymbol{x})\Big) \wedge \Big(inv(\boldsymbol{x}) \rightarrow inv(\boldsymbol{trans}(\boldsymbol{x}))\Big) \\ \wedge \Big(inv(\boldsymbol{x}) \rightarrow post(\boldsymbol{x})\Big)\end{aligned}$$

where $pre(\boldsymbol{x})$ *and* $post(\boldsymbol{x})$ *are CLIA formulae ,* $\boldsymbol{trans}(\boldsymbol{x})$ *defines a vector of CLIA terms such that* $|\boldsymbol{trans}(\boldsymbol{x})| = |\boldsymbol{x}|$*.*

Intuitively, $(pre(\boldsymbol{x}), \boldsymbol{trans}(\boldsymbol{x}), post(\boldsymbol{x}))$ represents a program with a set of variables $\boldsymbol{x}$. $pre(\boldsymbol{x})$ and $post(\boldsymbol{x})$ are the pre- and post-conditions, respectively. $\boldsymbol{trans}(\boldsymbol{x})$ represents the iterative transition: $\boldsymbol{x} := \boldsymbol{trans}(\boldsymbol{x})$. The loop terminates when $\boldsymbol{trans}(\boldsymbol{x}) = \boldsymbol{x}$. The goal of

the synthesis problem is to find a loop invariant guaranteeing the partial correctness of the program with respect to *pre* and *post*.

Example 3.2.7. Consider a simple program of increasing variable x in a loop by 1 each iteration until it reaches 100:

int x = 0; **while** (x < 100) x = x + 1; **assert** x == 100;

The invariant synthesis problem for this program could be encoded to the following way:

$$\begin{aligned} pre(x) &\equiv (x = 0) \\ \boldsymbol{trans}(x) &\equiv \texttt{ite}(x < 100, x + 1, x) \\ post(x) &\equiv (\neg(x < 100) \Rightarrow (x = 100)) \end{aligned} \tag{3.2.5}$$

3.3 A Cooperative Synthesis Framework

In this section, we present a cooperative synthesis framework as a novel synergy of enumerative and deductive synthesis. In a nutshell, this framework encompasses a deductive synthesis engine and an enumerative synthesis engine, and solves synthesis problems by divide-and-conquer: it splits a synthesis problem into subproblems and solves them separately using deduction or enumeration.

3.3.1 Divide-And-Conquer Splitter

The cooperative synthesis framework features a divide-and-conquer splitter. The common pattern for these strategies is as follows: when the current synthesis problem p cannot be directly solved, the algorithm tries to identify a simpler problem (we call *Type-A Subproblem*) such that a solution to it can help simplify p to an easier-to-solve problem (we call *Type-B Subproblem*). We have identified several strategies that divide the original problem into subproblems A and B in different ways (see more details in Section 3.4).

Figure 3.2 illustrates the workflow of cooperative synthesis. Given a synthesis problem p, the deductive synthesis engine attempts to simplify/solve the input synthesis problem p purely deductively. If p is not completely solved, the divide-and-conquer splitter takes over

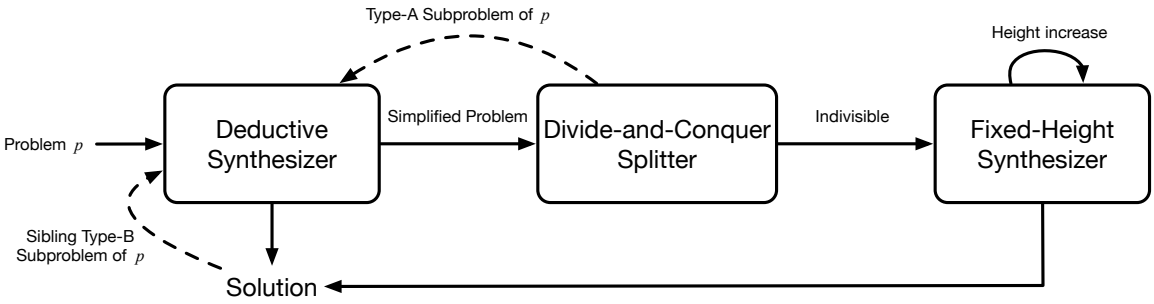


Figure 3.2. Workflow of cooperative synthesis.

and attempts to split the problem using a strategy. If the problem is divisible, the synthesis switches to the Type-A subproblem of p and starts over from the deductive synthesizer.

Otherwise, as the last resort, the problem is sent to the enumerative synthesizer. Notice that every possible solution has a syntax-tree representation, and the synthesizer just enumerates every height h and searches for syntax trees of fixed-height h, starting from 1, until a solution is found. Notice that this height-based enumeration guarantees to find the smallest possible solution, which is critical for many synthesis tasks that prefer compact solutions.

Whenever p is solved as a Type-A subproblem of another parent problem, the solution is used to generate the corresponding Type-B subproblem, for which the synthesis procedure repeats similarly.

3.3.2 Subproblem Graph

Notice that a problem can be split in multiple ways, split problem can be further split, and a subproblem can be generated and shared between multiple parent problems. We use a *subproblem graph* to represent the relations between problems.

Definition 3.3.1 (Subproblem Graph)**.** *Given a* SyGuS *problem S, a subproblem graph with respect to S is a directed acyclic graph (DAG) with a unique source (the node with no incoming edge) such that:*

- *every node represents a* SyGuS *problem; in particular, the source node represents S;*

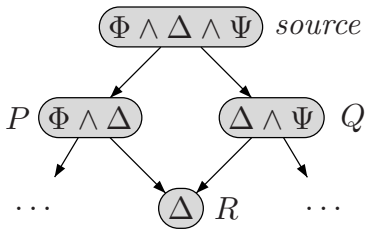


Figure 3.3. Example of subproblem graph.

- *if there is an edge from the node representing P to the node representing Q, then Q is a type-A subproblem of P based on any divide-and-conquer strategy described in Section 3.6 (subterm-based, fixed-term-based, and weaker-spec-based).*

For example, Figure 3.3 shows the subproblem graph, in which every node is annotated with the specification of the problem it represents. The source node represents the full specification $\Phi \wedge \Delta \wedge \Psi$. According to weaker-spec-based division (see Section 3.4.3), there are two Type-A subproblems $\Phi \wedge \Delta$ and $\Delta \wedge \Psi$, represented by the two successors P and Q, respectively. Moreover, the two subproblems can be further split to even simpler subproblems, among which R is their common subproblem because the specification Δ is the common conjunct of P and Q.

3.3.3 Cooperative Synthesis Algorithm

Algorithm 3 presents the overall cooperative synthesis algorithm. The algorithm takes as input a SyGuS problem $(f, \Phi, \mathcal{G})$ and maintains three data structures. PG is a subproblem graph with respect to the synthesis problem, initially built by buildGraph$(f, \Phi, \mathcal{G})$ (line 2). The procedure just builds a graph with a single source node representing Φ. *DedQueue* is a queue of (sub)problems to be solved by deduction; *EnumQueue* is a priority queue of (sub)problems to be solved by height-based enumeration. A (sub)problem of priority h is to be solved by fixed-height synthesis at height h (cf. Algorithm 4). Initially, *EnumQueue* is empty and *DedQueue* contains the source node of PG only.

The main part of the algorithm is a cooperative loop (lines 6–23) of deductive and enumerative synthesis which ends when a solution to the original problem is found. In each

```
   input  : A SyGuS problem (f, Φ, G)
   output: A solution λx.e(x), if any; otherwise ⊥
 1 def cooperative-synth(f, Φ, G):
 2     PG ← buildGraph(f, Φ, G)
 3     DedQueue ← emptyQueue()
 4     EnumQueue ← emptyPriorityQueue()
 5     enqueue (DedQueue, PG.source)
 6     repeat
 7         if DedQueue ≠ ∅ :
 8             p ← dequeue (DedQueue); h ← 0
 9             p.solution ← deduct (p)
10             if p.solution = ⊥ :
11                 p.succ ← TypeASubproblems (p)
12                 foreach c ∈ p.succ :
13                     enqueue (DedQueue, c)
14         elif EnumQueue ≠ ∅ :
15             (p, h) ← dequeue (EnumQueue)
16             p.solution ← fixed-height (p, h)
17         if p.solution = ⊥ :
18             enqueue (EnumQueue, p, h + 1)
19         elif p ≠ PG.source :
20             foreach t ∈ p.pred :
21                 t ← TypeBSubproblem(t, p.solution)
22                 enqueue (DedQueue, t)
23     until PG.source.solution ≠ ⊥ ;
24     return PG.source.solution
```

Algorithm 3: Cooperative synthesis framework.

iteration of the loop, the algorithm dequeues one task p from *DedQueue* or *EnumQueue*. As deductive synthesis has higher priority, the task p is always dequeued from *DedQueue* if it is nonempty; otherwise from *EnumQueue*.

If p is from *DedQueue* (lines 7–13), it will be handled by the **deduct** function which conducts pure deductive synthesis (which will be elaborated in Section 3.6). If p can't be solved, the algorithm will expand PG with all possible type-A subproblems of p, adding each one as successor of the node representing p. All these newly created problems will be added to *DedQueue* for solving in future iterations. If p is from *EnumQueue* and has priority h (lines 14–16), the algorithm invokes the **fixed-height** function (cf. Algorithm 4 in Section 3.5) to find a solution of p at height h.

For both cases, if no solution to p is found, the problem will be added back to *EnumQueue* with priority $h + 1$ (lines 17–18). [2] Otherwise, if a solution of p is found, as long as p is not the original problem, the solution will help simplify every parent problem of p to the corresponding type-B subproblem. The algorithm does the simplification through the TypeBSubproblem procedure and add the updated problem to *DedQueue* for future iterations (lines 19–22).

Example 3.3.1. Let us consider Example 3.2.6 and see how Algorithm 3 solves this problem. Recall that there is no specific rule in our deductive synthesis algorithm for the ad hoc operator qm, hence the **deduct** function cannot solve the original problem and adds it to *EnumQueue*. However, the algorithm finds that the reference implementation has a subterm $\texttt{ite}(y \geq z, y, z)$, which allows a subterm-based division (cf. Section 3.4.1). Again, the **deduct** function cannot solve subproblem A and add it to *EnumQueue*. Then in the next several iterations, the **fixed-height** function takes over and tries to find a height-1 solution of the original problem or the subproblem and fails; when the height is moved up to 2, solution (3.4.1) is found for subproblem A and simplify the original problem to subproblem B. Finally, **fixed-height** finds a solution subproblem B at height 2 as well and combine the two solutions to form the final solution to the whole problem, as shown in Equation 3.4.2. The whole procedure takes only 4 seconds to solve this problem. In contrast, this problem can be solved by neither height-based enumeration nor pure deduction alone. State-of-the-art SyGuS solver CVC4 [97] spent 28 minutes to solve it and EUSolver [4] timed out.

3.4 Divide-And-Conquer Strategies

In this section, we describe the three divide-and-conquer strategies we developed for splitting SyGuS problems. The cooperative synthesis technique can be extended with more splitting strategies in the future. We explain each of them through examples.

[2] ↑Note that h is set to 0 for the deduction case and the next search will be at height 1.

3.4.1 Subterm-Based Division

Let us start from subexpression-based division. Let us continue on Example 3.2.6. The solution to the SyGuS problem $(max3, \Phi, \mathcal{G}_{qm})$ (Expression 3.2.2) has a large syntax-tree representation (height 6 and size 13) and is difficult to be synthesized. If a synthesizer is stuck with this problem, one may wonder if it is possible to synthesize a simpler, auxiliary function equivalent to a subexpression of the target expression (3.2.1). For example, can we synthesize an auxiliary function aux such that $aux(y, z) = \texttt{ite}(y \geq z, y, z)$? Then the original synthesis problem has been divided into two subproblems:

- *Subproblem A:* synthesize the auxiliary function aux;
- *Subproblem B:* once an implementation of aux is found, add aux to the grammar and synthesize f with the new grammar.

For example, assume the following solution for Subproblem A has been found:

$$aux(x_1, x_2) \stackrel{\text{def}}{=} x_1 + qm(x_2 - x_1, 0) \tag{3.4.1}$$

Then we can extend the grammar $\mathcal{G}_{qm}$ with the new operator aux. The grammar extension forms Subproblem B and allows us to find the following solution:

$$f(x, y, z) \stackrel{\text{def}}{=} aux(z, aux(x, y)) \tag{3.4.2}$$

Note that both solutions (3.4.1) and (3.4.2) are small and easier to be synthesized than the original problem, and inlining the implementation of aux in (3.4.1) into (3.4.2) just yields the expected solution (3.2.2).

Formally, this divide-and-conquer strategy is formulated as the rule Subterm in Figure 3.4: when e' is a subexpression of e (denoted as $\mathrm{e}' \preccurlyeq \mathrm{e}$), we first solve $f(\boldsymbol{y}) = \mathrm{e}'$ as subproblem A, then solve $g(\boldsymbol{y}, \mathrm{e}') = \mathrm{e}$ as subproblem B, which is simpler than the original problem because g is allowed to use an extra argument e'.

$$\textsc{Subterm}\quad \frac{(\mathcal{T}, f, f(\boldsymbol{y}) = \mathrm{e}', \mathcal{G}) \rightsquigarrow P \quad (\mathcal{T}, g, g(\boldsymbol{y}, \mathrm{e}') = \mathrm{e}, \mathcal{G}) \rightsquigarrow Q}{(\mathcal{T}, f, f(\boldsymbol{y}) = \mathrm{e}, \mathcal{G}) \rightsquigarrow \lambda \boldsymbol{y}, y'.Q(\boldsymbol{y}, y')[P(\boldsymbol{y})/y']} \quad \text{if } \mathrm{e}' \preccurlyeq \mathrm{e}$$

$$\textsc{FixedTerm}\quad \frac{(\mathcal{T}, g, \Phi[\mathrm{e}/f(\mathrm{e})] \vee \Phi[g/f], \mathcal{G}) \rightsquigarrow P \quad (\mathcal{T}, f, f(\boldsymbol{y}, y') = \texttt{ite}(\Phi[\mathrm{e}/f(\mathrm{e})], \mathrm{e}, y'), \mathcal{G}) \rightsquigarrow Q}{(\mathcal{T}, f, \Phi, \mathcal{G}) \rightsquigarrow \lambda \boldsymbol{y}.Q(\boldsymbol{y}, P(\boldsymbol{y}))} \quad \text{if } f(\mathrm{e}) \sim \mathrm{e} \preccurlyeq \Phi \text{ for a connective } \sim$$

$$\textsc{WeakerSpec}\quad \frac{(\mathcal{T}, f, \Psi, \mathcal{G}) \rightsquigarrow P \quad (\mathcal{T}, g, \Phi[\lambda \boldsymbol{y}.\big(P(\boldsymbol{y}) \oplus g(\boldsymbol{y})\big)/f], \mathcal{G}) \rightsquigarrow Q}{(\mathcal{T}, f, \Phi, \mathcal{G}) \rightsquigarrow P \oplus Q} \quad \text{if } \mathcal{T} \models \Psi \preccurlyeq_{\oplus} \Phi$$

Figure 3.4. Deductive rules for divide-and-conquer.

3.4.2 Fixed-Term-Based Division

To understand fixed-term-based division, consider solving Example 3.6.1 using the CEGIS algorithm. Suppose a candidate solution $max2(x, y)$ is generated, even though it is not the expected solution, the candidate allows us to divide and simplify the synthesis problem. Notice that $max2(x, y)$ is a good implementation and satisfies the specification Φ when the inputs to the program satisfies $\Phi[max2(x, y)/f]$. Therefore, the synthesis problem can be divided into the following subproblems:

- *Subproblem A:* synthesize a function g that satisfies the specification only when the input does not satisfy $\Phi[max2(x, y)/f]$. In other words, the specification for g is $\Phi[max2(x, y)/f] \vee \Phi[g/f]$;
- *Subproblem B:* Combine $max2(x, y)$ and the synthesized function g to form an implementation that satisfies Φ for all inputs, no matter $\Phi[max2(x, y)/f]$ is satisfied or not.

Formally, this divide-and-conquer strategy can be formulated as the rule FixedTerm in Figure 3.4. Note that we apply this strategy only when $f(\mathrm{e}) \sim \mathrm{e} \preccurlyeq \Phi$, which means $f(\mathrm{e}) \sim \mathrm{e}$ occurs in Φ and $\sim$ is an arbitrary connective.

3.4.3 Weaker-Spec-Based Division

We illustrate how weaker-spec-based division works through the loop invariant synthesis problem defined in Definition 3.2.7. Recall that the specification consists of three parts:

$$\underbrace{pre(\boldsymbol{x}) \rightarrow inv(\boldsymbol{x})}_{\Phi} \ \wedge \ \underbrace{inv(\boldsymbol{x}) \rightarrow inv(\boldsymbol{trans}(\boldsymbol{x}))}_{\Delta} \ \wedge \ \underbrace{inv(\boldsymbol{x}) \rightarrow post(\boldsymbol{x})}_{\Psi}$$

When the whole synthesis problem is challenging and it is hard to generate appropriate $inv(\boldsymbol{x})$ to satisfy Φ, Δ and Ψ in tandem, one may divide the problem as follows:

- *Subproblem A:* synthesize an expression $P(\boldsymbol{x})$ that satisfies $\Phi \wedge \Delta$ (resp. $\Delta \wedge \Psi$);
- *Subproblem B:* synthesize an expression $Q(\boldsymbol{x})$ such that $P(\boldsymbol{x}) \wedge Q(\boldsymbol{x})$ (resp. $P(\boldsymbol{x}) \vee Q(\boldsymbol{x})$) satisfies the original specification $\Phi \wedge \Delta \wedge \Psi$.

Notice that both the two subproblems have weaker specifications than the original problem. Subproblem A is obviously easier as $P(\boldsymbol{x})$ only needs to satisfy two of the all three conjuncts. Subproblem B is also easier: imagine there is a solution $Q(\boldsymbol{x})$ to the original problem, it is also a solution to the subproblem B.

In loop invariant synthesis, the solutions P and Q from the subproblems are combined using conjunction or disjunction. However, weaker-spec-based division allows the combination P and Q using arbitrary binary functor $\oplus$. Now we define weaker specification in the most general way:

Definition 3.4.1 (Weaker Specification)**.** *Let $(\mathcal{T}, f, \Phi, \mathcal{G})$ be a* SyGuS *problem where f's return type is τ, and let $\oplus$ be a binary functor whose two input functions and output function are all of type τ, and $\oplus \stackrel{def}{=} \lambda g_1, g_2.\lambda \boldsymbol{y}.E(g_1(\boldsymbol{y}), g_2(\boldsymbol{y}))$ where $E(x, y) \in [\![\mathcal{G}(x, y)]\!]$. Then Ψ is a weaker specification of Φ with respect to $\oplus$, denoted as $\mathcal{T} \models \Psi \preccurlyeq_{\oplus} \Phi$, if the following conditions hold:*

1. $\mathcal{T} \models \Phi \rightarrow \Psi$;
2. $\mathcal{T} \models \forall g_1, g_2 : \Psi[g_1/f] \wedge \Psi[g_2/f] \rightarrow \Psi[g_1 \oplus g_2/f]$;
3. $\mathcal{T} \models \forall g_1, g_2 : \Big(\Psi[g_1/f] \wedge \Psi[g_1 \oplus g_2/f]\Big) \rightarrow (\Phi[g_1/f] \rightarrow \Phi[g_1 \oplus g_2/f])$.

With this general definition, we formulate the weaker-spec-based division as the rule WeakerSpec in Figure 3.4. Note that the loop invariant synthesis example we discussed above is just instances of the rule, in which $\oplus$ is instantiated to $\wedge$ or $\vee$.

3.4.4 Soundness and Completeness

All divide-and-conquer rules in Figure 3.4 are sound, as readers can verify. Although not all problems are splittable, these rules are complete in the sense that whenever a divide-and-conquer rule is applicable, the problem can be safely divided into subproblems without missing any possible solution. We formulate the completeness as the following theorem:

Theorem 3.4.1. *Let Ω be a* SyGuS *problem, and let Ω_A and Ω_B be a pair of subproblems obtained by dividing Ω using a divide-and-conquer strategy. If Ω has a solution P, then P is also a solution to Ω_A and Ω_B.*

Proof. The completeness can be verified for each rule in Figure 3.4 separately. In particular, if the division is a WeakerSpec division with respect to a weaker specification $\Psi \preccurlyeq_{\oplus} \Phi$, then according to Definition 3.4.1, the first and second conditions guarantee that P is also a solution to Ω_A and Ω_B, respectively. □

3.5 Fixed-Height Synthesis

Recall that the height-based enumeration algorithm sticks to a fixed size/height limit and searches for a solution within the limit symbolically, and gradually increase the limit when a solution of smaller size can't be found. While the algorithm is straightforward, it has its own merit and the idea does not seem be explored by any existing techniques. On the one hand, it still guarantees to synthesize the smallest satisfying program; on the other hand, it leverages as much power of symbolic solving as possible. In this section, we elaborate how the fixed-height synthesis component is implemented.

input : A SyGuS problem $p = (f, \Phi, \mathcal{G})$ and a positive integer h
output: A solution $\lambda \boldsymbol{x}.\mathrm{e}(\boldsymbol{x})$ such that the syntax-tree representation of $\mathrm{e}(\boldsymbol{x})$ is a full tree of height h, if any; otherwise $\perp$

```
// E_Φ is the set of counterexamples for spec Φ
1  def fixed-height(p, h) :
2      f ← p.target; Φ ← p.spec; G ← p.grammar
3      if h = 1 :
4          E_Φ ← ∅
5      q ← Init(G, h)
6      repeat
7          result ← verify(¬Φ[q/f])
8          if result = unsat :
9              break
10         else:
11             E_Φ ← E_Φ ∪ {result}
12             q ← ind-synth( ⋀_{e∈E} Φ[e/x], G, h)
13     until q = ⊥;
14     return q
```

Algorithm 4: Fixed-height synthesis.

3.5.1 Concrete Height Enumeration

Algorithm 3 solves the fixed-height synthesis problem through a function **fixed-height**. When the height of the solution's syntax tree is fixed to h, the synthesis problem is simplified and usually can be solved *purely symbolically*. In Algorithm 4, we present a simple implementation of **fixed-height** based on the standard CEGIS framework [3]. The algorithm assumes there are function **verify** as the verifier and function **ind-synth** as the inductive synthesizer for fixed-height solutions, and maintains a set of counterexamples E. The algorithm starts with a random candidate solution $\mathit{Init}(\mathcal{G}, h)$ (line 5). In each following iteration, the synthesizer proposes a height-h candidate solution q that satisfies the specification when $\boldsymbol{x}$ is assigned values from E (line 12). Then the verifier checks condition (3.2.4), i.e., whether the candidate satisfies the specification Φ (line 7). If the result is unsat, then q is the desired implementation and the algorithm terminates; otherwise the verifier reports a counterexample as the witness of the failed verification and add it to E (line 11). Then the CEGIS loop continues with the next iteration repeatedly, until a solution is found.

Parallelization. While the naïve height-based enumeration in Algorithm 3 incrementally searches all possible heights of the decision tree, starting from 1, the enumeration can be naturally parallelized. If there are n cores available on the machine, the parallelized version runs the **fixed-height** algorithm at n different heights on n threads while sharing the set of counterexamples among them with proper synchronization. The algorithm starts with the n smallest heights, $\{1, \ldots, n\}$. It also maintains a variable k as the next height to search, starting from $n + 1$. Whenever a thread concludes that there is no solution at the current height, it starts a new CEGIS loop at height k, and the value of k gets increased. The whole algorithm stops whenever a thread finds a solution.

3.5.2 Symbolic Inductive Synthesis

In Algorithm 4, **verify** is just the standard background decision procedure, and **ind-synth** is an inductive synthesizer with the assumption that the solution's height is up to h. In our framework, this synthesis task is encoded and symbolically solved by the background decision procedure. We first illustrate the idea with the assumption that the grammar is $\mathcal{G}_{CLIA}$, then extend the encoding to arbitrary grammar $\mathcal{G}$.

Decision Tree Representation. Let the input SyGuS problem be $(f, \Phi, \mathcal{G}_{CLIA})$, then any implementation of f can be represented in a *decision tree normal form*, as described in Figure 3.5. It is not hard to see that every *CLIA* expression can be converted to this normal form. The decision tree representation of a *CLIA* expression is a binary tree in which every node with id i contains a vector $\boldsymbol{c}_\text{i}$ of integer constants and an extra constant d_i. Then each decision node (non-leaf node) tests whether $\boldsymbol{c}_\text{i} \cdot (\boldsymbol{x} \oplus (1)) \geq 0$ and according to the test result proceeds to the "true" child or "false" child. Each leaf node determines the value of the function as $\boldsymbol{c}_\text{i} \cdot (\boldsymbol{x} \oplus (1))$. For example, if f is a binary function and the solution is a binary max function of height 2: $f(x_1, x_2) \stackrel{def}{=} \texttt{ite}(x_1 \geq x_2, x_1, x_2)$. It can be represented as the tree shown in Figure 3.6. Notice that the full decision tree of height h consists of $2^h - 1$ nodes, and the node id's can be fixed in the range between 0 and $2^h - 2$. This allows us to reduce **ind-synth** to the problem of searching for the vector $\boldsymbol{c}_\text{i}$ for each node i.

$$
\begin{array}{rcl}
\text{Int Const Vector: } \boldsymbol{c}_{\mathrm{i}} & & \text{Int Const: } d_{\mathrm{i}} \\
\text{Atom Expr: e} & ::= & \boldsymbol{c}_{\mathrm{i}} \cdot x + d_{\mathrm{i}} \\
\text{Atom Cond: } \alpha & ::= & \mathrm{e} \geq 0 \\
\text{Expr: } E, E_1, E_2 & ::= & \mathrm{e} \mid \texttt{ite}(\alpha, E_1, E_2) \\
\text{Condition: } \varphi & ::= & \alpha \mid \texttt{ite}(\alpha, \varphi_1, \varphi_2)
\end{array}
$$

Figure 3.5. Decision tree normal form.

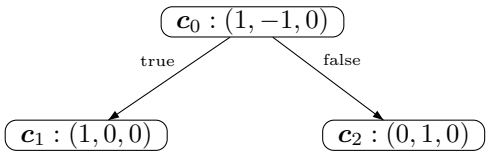


Figure 3.6. Representation of the $max2$ function.

Interpret Function. Then for a fixed height h, we can build an $\mathsf{interpret}_h$ function that interprets the vectors back to the function of height h. For any function f of height h, its decision tree can be represented as 2^{h-1} vectors $\boldsymbol{c}_0, \ldots, \boldsymbol{c}_{2^h-1}$. Then for any vector of constants $\boldsymbol{d}$, $\mathsf{interpret}_h$ essentially interprets the decision tree on $\boldsymbol{d}$ and determines the value of $f(\boldsymbol{d})$. In other words, $\mathsf{interpret}_h(\boldsymbol{c}_0, \ldots, \boldsymbol{c}_{2^h-1}, \boldsymbol{d}) = f(\boldsymbol{d})$. For example, continuing on the example of Figure 3.6. Assume $\boldsymbol{d} = (1, -2)$, then the value of $f(\boldsymbol{d})$ can be computed as:

$$
\begin{aligned}
&\mathsf{interpret}_2(\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{d}) \\
&\quad = \texttt{ite}\Big(\boldsymbol{c}_0 \cdot \boldsymbol{d} \oplus (1) \geq 0, \quad \boldsymbol{c}_1 \cdot \boldsymbol{d} \oplus (1), \quad \boldsymbol{c}_2 \cdot \boldsymbol{d} \oplus (1)\Big) \\
&\quad = \texttt{ite}\Big(\boldsymbol{c}_0 \cdot (1, -2, 0) \geq 0, \quad \boldsymbol{c}_1 \cdot (1, -2, 0), \quad \boldsymbol{c}_2 \cdot (1, -2, 0)\Big)
\end{aligned}
$$

Now to solve **ind-synth(** $\bigwedge_{\mathrm{e} \in E} \Phi[\mathrm{e}/\boldsymbol{x}]$, $\mathcal{G}$, h**)**, we can replace every occurrences of f in $\Phi[\mathrm{e}/\boldsymbol{x}]$ with a corresponding $\mathsf{interpret}$ function. The resulting *CLIA* formula involves variables $\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2$ only and can be solved by a single SMT query.

Extension to General Grammar. We have generalized above encoding to arbitrary grammar $\mathcal{G}$. As an example, consider the $\mathcal{G}_{qm}$ grammar defined in Figure 3.1a. In the

decision-tree representation, non-leaf nodes will represent the qm function invocation, which can be interpreted by the following adapted $\mathsf{interpret}_2$ function:

$$\begin{aligned}
&\mathsf{interpret}_2(\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{d}) \\
&\quad = qm\Big(\mathsf{interpret}_1(\boldsymbol{c}_1, \boldsymbol{d}), \mathsf{interpret}_1(\boldsymbol{c}_2, \boldsymbol{d})\Big) \\
&\quad = \texttt{ite}\Big(\boldsymbol{c}_1 \cdot (1, -2, 0) < 0,\ \boldsymbol{c}_1 \cdot (1, -2, 0),\ \boldsymbol{c}_2 \cdot (1, -2, 0)\Big)
\end{aligned}$$

This generalization allows us to solve arbitrary SyGuS problems with the *CLIA* background theory (see the General track benchmarks in Section 3.7). We leave further generalization to other background theories, e.g., bit vectors, to future work.

3.6 The Deductive Component

In this section, we introduce the deductive component of the framework (i.e., the **deduct** function in Algorithm 3). This component integrates a set of deductive rules that can simplify the specification Φ or find a solution directly. The implementation, as shown in Algorithm 5, just repeatedly and exhaustively applies these rules to simplify the specification Φ. If the simplified Φ is already a solution (of the form $f(x_1, \ldots, x_n) = \mathrm{e}$), return the solution; otherwise return $\bot$. As deduction can be performed very efficiently, this component serves as the first step for all (sub)problems.

Note that our deductive rules are designed as a component for the cooperative synthesis framework, they are not expected to be complete in any sense. That said, they are already powerful enough to solve many synthesis problems. For instance, the rules in Figures 3.7 and 3.8 have already superseded the class of *Single Invocation Problems*, a common class of problems that can be solved using the counterexample-guided quantifier instantiation algorithm [97].

We next present deductive rules that are general and applicable to arbitrary grammar, followed by special simplification for two special classes of problems. To the best of our knowledge, these rules are not explicitly integrated in any existing deductive synthesizer. Our framework can also integrate more deductive rules in the future.

input : A SyGuS problem $p = (f, \Phi, \mathcal{G})$
output: A solution $\lambda \boldsymbol{x}.\mathrm{e}(\boldsymbol{x})$, if any; otherwise $\perp$

```
def deduct(p):
    f ← p.target; Φ ← p.spec; G ← p.grammar
    Φ ← Simplify(f, Φ, G); p.spec ← Φ
    if IsSolution(Φ, G) :
        return Φ
    else:
        return ⊥
```

Algorithm 5: Deductive synthesis.

General Deduction. Figure 3.7 shows a set of general deductive rules for arbitrary grammar. Assuming f is the function to be synthesized, these rules soundly substitute occurrences of f, arguments or variables with a concrete implementation. Most of the rules are self explanatory. In particular, the last rule Match applies when the specification is a reference implementation $f(\boldsymbol{y}) = \mathrm{e}$ but e does not conform to the grammar $\mathcal{G}$. In that case, we can exhaustively match and replace subexpressions of e with interpreted functions in $\mathcal{G}$, and check if the final expression falls in $[\![\mathcal{G}]\!]$. For example, let e be $x + x + x + x$ and let $\mathcal{G}$ be a grammar that contains only one operator $\mathit{double}(x) \stackrel{\text{def}}{=} x + x$, then e can be rewritten to $\mathit{double}(\mathit{double}(x))$.

Merging and Substituting for CLIA. For $\mathcal{G}_{CLIA}$, a very common grammar for syntax-guided synthesis, we designed a set of ad hoc rules as illustrated in Figure 3.8. Intuitively, these rules find two occurrences of f and merge them into a single occurrence.

Example 3.6.1. Let $\mathcal{G}$ be a grammar with only an operator $\mathit{max2}(x, y) \stackrel{\text{def}}{=} \texttt{ite}(x \geq y, x, y)$. Our deductive synthesis algorithm can synthesize a ternary maximum function $f(x, y, z)$ using the rewriting sequence shown in Figure 3.9.

Loop Summary for Invariant Synthesis. We also developed a special class of simplification rules for loop invariant synthesis. The idea is to find a predicate that precisely summarizes the effect of arbitrary k-steps of loop transformation. Formally, if there exists a binary predicate ***fast-trans*** such that

$$\boldsymbol{\mathit{fast\text{-}trans}}(\boldsymbol{x}, \boldsymbol{y}) \Leftrightarrow \exists k \geq 0. \boldsymbol{\mathit{trans}}^k(\boldsymbol{x}) = \boldsymbol{y}$$

$$
\begin{array}{rcl}
\textsc{IntEq} & & \\
f(\boldsymbol{y}) = \mathrm{e} \wedge \Psi & \Longrightarrow & f(\boldsymbol{y}) = \mathrm{e} \wedge \Psi[\lambda \boldsymbol{y}.\mathrm{e}/f] \\
\textsc{IntNeq} & & \\
f(\boldsymbol{y}) \neq \mathrm{e} \vee \Psi & \Longrightarrow & f(\boldsymbol{y}) \neq \mathrm{e} \vee \Psi[\lambda \boldsymbol{y}.\mathrm{e}/f] \\
\textsc{BoolPos} & & \\
(f(\boldsymbol{y}) \vee \Phi) \wedge \Psi & \Longrightarrow & \Psi[\lambda \boldsymbol{y}.((\neg\Phi) \vee f(\boldsymbol{y}))/f] \\
& & \text{if } f \text{ does not occur in } \Phi \\
\textsc{BoolNeg} & & \\
(\neg f(\boldsymbol{y}) \vee \Phi) \wedge \Psi & \Longrightarrow & \Psi[\lambda \boldsymbol{y}.(\Phi \wedge f(\boldsymbol{y}))/f] \\
& & \text{if } f \text{ does not occur in } \Phi \\
\textsc{RemoveVar} & & \\
\Psi & \Longrightarrow & \Psi[0/y_{\mathrm{i}}] \quad \text{if } \mathcal{T} \models \Phi \leftrightarrow \Phi[y'_{\mathrm{i}}/y_{\mathrm{i}}] \\
\textsc{RemoveArg} & & \\
(f, \Phi, \mathcal{G}) & \Longrightarrow & (g, \Phi[g(\mathrm{e}, \mathrm{e}')/f(\mathrm{e}, C, \mathrm{e}')], \mathcal{G}) \\
& & \text{if the i-th arg of } f \text{ is always constant } C \\
\textsc{Match} & & \\
(f, f(\boldsymbol{y}) = \mathrm{e}, \mathcal{G}) & \Longrightarrow & (f, f(\boldsymbol{y}) = \mathrm{e}', \mathcal{G}) \\
& & \text{if } \mathrm{e} \Longrightarrow^*_{\mathcal{G}} \mathrm{e}' \text{ and } \mathrm{e}' \in [\![\mathcal{G}(\boldsymbol{y})]\!]
\end{array}
$$

Figure 3.7. Deductive rules for arbitrary grammar.

then the original specification can be reduced to a simpler constraint:

$$\Big(\big(\mathit{pre}(\boldsymbol{x}) \wedge \boldsymbol{\mathit{fast\text{-}trans}}(\boldsymbol{x}, \boldsymbol{y})\big) \rightarrow \mathit{inv}(\boldsymbol{y})\Big) \wedge \Big(\mathit{inv}(\boldsymbol{y}) \rightarrow \mathit{post}(\boldsymbol{y})\Big)$$

For example, the loop transformation in Example 3.2.7 can be summarized as:

$$\boldsymbol{\mathit{fast\text{-}trans}}(x, y) \equiv (x < 100 \wedge x \leq y) \vee x = y$$

3.7 Experimental Evaluation

We have prototyped our cooperative synthesis technique as a system called DryadSynth,[3] which supports the *CLIA* background theory. DryadSynth is written in Java with around 11k LOC, and employs Z3 [41] as the constraint solving engine. This is a relatively small and lightweight implementation in terms of engineering (Comparing to, e.g. 343k+ LOC of the CVC4 code base and 33k+ LOC of the EUSolver code base).

[3]↑https://github.com/purdue-cap/DryadSynth

GeMax
$f(\mathrm{e}) \geq \mathrm{e}_1 \wedge f(\mathrm{e}) \geq \mathrm{e}_2 \implies f(\mathrm{e}) \geq \mathtt{ite}(\mathrm{e}_1 \geq \mathrm{e}_2, \mathrm{e}_1, \mathrm{e}_2)$

LeMin
$f(\mathrm{e}) \leq \mathrm{e}_1 \wedge f(\mathrm{e}) \leq \mathrm{e}_2 \implies f(\mathrm{e}) \leq \mathtt{ite}(\mathrm{e}_1 \geq \mathrm{e}_2, \mathrm{e}_2, \mathrm{e}_1)$

GeMin
$f(\mathrm{e}) \geq \mathrm{e}_1 \vee f(\mathrm{e}) \geq \mathrm{e}_2 \implies f(\mathrm{e}) \geq \mathtt{ite}(\mathrm{e}_1 \geq \mathrm{e}_2, \mathrm{e}_2, \mathrm{e}_1)$

LeMax
$f(\mathrm{e}) \leq \mathrm{e}_1 \vee f(\mathrm{e}) \leq \mathrm{e}_2 \implies f(\mathrm{e}) \leq \mathtt{ite}(\mathrm{e}_1 \geq \mathrm{e}_2, \mathrm{e}_1, \mathrm{e}_2)$

Eq
$f(\mathrm{e}) \geq \mathrm{e}_1 \wedge f(\mathrm{e}) \leq \mathrm{e}_2 \implies f(\mathrm{e}) = \mathrm{e}_1$
if $\mathcal{T} \models \mathrm{e}_1 = \mathrm{e}_2$

NotEq
$f(\mathrm{e}) \geq \mathrm{e}_1 \vee f(\mathrm{e}) \leq \mathrm{e}_2 \implies f(\mathrm{e}) \neq \mathrm{e}_1 - 1$
if $\mathcal{T} \models \mathrm{e}_1 = \mathrm{e}_2 + 2$

CNF
$(\Phi \vee \Psi_1) \wedge (\Phi \vee \Psi_2) \implies \Phi \vee (\Psi_1 \wedge \Psi_2)$
if f does not occur in Ψ_1 or Ψ_2

Figure 3.8. Deductive rules for $\mathcal{G}_{CLIA}$.

To evaluate our algorithms, we compared DryadSynth with state-of-the-art SyGuS solvers, CVC4, EUSolver and LoopInvGen. [4] They are winning solvers in recent years' SyGuS competition and we used the latest version from their public repositories. CVC4 and EUSolver are two general-purpose solvers that participate in the General, CLIA and INV tracks. LoopInvGen focuses on INV track only.

Experimental Setting. Experiments were conducted on the StarExec platform [102], on which each solver is executed on a 4-core, 2.4GHz CPU and 128GB memory node, with a 30-minute timeout. We adopted 403 INV benchmarks, 88 CLIA benchmarks, and 224 General track benchmarks with the *CLIA* background theory included in the SyGuS competition of 2019. [5] We excluded 372 General benchmarks that are based on the *BitVector* background theory and 426 INV benchmarks that contain let-macros, which DryadSynth does not support at present. It wound up with all of 715 benchmarks.

[4]↑We omitted other solvers as they focus on other background theories and not comparable with ours. For example, while Euphony's AI-guided algorithm [101] is promising, it supports String and BitVector theories only, and its old algorithm was not competitive in the *CLIA* theories.
[5]↑We slightly adapted 21 of General benchmarks to remove the let-macros.

$$
\begin{aligned}
&f(x,y,z) \geq x \ \wedge \ f(x,y,z) \geq y \ \wedge \ f(x,y,z) \geq z \ \wedge \\
&\Big(f(x,y,z) = x \vee f(x,y,z) = y \vee f(x,y,z) = z\Big) \xRightarrow{\text{CNF}} \\
&\quad f(x,y,z) \geq x \ \wedge \ f(x,y,z) \geq y \ \wedge \ f(x,y,z) \geq z \\
&\quad \wedge \ \Big(f(x,y,z) \geq x \vee f(x,y,z) \geq y \vee f(x,y,z) \geq z\Big) \\
&\quad \wedge \ \Big(f(x,y,z) \leq x \vee f(x,y,z) \leq y \vee f(x,y,z) \leq z\Big) \\
&\quad \wedge \ \ldots \xRightarrow{\text{GeMax,LeMax,...}} \\
&f(x,y,z) \geq \texttt{ite}(\texttt{ite}(x \geq y, x, y) \geq z, \texttt{ite}(x \geq y, x, y), z) \\
&\ \wedge \ f(x,y,z) \leq \texttt{ite}(\texttt{ite}(x \geq y, x, y) \geq z, \texttt{ite}(x \geq y, x, y), z) \\
&\ \wedge \ \ldots \xRightarrow{\text{Eq,IntEq}} \\
&\quad f(x,y,z) = \texttt{ite}(\texttt{ite}(x \geq y, x, y) \geq z, \texttt{ite}(x \geq y, x, y), z) \\
&\quad \xRightarrow{\text{Match}} f(x,y,z) = \mathit{max2}(\mathit{max2}(x,y),z)
\end{aligned}
$$

Figure 3.9. Rewriting sequence for Example 3.6.1.

We summarize the experimental results through a set of figures. Figures 3.10 and 3.11 compare the number of benchmarks correctly solved within the 30-minute limit and the number of benchmarks solved the fastest among all solvers, respectively. [6] Figure 3.12 shows the comparison of the solvers in terms of the number of benchmarks solved and the total amount of time spent. Figure 3.13 shows the amounts of time spent for every benchmark, sorted in ascending order. All figures break down the comparison by tracks. It is noteworthy that DryadSynth allows us to solve **58** benchmarks which were not solvable by other synthesizers, while LoopInvGen have 9 benchmarks uniquely solved.

Observation. Observing the figures, we are encouraged by the following facts: 1) Figures 3.10 and 3.11 show that DryadSynth solved and *fastest* solved more benchmarks than all other solvers in all tracks. 2) Figure 3.12 indicates that DryadSynth solved more CLIA and General benchmarks than all other solvers, with less total time spent. 3) Figure 3.13 shows that DryadSynth has better scalability than all other synthesizers: although DryadSynth had a constant overhead on easier-to-solve problems, the solving time increases more mildly toward more challenging benchmarks than other synthesizers.

[6] ↑Following the criterion of SyGuS competition, the time amounts are classified into buckets of pseudo-logarithmic scales: $[0,1), [1,3), [3,10), \ldots, [1000, 1800)$.

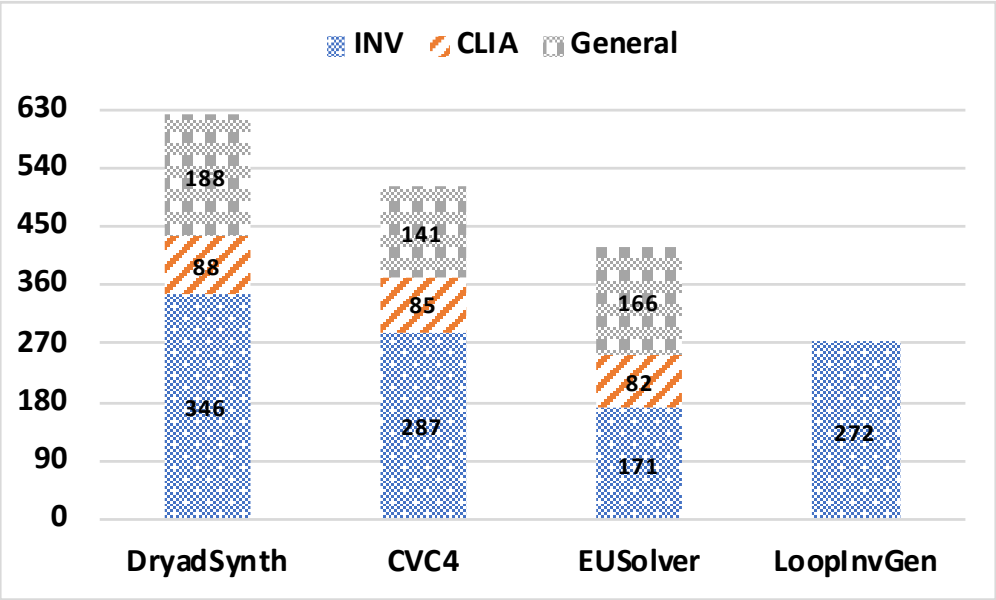


Figure 3.10. Solved benchmarks (breakdown by tracks).

Table 3.1. Number of smallest solutions and median of solution size (in small text). Best numbers in grey.

Track	DryadSynth	CVC4	EUSolver	LoopInvGen
INV	132 38	118 26	171 3	220 7
CLIA	56 278.5	56 361	67 201.5	-
General	141 19	124 19	166 19	-

In summary, our cooperative synthesis technique outperforms state-of-the-art synthesizers in both scalability and diversity of the solved problems, and tends to be a general and efficient synthesis engine for syntax-guided synthesis.

Remark: We also roughly compared the size of solutions as our deductive component does not control the solution size, as shown in Table 3.1. Based on the number of smallest solutions [7] and the median size of solutions for the commonly solved benchmarks, DryadSynth is slightly better than CVC4 but worse than EUSolver (purely enumerative) and LoopInvGen (for INV track only).

[7]↑Following the criterion of SyGuS competition, the size amounts are classified into buckets of pseudo-logarithmic scales: [1,10), [10,30), [30,100), [100,300), [300,1000), ≥ 1000.

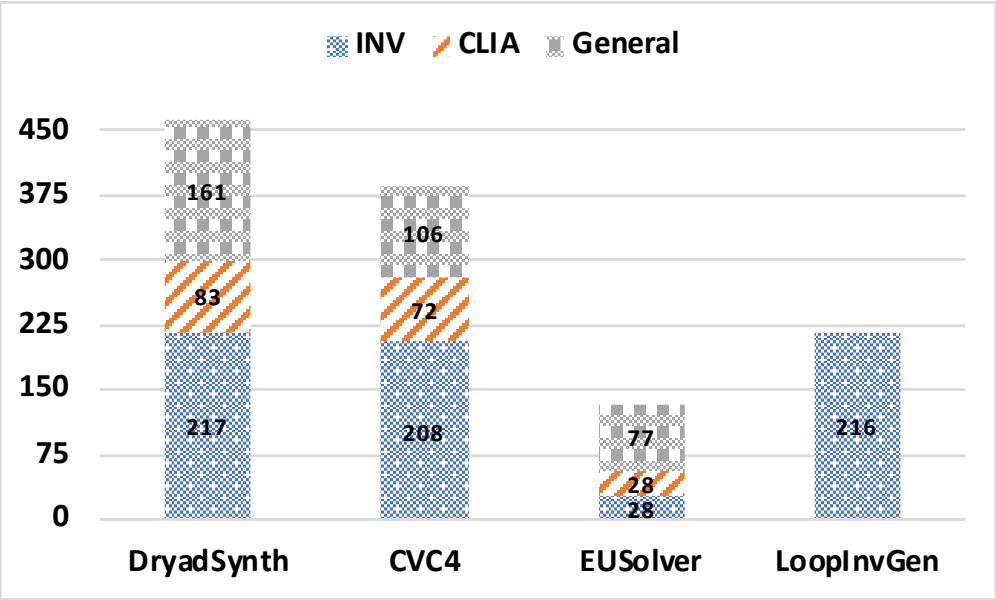


Figure 3.11. Fastest solved benchmarks (breakdown by tracks).

Ablation Studies. To evaluate whether the combination of enumerative and deductive synthesis improves the performance, we also compare DryadSynth, in which the full-fledged cooperative synthesis framework is implemented, with our implementation of the plain height-based enumeration synthesis algorithm (Algorithm 4), the plain deductive synthesis algorithm (Algorithm 5), and our cooperative synthesis framework with the height-based enumeration synthesis algorithm replaced by EUSolver, a representative enumerative synthesizer.

Figure 3.14 compares the solving time of the full-fledged cooperative synthesis framework and the plain height-based enumeration synthesis algorithm on all benchmarks. As the figure illustrates, with the help of divide-and-conquer deduction, our cooperative synthesis clearly outperformed the plain height-based enumeration for the vast majority of all benchmarks. The plain height-based enumeration procedure performed slightly better for several easier-to-solve problems, though, as they are simple enough and divide-and-conquer cannot help much.

Figure 3.15 shows the number of solved benchmarks by the plain deductive synthesis algorithm (per category) and the number of extra benchmarks solved with the help of the

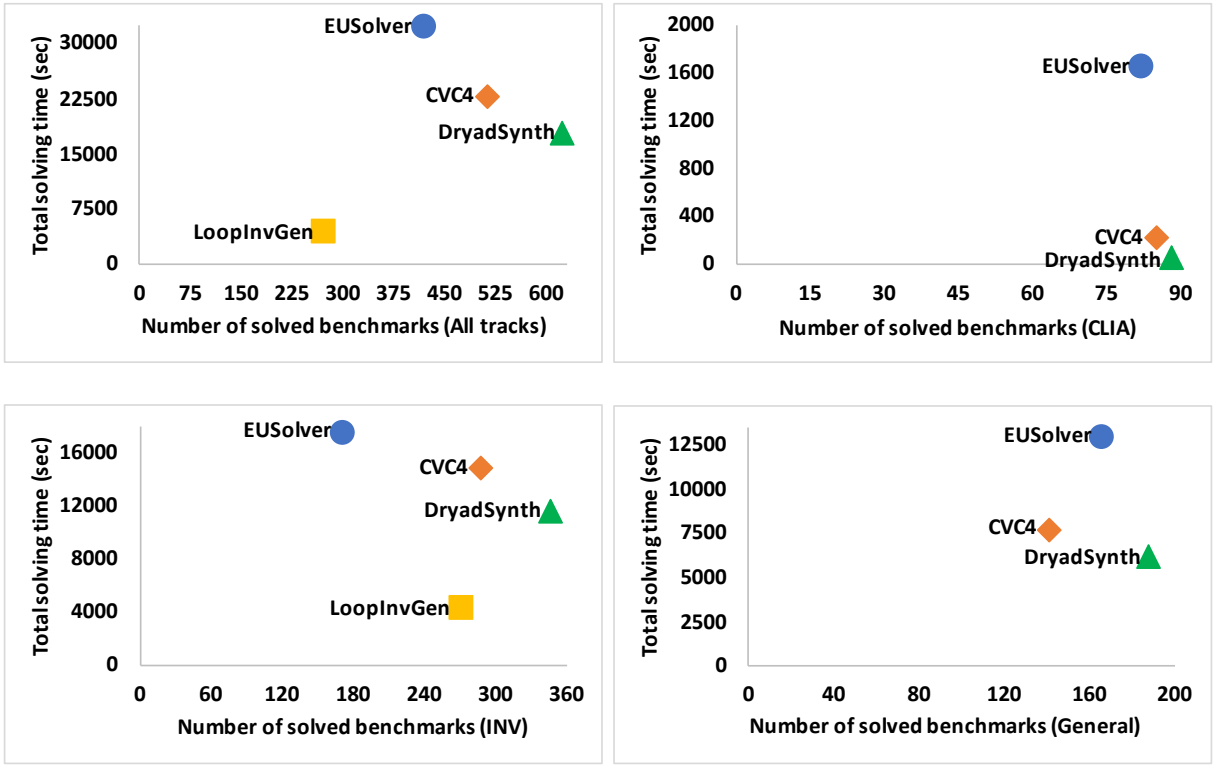


Figure 3.12. Comparison of solvers on total solved benchmarks and total solving time.

height-based enumeration (per category). Figure 3.15 shows that among all the benchmarks solved by the cooperative synthesis framework, only 32.6% of them were solved by pure divide-and-conquer deduction. The vast majority of all benchmarks were further solved with the help of height-based enumeration.

Figure 3.16 compares the amounts of time spent for benchmarks between the vanilla DryadSynth and a version using EUSolver as the enumerative synthesis component. In the EUSolver-backed DryadSynth, every invocation to the fixed-height synthesis algorithm (Algorithm 4) is replaced with a query to EUSolver. As we could not find a proper way to control the search space when invoking EUSolver, the query to EUSolver searches with unbounded height. Therefore, while our height-based enumeration was parallelized, it became ineffectual to parallelize EUSolver. We omitted those benchmarks purely solved by the deductive synthesis algorithm. That wound up with a comparison on 496 benchmarks. The

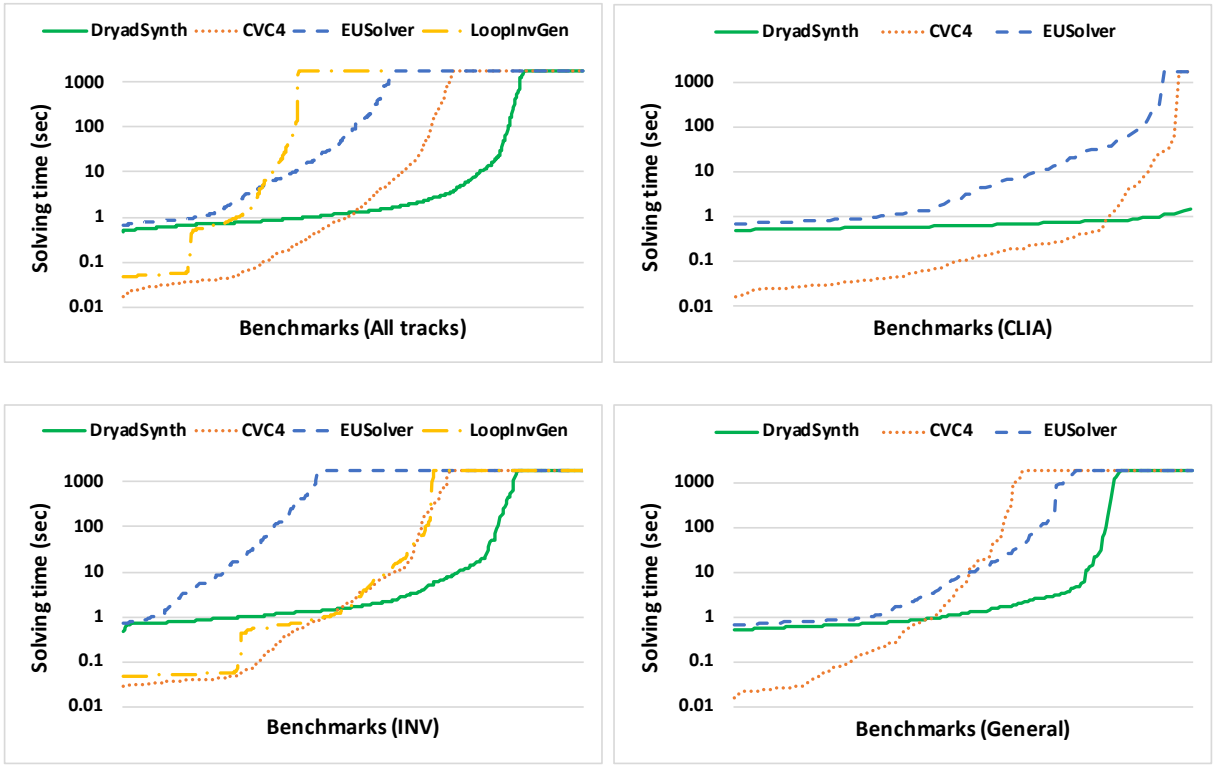


Figure 3.13. Solving time per benchmark in increasing order.

figure indicates that the vanilla DryadSynth consistently performed better and solved 135 more benchmarks than the EUSolver-backed DryadSynth.

In short, our studies show that the cooperation of our enumerative and deductive algorithms improves the performance from the standalone enumeration, and solves more benchmarks than the standalone deduction. Our height-based enumeration algorithm also performs better than an existing enumeration algorithm when serving as the enumerative synthesis component.

3.8 Related Work

Syntax-Guided Synthesis. As we mentioned in Section 3.1, the most important dimension along which we characterize existing syntax-guided synthesis approaches is their synthesis strategies. For example, among the winning SyGuS synthesizers we compared with, EUSolver [4] adopts a purely enumerative search strategy, and CVC4 [97] solves synthesis

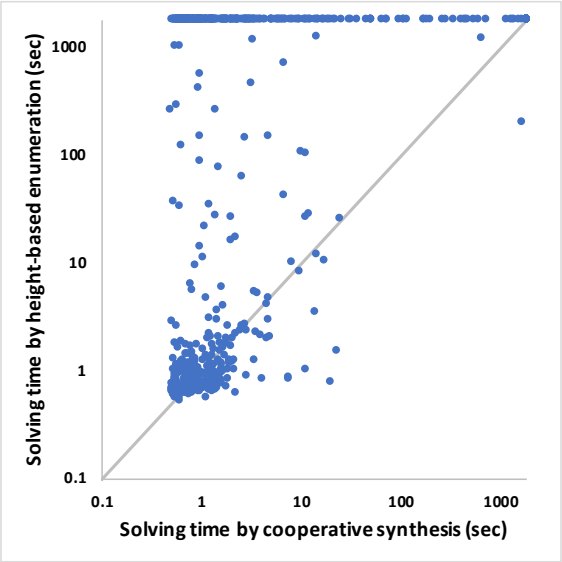


Figure 3.14. Cooperative synthesis vs. Plain height-based enumeration.

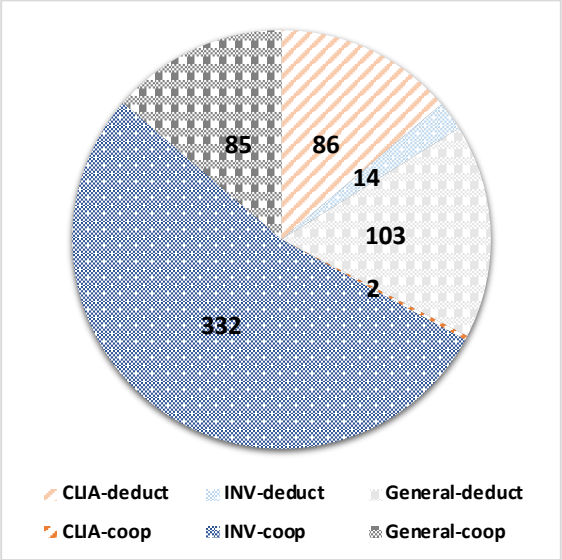


Figure 3.15. Cooperative synthesis vs. Plain deduction.

problems purely symbolically through a procedure called counterexample guided quantifier instantiation. The Counterexample-Guided Inductive Synthesis (CEGIS) framework [90] has been a common theme underlying several solvers which differ in how the synthesizer generates candidates from counterexamples: Sketch [33], [76] solves constraints that encode the counterexamples; Alchemist [103] and ICE-DT [104] find likely candidates using learning algorithms.

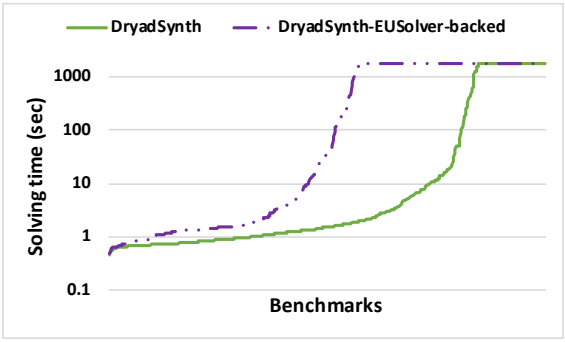


Figure 3.16. Vanilla DryadSynth vs. EUSolver-backed DryadSynth.

The decision tree representation for fixed-height synthesis (Sec 3.5.2) is similar to the representations used in the ICE-DT learning algorithm [104] and the enumerative search algorithm underlying EUSolver [4]. Our decision tree for *CLIA* (e.g., Figure 3.6) is different: it assumes a fixed height for the decision tree and represents the whole function as a vector of coefficients, and helps us encode the fixed-height synthesis problem to a *CLIA* query.

A lot of general search-based algorithms for syntax-guided synthesis have been developed in the past few years, including learning-based ones [72], [103], [104] and enumerative ones [105], [106]. Caulfield *et al.* [107] identified several decidable fragments of synthesis problems in the theories of *EUF* and *BitVector*. The SyGuS solver Euphony [101] predicts the *likelihood* of candidate programs, and enumerates them starting from the most likely correct candidates. Their algorithm focuses on theories of *String* and *BitVector*.

Combining Concrete and Symbolic Search. The idea of height-based enumeration is inspired by a recent trend of combining concrete and symbolic search, but differs from existing approaches in the target program and/or the enumeration method.

The work of Gulwani *et al.* [108] is, to the best of our knowledge, the first attempt of combining enumerative and symbolic search. Their system enumerates the number of components and encodes each case as a symbolic constraint. The Adaptive Concretization algorithm [84], [85] developed for Sketch is another instance of enumerative-symbolic combination. As a Sketch-based algorithm, adaptive concretization supports a general class of SyGuS problems. Its algorithm statistically determines a class of *highly influential unknowns* and explicitly enumerates all possible values of these unknowns. Unlike our decision-

tree-based enumeration, their enumeration strategy is not supported by integer arithmetic decision procedures and seems not competitive in synthesizing *CLIA* functions [82]. SyncQuid [86] synthesizes recursive functional programs using an algorithm that enumerates the top part of the program and synthesizes the remaining part of the program through liquid abduction. Hades [87] is a system that synthesizes transformations on hierarchical data trees. A key component of Hades is an algorithm for synthesizing path transformations from examples, which enumerates all possible partitions of the examples, checks the unifiability of each partitioned set using SMT solvers, and combines the unifiers into a decision tree through machine learning.

Our height-based enumeration is different: the shapes (or sketches) of the syntax tree are not explicitly enumerated or learned, but grouped by their heights and then enumerated and solved symbolically. This is a nice combination as it still guarantees to synthesize the smallest satisfying program while leveraging more power of symbolic solving. To the best of our knowledge, this idea is not explored by any existing techniques.

Deductive Synthesis. There has been significant research effort on deductive synthesis. Spiral [5] is an automatic system that synthesizes digital signal processing algorithms and programs. Paraglide [7] derives algorithms for concurrent programs from their sequential implementations using domain specific knowledge to constraint the search space. Fiat [6] also utilizes deductive synthesis to synthesize abstract data types that package methods with private data. As an interactive system, the synthesizer also requires user's guidance to help with the synthesis. Recently, Polikarpova *et al.* [96] present SusLik as a deductive-based synthesizer that generates imperative heap-manipulating programs. The synthesizer takes a pair of pre- and post- conditions written in separation logic as input and derives the programs based on a set of deductive rules with structural constraints of the heap baked in. Above deductive synthesizers are all designed to serve a specific application and seem hard to extend to other synthesis purposes, as domain specific knowledge is critical for these systems. Our approach is more general and not limited to a fixed grammar.

Combining Deduction and Enumeration. We are not the first to combine deduction and enumeration. $\lambda 2$ [98] and FleshMeta [99] have used deduction in novel ways (inverse semantics or refutation) to decompose the synthesis task and guide the program search.

They can be perceived as special type-directed search algorithms [81] and comparable to other search strategies. More recently, MORPHEUS [88] also combines deduction and enumeration to synthesize programs manipulating tabular data. MORPHEUS uses enumerative search to find possible candidate programs and uses deduction to prune the search space. With similar ideas, NEO [100] synthesizes programs in several domains, including tabular data transformation and list manipulation, by supplying a DSL of the target domain. Our deduction-enumeration combination is different from the techniques above: it repeatedly performs divide-and-conquer and solves subproblems by deduction or enumeration separately.

Li *et al.* [109] present a technique for searching proofs for program correctness. They use abductive inference to decompose the verification task to several lemmas then discharge each lemma separately. Our idea of divide-and-conquer deduction is similar to their lemma abductive inference. The difference from the prior technique is that we focus on syntax-guided program synthesis and design our own deductive rules that are general enough for arbitrary grammars.

3.9 Conclusion

We introduced cooperative synthesis, a syntax-guided synthesis technique in which enumerative and deductive synthesis strategies are combined for solving SYGUS problems with *CLIA* background. The framework repeatedly splits large synthesis problems to smaller subproblems and have them solved by a deductive synthesis engine or a height-based enumeration algorithm. Then the solutions are combined to form a final solution. We found that, compared to state-of-the-art synthesizers, cooperative synthesis has better scalability and solved many benchmarks not possible before. This synthesis technique may be extended to handle other background theories in the future.

4. REASONING ABOUT RECURSIVE TREE TRAVERSALS

In this chapter, towards provably-correct optimizations over tree traversals, we present a fined-grained representation for iterations in tree traversals and an encoding to Monadic Second-Order logic over trees.

4.1 Introduction

Trees are one of the most widely used data structures in computer programming and data representations. Traversal is a common means of manipulating tree data structures for various systems, as diverse as syntax trees for compilers [110], k-d trees for scientific simulation [111]–[114], and DOM trees for web browsers [115]. Due to dependence and locality reasons, these traversals may iterate over the tree in many different orders: pre-order, post-order, in-order or more complex, and in parallel for disjoint regions of the tree. A tree traversal can be regarded as a sequence of *iterations* (each executing a code block on a tree node) [1] and many transformations essentially tweak the order of iterations for better performance or code quality, with the hope that no dependence between iterations is violated.

Matching this wide variety of applications, orders, and transformations, there has been a fragmentation of mechanisms that represent and analyze tree traversal programs, each making different assumptions and tackling a different class of traversals and transformations, using a different formalism. For instance, Meyerovich *et al.* [115], [116] use attribute grammars to represent webpage rendering passes and automatically compose/parallelize them, but the traversals representable and fusible are limited, as the dependence analysis is coarse-grained at the attribute level. TreeFuser [117] uses a general imperative language to represent traversals, but the dependence graph it can build is similarly coarse-grained. In contrast, the recently developed PolyRec [118] framework supports precise instance-wise analyses for tree traversals, but the underlying transducer representation limits the traversals they can handle to a class called *perfectly nested programs*, which excludes mutual recursion and tree mutation. All these mechanisms are ad hoc and incompatible, making it impossible to

[1] ↑We call it an iteration because it is akin to a loop iteration in a loop.

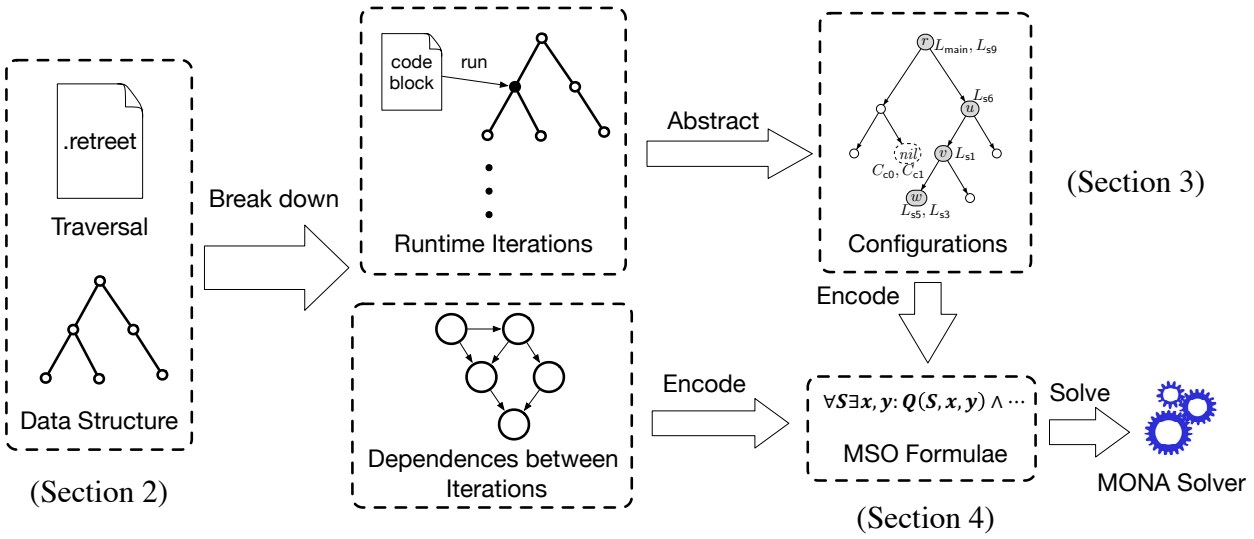


Figure 4.1. RETREET reasoning framework

represent more complex traversals or combine heterogeneous code transformations. For instance, a simple, mutually recursive tree traversal is already beyond the scope of all existing approaches (see our running example later).

Therefore, toward automated reasoning about tree traversals arising from emerging computing applications, we believe that there are two fundamental research questions. First, how to generally represent tree traversals and analyze the dependences between iterations? An expressive language in which one can freely write and combine complex tree traversals is a precursor of handling many new applications. Second, how to automatically verify the validity of subtle transformations between tree traversals? From the perspective of static analyses, the key challenge is how to design an appropriate abstraction of the program such that it is as precise as possible yet amenable for automated reasoning. Our answers to these questions are RETREET, a general framework (as illustrated in Figure 4.1) in which one can write almost arbitrary tree traversals, reason about dependences between iterations of fine granularity, and check correctness of transformations automatically. This framework features an abstract yet detailed characterization of iterations, schedules and dependences, which we call *Configuration*, as well as a powerful reasoning algorithm.

In this paper, we first present RETREET ("REcursive TREE Traversal") as an expressive intermediate language that allows the user to flexibly describe tree traversals in a recursive fashion (Section 4.2). Remarkably, RETREET can express mutually recursive traversals, which cannot be handled by existing techniques. Second, we propose *Configuration* as a detailed, stack-based abstraction for dynamic instances in a traversal (Section 4.3). Intuitively, a configuration profiles the call stack maintained during the execution; it preserves the full computation history except for function calls, i.e., recursive calls become abstract and may return arbitrary values. Furthermore, this abstraction can be encoded to Monadic Second-Order (MSO) logic over trees, which allows us to reason about dependences and check data-race-freeness and equivalence of RETREET programs (Section 4.4). The encoded formulae can be checked using MSO-based decision procedures such as the one implemented in MONA [119]. Our framework is sound and incomplete. In other words, all verified programs are indeed data-race-free/equivalent, but there is no guarantee that all data-race-free/equivalent programs can be verified. Therefore, finally, we show our framework is practically useful by synthesizing or verifying provably-correct optimizations for four different classes of programs, including real-world applications such as CSS minification and Cycletree routing, for the first time. One of these case studies also shows how RETREET is integrated with other MSO-based analysis techniques to verify list-traversal transformations that cannot be handled by RETREET alone (Section 4.5).

4.2 A Tree Traversal Language

In this section, we present RETREET, our imperative, general tree traversal language. While RETREET is syntactically simple and not intended to serve as an end-user programming language, we envision RETREET as an intermediate language for automatic analyses and many language features commonly used in practice should be translated to RETREET through a preprocessor. See more discussion in Section 4.2.1.

RETREET programs execute on a tree-shaped heap which consists of a set of locations. Each location, also called node, is the root of a (sub)tree and associated with a set *dir* of

$dir \in \mathsf{Loc}$ Fields $\quad v \in \mathsf{Int}$ Vars $\quad n \in \mathsf{Loc}$ Vars
$f \in \mathsf{Int}$ Fields $\qquad\qquad g$: Function IDs

$$
\begin{array}{rcl}
\mathit{LExpr} & ::= & n \mid \mathit{LExpr}.\mathit{dir} \\
\mathit{AExpr} & ::= & 0 \mid 1 \mid n.f \mid v \mid \mathit{AExpr} + \mathit{AExpr} \mid \mathit{AExpr} - \mathit{AExpr} \\
\mathit{BExpr} & ::= & \mathit{LExpr} == \mathsf{nil} \mid \mathsf{true} \mid \mathit{AExpr} > 0 \mid \,!\, \mathit{BExpr} \\
 & & \mid \mathit{BExpr} \;\&\&\; \mathit{BExpr} \\
\mathit{Assgn} & ::= & n.f = \mathit{AExpr} \mid v = \mathit{AExpr} \\
\mathit{Block} & ::= & \bar{v} = g(\mathit{LExpr}, \overline{\mathit{AExpr}}) \mid \mathit{Assgn}^{+} \\
\mathit{Stmt} & ::= & \mathit{Block} \mid \mathsf{if}\ (\mathit{BExpr})\ \mathit{Stmt}\ \mathsf{else}\ \mathit{Stmt} \mid \mathit{Stmt}\ ;\ \mathit{Stmt} \\
 & & \mid \{\mathit{Stmt} \parallel \mathit{Stmt}\} \\
\mathit{Func} & ::= & g(n, \bar{v})\{\ \mathit{Stmt}\ \} \\
\mathit{Prog} & ::= & \mathit{Func}^{+}
\end{array}
$$

Figure 4.2. Syntax of RETREET

pointer fields and a set f of local fields. Pointer fields dir contain the references to the children of the original location; local fields f store the local Int values.

The syntax of RETREET is shown in Figure 4.2. A program consists of a set of functions; each has a single Loc parameter and optionally, a vector of Int parameters. We assume every program has a Main function as the entry point of the program. The body of a function comprises *Block*s of code combined using conditionals, sequentials and parallelizations.

A block of code is either a function call or a straight-line sequence of assignments. A function call takes as input a *LExpr* which can be the current Loc parameter or any of its descendants, and a sequence of *AExpr*'s of length as expected. Each *AExpr* is an integer expression combining Int parameters and local fields of the Loc parameter. Non-call assignments compute values of *AExpr*'s and assign them to Int parameters, fields or special return variables. Note that the functions in RETREET can be mutually recursive, i.e., two or more functions call each other. However, there is a special syntactic restriction: every function $g(n, \bar{v})$ should not call, directly or indirectly through inlining, itself, i.e., $g(n, \dots)$ with arbitrary Int arguments (see more discussion below).

The semantics of RETREET is common as expected and we omit the formal definition. In particular, all function parameters are call-by-value; the parallel execution adopts the statement-level interleaving semantics (every execution is a serialized interleaving of atomic statements).

```
Odd(n)
  if (n == nil) return 0
  else return Even(n.l) + Even(n.r) + 1
Even(n)
  if (n == nil) return 0
  else return Odd(n.l) + Odd(n.r)
Main(n)
  { o = Odd(n) || e = Even(n) }
  return (o, e)
```

Figure 4.3. Mutually recursive traversals (original)

Example 4.2.1. Figure 4.3 illustrates our running example, which is a pair of mutually recursive tree traversals. Odd(n) and Even(n) count the number of nodes at the odd and even layers of the tree n, respectively (n is at layer 1, n.l is at layer 2, and so forth). Odd and Even recursively call each other; and the Main function runs Odd and Even in parallel, and returns the two computed numbers. Note that the mutual recursion is beyond the capability of all existing automatic frameworks that handle tree traversals [115]–[118], [120], [121].

4.2.1 Discussion of the Language Design

We remark about some critical design features of RETREET. Served as an intermediate language for analyses, RETREET is semantically expressive but syntactically simple. In a nutshell, RETREET has been carefully designed to be *maximally permissible* of describing tree traversals, yet *encodable* to the MSO logic.

Key Language Restrictions

Three major design features make possible our MSO encoding presented in Section 4.4: *obviously terminating*, *single node traversal* and *no-tree-mutation*. Despite these restrictions, RETREET is still more general and more expressive than the state of the art—to the best of our knowledge, all the restrictions we discuss below can be seen in all existing approaches (find more discussion in Section 4.6).

Termination: RETREET allows *obviously terminating* tree traversals. Any function $g(n, \bar{v})$ should not contain recursive calls to $g(n, \dots)$, regardless of directly in *Stmt* or indirectly through inlining arbitrarily many calls in *Stmt*. The restriction guarantees not only the termination, but also a bound of the steps of executions. With this restriction, every function call makes progress toward traversing the tree downward. Hence, the height of the call stack will be bounded by the height of the tree, and every statement [2] is executed on a node at most once. Therefore, running a RETREET program P on a tree T will terminate in $\mathcal{O}(|P|^{h(T)})$ steps where $h(T)$ is the height of the tree. This bound is critical as it allows us to encode the program execution to a tree model, with only a fixed amount of information on each node. In contrast, RETREET excludes the following program:

```
A(n, k)
  if (k <= 0) return 0
  else return A(n, k–1) + ...
```

The program terminates, but the length of execution on node n is determined by the input value k, which can be arbitrarily large and makes our tree-based encoding impossible.

Single node traversal: In RETREET, all functions take only one Loc parameter. Intuitively, this means the tree traversal is not allowed to manipulate more than one node at one time. For example, traversing two trees at the same time to find the max height is not allowed in RETREET. This is a nontrivial restriction and necessary for our MSO encoding. The insight of this restriction will be clearer in Section 4.4.

No tree mutation: Mutation to the tree topology is generally disallowed in RETREET. General tree mutations will possibly affect the tree-ness of the topology, where our tree-based encoding cannot fit in.

[2]↑Notice that two different call sites of the same function are considered two different statements. So the number of statements is bounded by the size of the program.

Other Restrictions for Simplification

As an intermediate language, RETREET has more syntactic restrictions, which are not fundamental and does not jeopardize the expressivity. In particular, RETREET does not support loops, global variables, return statements or integer arguments. These restrictions are not essential because loops or global variables can be rewritten to recursion and local variables, respectively. Return values or integer arguments can also be rewritten to local fields. As long as the rewritten program satisfies the real restrictions we set forth above, it can be handled by our framework. See our discussions below.

Loop-freeness: The RETREET language does not allow iterative loops. Recall that RETREET is meant to describe tree traversals, and the no-loop restriction guarantees that the program manipulates every node only a bounded number of times, and hence the termination of the program. That said, most typical loops or even nested loops traversing a tree only compute a limited number of steps on each node, and hence can be naturally converted to recursive functions in RETREET.

No global variables: We omit global variables in RETREET. However, it is not difficult to extend for global variables. Note that when the program is sequential, i.e., no concurrency, one can simply replace a global variable with an extra parameter for every function, which copies in and copies out the value of the global variable. In the presence of concurrency, we need to refine the current syntax to reason about the schedule of manipulations to global variables. Basically, every statement accessing a global variable forms a separate *Block*, so that we can compare the order between any two global variable operations.

No return statements and no integer arguments: We handle recursive calls with return values with the following preprocessing. For every function, we introduce a special local field with the function name (if no conflict occurs) in each node to store the return value of the function call. Each return statement can be rewritten to a writing to the special local field in the callee (the unique `Loc` argument of the call). For each recursive call to a function in the program, we ignore the return value from the call and instead read from the corresponding special local field of the callee. Recursive calls with integer arguments are handled with similar preprocessing: the caller writes to special local fields of the callee

```
Odd(n)
  if (n == nil) // c0
    n.Odd = 0 // s0
  else
    Even(n.l)   // s1
    Even(n.r)   // s2
    n.Odd = n.l.Even +
            n.r.Even + 1 // s3

  Main(n)
    { Odd(n) || // s8
      Even(n) }   // s9
    n.Main = (n.Odd, n.Even)   // s10
```

```
Even(n)
  if (n == nil)  // c1
    n.Even = 0  // s4
  else
    Odd(n.l)  // s5
    Odd(n.r)  // s6
    n.Even = n.l.Odd + n.r.Odd // s7
```

Figure 4.4. Mutually recursive traversals (no-return-value)

such that the callee can read them as integer arguments. After this preprocessing step, the running example shown in Figure 4.3 are rewritten to the one in Figure 4.4. [3]

For the simplification of presentation, in the rest of the paper, we also assume: all trees are binary with two pointer fields l and r, every function only calls itself or other functions on n.l or n.r, and returns only a single Int value (which is rewritten to a special local field), and every boolean expression is atomic, i.e., of the form $LExpr == \mathsf{nil}$ or $AExpr > 0$. Calls to other functions on n are always inlined to make all operations on fields of n explicit. In addition, we assume the program is free of null dereference, i.e., every term $le.dir$ is preceded by a guard le != nil. Note that relaxing these assumptions will not affect any result of this paper, because any RETREET program violating these assumptions can be easily rewritten to a version satisfying the assumptions.

4.2.2 Code Blocks

With assumptions made above, RETREET programs can be decomposed to code blocks, which are a key to our framework. Each code block is a function call or a sequence of straight-line, non-call assignments derived from the *Block* symbol of the grammar (see Figure 4.2). We

[3]↑Composed expressions, such as n.l.Even, are used for code readability. Accesses to the special local field are allowed when $n == nil$, since the transformed program is not executable and used for analysis only. In our MSO encoding (described in Section 4.4), $isNil$ is a special MSO predicate.

AllFuncs	the set of all functions
AllParams	the set of all Int function parameters
AllBlocks	the set of all blocks
AllCalls	the set of all blocks for function calls
AllNonCalls	the set of all blocks for straight-line non-call assignments
Blocks(f)	the set of all blocks belonging to a function f
Calls(f)	Blocks(f) $\cap$ AllCalls
Params(f)	the set of Int parameters for f
Nodes(T)	the set of all nodes in the tree T
Paths(t)	the set of all possible paths (through statement-level interleaving) to t from the entry point of the function that t belongs to

Figure 4.5. Commonly used notations

use some necessary notations for blocks, of which the meaning is determined by the syntactic structure of the program. Figure 4.5 lists common sets of functions, blocks, parameters and nodes that will be frequently used in this paper.

We also define the possible relations between blocks, as shown in Figure 4.6. Every function's body can be represented as a syntax tree whose leaves are statement blocks and non-leaf nodes are sequentials, conditionals or parallels. Then the relation between two statement blocks is determined by their positions in the syntax tree. In particular, when two blocks s $\sim$ t belong to the same function f, there are three possible relations, determined by the least common ancestor (LCA) node of s and t that is a sequential, conditional or parallel.

Example 4.2.2. In our running example (Figure 4.4), there are 11 blocks. We number the blocks with s0 through s10, as shown in the comment following each block. There are six call blocks: AllCalls = {s1, s2, s5, s6, s8, s9}; and five non-call blocks: AllNonCalls = {s0, s3, s4, s7, s10}. Take s6 for example, Path(s6) is just the path from the beginning of function Even (which s6 belongs to) to s6, i.e., from $\neg$ c1 to s5 then s6. The $\sim$ relation holds between any two blocks from the same group: s0 through s3, s4 through s7, or s8 through s10. s2 $\triangleleft$ s7 because s2 calls Even and s7 $\in$ Blocks(Even); s5 $\prec$ s7 because s5 precedes s7; s0 $\uparrow$ s1 because s0 belongs to the if-branch and s1 belongs to the else-branch; s8 $\parallel$ s9 because they are running in parallel.

LCA(s, t)	The least common ancestor (LCA) of blocks s and t in the syntax tree.
$s \triangleleft t$	s is a function call to f and $t \in$ Blocks(f).
$s \sim t$	s and t are from the same function definition, i.e., $s, t \in$ Blocks(f) for some function f.
$s \prec t$	LCA(s, t) is a sequential, i.e., s precedes t.
$s \uparrow t$	LCA(s, t) is a conditional, i.e., there is a conditional if (...) then A else B such that s and t belong to A and B, respectively.
$s \parallel t$	LCA(s, t) is a parallel, i.e., s and t can be executed in arbitrary order.

Figure 4.6. Relations between blocks

Lemma 4.2.1. For any two statement blocks s and t, $s \sim t$ if and only if exactly one of the following relations holds: $s \prec t$, $s \uparrow t$, $t \prec s$, $t \uparrow s$ or $s \parallel t$.

Read&Write analysis.

In our framework, data dependences are represented and analyzed at the block level. We perform a static analysis over the program to extract the sets of local fields and variables being accessed in each non-call block. Intuitively, we use several read sets and write sets to represent local fields and global variables being read or written, respectively, in each statement block.

For every non-call block s, we build the read set R_s by adding all data fields and local variables that occur on the RHS of an assignment. The data fields can be from the current node (such as n.v) or a neighbor node (such as n.l.v). The write set W_s can be built similarly: all data fields and local variables that occur on the LHS of an assignment are added.

4.3 Iteration Representation

As we mentioned above, code blocks (function calls or straight-line assignments) are building blocks of RETREET programs and are a key to our framework. In our running example (Figure 4.4), there are 11 blocks. Then the execution of a RETREET program is a sequence of iterations, each running a non-call code block on a tree node. For example,

consider executing our running example on a single-node u (i.e., $u.\mathsf{l} = u.\mathsf{r} = \mathsf{nil}$), one possible execution is a sequence of iterations (also called instances in the literature):

$$(\mathsf{s0}, u.l), (\mathsf{s0}, u.r), (\mathsf{s7}, u), (\mathsf{s4}, u.l), (\mathsf{s4}, u.r), (\mathsf{s3}, u)$$

Note that every iteration is unique and appears at most once in a traversal, as per the obviously terminating restriction of RETREET. However, this representation is not sufficient to reason about the dependences between steps. For example, if the middle steps $(\mathsf{s7}, u), (\mathsf{s4}, u.l)$ were swapped, is that still a possible sequence of execution? The question can't be answered unless we track back the contexts in which the two steps are executed: $(\mathsf{s7}, u)$ is executed in the call to $\mathsf{Even}(u)$ (block $\mathsf{s9}$); $(\mathsf{s4}, u.l)$ is executed in the call to $\mathsf{Even}(u.\mathsf{l})$ (block $\mathsf{s1}$), which is further in $\mathsf{Odd}(u)$ (block $\mathsf{s8}$). As the two calls are running in parallel, swapping the two steps yields another legal sequence of execution. Automating this kind of reasoning is extremely challenging. In fact, even determining if an iteration exists is already undecidable:

Theorem 4.3.1. *Determining if an iteration may occur in a* RETREET *program execution is undecidable.*

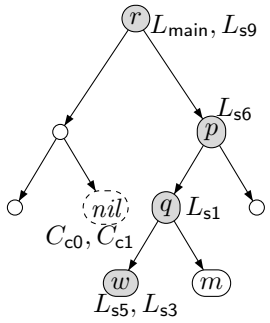
Proof. We prove the undecidability through a reduction from the halting problem of 2-counter machines [122]. We can build a RETREET program to simulate the execution of a 2-counter machine. Given a 2-counter machine M, every line of non-halt instruction c in M can be converted to a function in a RETREET program. The function is of the form $f_c(\mathsf{n}, \mathsf{v}_1, \mathsf{v}_2)$: n is a Loc parameter and $\mathsf{v}_1, \mathsf{v}_2$ are Int parameters. It treats $\mathsf{v}_1, \mathsf{v}_2$ as the current values of the two counters, updates the two counter values to $\mathsf{u}_1, \mathsf{u}_2$ by simulating the execution of c, then recursively calls $f_{c'}(\mathsf{n.l})$ if c' the next instruction. for the halt instruction, a special function f_{halt} will pass up the signal by recursive calls, and finally run a special line of code s on the root. Then M halts if and only if the iteration $(\mathsf{s}, \mathit{root})$ occurs. □

4.3.1 Configuration

As precise reasoning about RETREET is undecidable, we propose an iteration representation called *configuration*, which is a right level of abstraction for which automated reasoning

R#	Content
0	$(\mathsf{main}, r, \mathsf{s8} = 5, \dots)$
1	$(\mathsf{s9}, r, \mathsf{s5} = 3, \dots)$
2	$(\mathsf{s6}, p, \dots)$
3	$(\mathsf{s1}, q, \dots)$
4	$(\mathsf{s5}, w, \mathsf{s1} = 0, \mathsf{s2} = 0)$
5	$(\mathsf{s3}, w)$

(a) A configuration



(b) Represented as labels on the tree

Figure 4.7. Example of configuration encoding

is possible. Intuitively, a configuration looks like a snapshot of the call stack, which consists of multiple records. The last record describes the current running block as we discussed above. Each other record describes a call context which includes: the callee block, the single Loc parameter, and other Int variables' values. These Int values include: first, for each Int parameter, the context records its *initial* value received when the call begins; second, for each function call within the current call, the context uses *ghost variables* to predict the return values.

Example 4.3.1. Figure 4.7a gives an example of a configuration, which consists of 6 records. The last record (record #5) indicates that the current step is running block s3 on tree node w, and the current values of local variables. In other records, we only show the callee stack, the Loc parameter, and other relevant Int variables. For example, the value $\mathsf{s8} = 5$ means that the call in s8 is predicted to finish and return value 5, which might be relevant to the next call context, s9.

Obviously, not all stacks of records are valid configurations. In particular, the beginning record should run main and the last record should run a non-call block. More importantly, for any non-beginning record, one of the *path conditions* [4] of the block should be satisfied, i.e., this block of code can be reached from the beginning of the function it belongs to. While a precise characterization of these constraints is expensive and leads to undecidability as per

[4]↑We consider all the finitely many possible statement-level interleavings.

Theorem 4.3.1, we make two key assumptions below which make it possible for configurations to be abstractions of real call stacks in an execution:

1. all function calls not shown in the stack can return arbitrary values;
2. a call stack is valid if every pair of adjacent records in the stack are consistent.

With these two assumptions, we can now formally define configuration:

Definition 4.3.1 (Configuration)**.** *A configuration of length k on a tree T is a mapping $\mathcal{C} : [k] \to \mathsf{AllBlocks} \times \mathsf{Nodes}(T) \times (\mathsf{AllParams} \cup \mathsf{AllCalls} \rightharpoonup \mathbb{Z})$ such that:*

- *For any $0 \leq \mathrm{i} < k$, $\mathcal{C}(\mathrm{i})$ is of the form (s, u, M) where $\mathsf{s} \in \mathsf{AllCalls}$ is a call to a function f, and M is only defined on $\mathsf{Params}(f) \cup \mathsf{Blocks}(f)$.*
- *The last record $\mathcal{C}(k)$ is of the form $(\mathsf{s}, u, \emptyset)$, where $\mathsf{s} \in \mathsf{AllNonCalls}$.*
- *The first record $\mathcal{C}(0)$ is of the form $(\mathsf{main}, root_T, ...)$.*
- *For any two adjacent records $\mathcal{C}(\mathrm{i}-1) = (\mathsf{s}, u, M)$, $\mathcal{C}(\mathrm{i}) = (\mathsf{t}, v, N)$, s is a call to the function that t belongs to (denoted as $\mathsf{s} \triangleleft \mathsf{t}$, see Figure 4.6). Moreover, (s, u, M)* speculatively reaches *(t, v, N).*

The speculative reachability mentioned in the last condition of the definition above does not relate to any concrete run of a program and is a key notion that captures the second assumption we made above. In other words, we consider two adjacent records consistent if the first one can speculatively reaches the second one. We next define speculative reachability formally.

4.3.2 Speculative Reachability

Intuitively, a record (s, u, M) speculatively reaches (or just reaches for short) another record (t, v, N) if an execution triggered by (s, u, M) can lead to the next record (t, v, N). More concretely, if s is a call to a function f, then one can run f on node u, with initial integer arguments from $M|_{\mathsf{Params}(f)}$. Whenever a function call within the body of f is encountered, one just skips the call and returns the speculative output from $M|_{\mathsf{Calls}(f)}$. The execution

$$
\begin{array}{lcl}
\mathsf{wp}(n.f = AExpr,\ \varphi,\ M) & = & \varphi[AExpr/n.f] \\
\mathsf{wp}(v = AExpr,\ \varphi,\ M) & = & \varphi[AExpr/v] \\
\mathsf{wp}(\bar{v} = t(\dots),\ \varphi,\ M) & = & \varphi[M(\mathsf{s})/v] \\
\quad \text{where } \mathsf{s} \text{ is the id of the current statement} & & \\
\mathsf{wp}(\mathsf{l}\ ;\ \mathsf{l}',\ \varphi,\ M) & = & \mathsf{wp}(\mathsf{l}, \mathsf{wp}(l', \varphi))
\end{array}
$$

Figure 4.8. Weakest precondition

should lead to a run of block t on node v. If t is also a function call, the input arguments for the call should match the expected, speculative inputs from N. We call this execution process a *speculative execution*:

Definition 4.3.2 (Speculative Execution)**.** *Given a function f, a group of initial values $I : \mathsf{Params}(f) \to \mathbb{Z}$ and a group of speculative outputs $O : \mathsf{Calls}(f) \to \mathbb{Z}$, a speculative execution of f with respect to I and O follows the following steps:*

1. *initialize each parameter p with value $I(p)$, and let the current block c be the first block in f;*
2. *if c is not a call, then simulate the execution of c, and move to the next block;*
3. *if c is a call of the form $v = g(\mathsf{le}, \bar{\mathsf{ie}})$, then update the special field $\mathsf{le}.g$'s value with $O(\mathsf{c})$.*

With the formal definition above, we can formulate the speculative reachability using logical formulae. Note that the speculative execution may be nondeterministic due to the concurrency. However, there are only finitely many possible paths with statement-level interleaving and each path is of finite length. Then for each concrete path, the speculative execution of a function is completely deterministic as all initial parameters and return values from function calls are determined by M. More specifically, for every code snippet l without branching and every logical constraint φ that should be satisfied after running l, we can compute the weakest precondition $\mathsf{wp}(\mathsf{l}, \varphi, M)$ that must be satisfied before running l. The definition of wp is shown in Figure 4.8.

Now if s is a call to function g, we can determine if the speculative execution of g with respect to M hits block t. The path from the entry point of g to t will be a straight-line sequence of statements of the form

$$\mathsf{l}_1; \mathsf{assume}(\mathsf{c}_1); \ldots; \mathsf{assume}(\mathsf{c}_{n-1}); \mathsf{l}_n; \mathsf{Block\ t}$$

where every branch condition is converted to a corresponding $\mathsf{assume}(\mathsf{c}_\mathsf{i})$. Then we can compute the path condition for t by computing the weakest precondition for every condition c_i on the path:

$$WP(\mathsf{c}_\mathsf{i}, M) \equiv \mathsf{wp}(\mathsf{l}_1; \ldots; \mathsf{l}_\mathsf{i},\ \mathsf{c}_\mathsf{i},\ M)[M(\bar{p})/\bar{p}]$$

where $\bar{p}$ is the sequence of arguments for g.

Moreover, when t is another call block, we also need to make sure that the initial parameters in N match the speculative execution of the above code sequence w.r.t. M. We denote this condition as $Match_{\mathsf{s},\mathsf{t}}(u, v, M, N)$.

Lemma 4.3.1. Let (s, u, M) and (t, v, N) be two records such that $\mathsf{s} \triangleleft \mathsf{t}$ (as defined in Figure 4.6). Then (s, u, M) speculatively reaches (t, v, N) if (u, v, M, N) satisfies
$PathCond_{\mathsf{s},\mathsf{t}}(u, v, M, N) \quad \equiv$

$$Match_{\mathsf{s},\mathsf{t}}(u, v, M, N) \wedge \bigvee_{P \in \mathsf{Paths(t)}} \Big(\bigwedge_{\mathsf{c} \in P} WP(\mathsf{c}, M) \Big)$$

Examples

We present several examples to illustrate how the paths and path conditions are determined.

Example 4.3.2. Consider a code block s calling a function $\mathsf{foo(n,\ p,\ r0)\ \{\ n.f = p + 1\ ;\ r1 = r0;\ if\ (n.f < r1)\ \{...\}\ else\ \{\ foo(n.l,\ p,\ r0)\ //\ Block\ t\ \}\ \}}$. For record (s, u, M) to reach record (t, v, N), there is only one path on which there is one condition, $\mathsf{n.f} < \mathsf{r1}$, which occurs negatively. In other words, the code sequence reaching t is $\mathsf{n.f = p + 1\ ;\ r1 = r0;\ assume\ (n.f \geq r1);\ Block\ t}$. In addition, since code blocks s and t invoke function foo on nodes n

and n.l, respectively, $Match(u, v, M, N)$ should ensure that v is the left child of u, i.e. in this case, $Match(u, v, M, N) \equiv u.\mathsf{l} = v$. Therefore the path condition can be represented as

$$PathCond_{\mathsf{s,t}}(u, v, M, N) \equiv M(\mathsf{p}) + 1 \geq M(\mathsf{r0}) \wedge u.\mathsf{l} = v$$

Example 4.3.3. This example illustrates how paths are determined in the presence of concurrency. Consider a function foo(n) { v = 0; if (v == 1) { foo(n.l) // Block t; } || v = 1; } in which the recursive call to foo(n.l) is parallel to the assignment v = 1. Since every possible statement-level interleaving is considered, weakest preconditions for all the three possible paths are computed: 1) v=0; v=1; assume v==1; foo(n.l); 2) v=0; assume v==1; v=1; foo(n.l); 3) v=0; assume v==1; foo(n.l); v=1;. The recursive call foo(n.l) is reachable in the first possible path.

Example 4.3.4. This example illustrates that non-recursive calls can be precisely handled without any speculation. Consider the code snippet foo(n) { n.f = 0; bar(n); if (n.f == 1) { foo(n.l) // Block t } } bar(n) { n.f = 1; } where function bar is indeed a single assignment manipulating the local field f of n. As we mentioned in Section 4.2.1, during preprocessing of function foo, calls to other functions on n are always inlined to make all operations on fields of n explicit. Function call to bar(n) in foo(n) will be inlined to n.f = 1, thus the recursive call to foo(n.l) is obviously reachable.

4.4 Encoding to Monadic Second-Order Logic

The configuration-based abstraction described above allows us to encode the schedules and dependences between configurations to Monadic Second-Order (MSO) logic over trees, a well known decidable logic. Furthermore, some common dependence analysis queries can be checked by checking MSO formulae. We show the encoding in this section. The syntax of the logic contains a unique *root*, two basic operators *left* and *right*. There is a binary predicate *reach* as the transitive closure of *left* and *right*, and a special *isNil* predicate with constraint $\forall v.\Big(isNil(v) \rightarrow isNil(left(v)) \wedge isNil(right(v))\Big)$.

4.4.1 Configurations, Schedules and Dependences

First of all, we need to encode configurations we presented in Section 4.3. Given a RETREET program, we define the following labels (each of which is a second-order variable):

- for each code block s, introduce a label (a second-order variable) L_{s} such that $L_{\mathsf{s}}(u)$ denotes that there exists a record $(\mathsf{s}, u, \dots)$ in the configuration;
- for each branch condition c, introduce a label C_{c} such that $C_{\mathsf{c}}(u)$ denotes that $WP(\mathsf{c}, M)$ is satisfied by a record of the form (s, u, M);
- for each pair of blocks s and t such that $\mathsf{s} \triangleleft \mathsf{t}$, introduce a label $K_{\mathsf{s},\mathsf{t}}$ such that $K_{\mathsf{s},\mathsf{t}}(u, v)$ denotes that $Match_{\mathsf{s},\mathsf{t}}(u, v, M, N)$ is satisfied by records (s, u, M) and (t, v, N).

Note that these labels allow us to build an MSO predicate $\overline{PathCond_{\mathsf{s},\mathsf{t}}}$ as an abstracted version of the path condition $PathCond_{\mathsf{s},\mathsf{t}}$ defined in Lemma 4.3.1:

$$\overline{PathCond_{\mathsf{s},\mathsf{t}}}(u, v) \quad \equiv \quad K_{\mathsf{s},\mathsf{t}}(u, v) \wedge \bigvee_{P \in \mathsf{Paths(t)}} \Big(\bigwedge_{\mathsf{c} \in P} C_{\mathsf{c}}(u) \Big)$$

Example 4.4.1. The configuration in Figure 4.7a can be encoded to labels on the tree in Figure 4.7b. Note that the labels C_{c0} and C_{c1} are labeled on *nil* nodes only. If a node has a particular label, the node belongs to the set represented by the corresponding second-order variable. For example, node u is in L_{s6} but nodes r, v and w are not.

As the set of blocks and the set of conditions are fixed and known, we can simply represent these second-order variables using labeling predicates $\mathcal{L} \subseteq \mathsf{AllCalls} \cup \mathsf{AllNonCalls} \times \mathsf{Nodes}(T)$ and $\mathcal{C} \subseteq \mathsf{AllConds} \times \mathsf{Nodes}(T)$ such that $\mathcal{L}(\mathsf{s}, u)$ if and only if $L_{\mathsf{s}}(u)$, $\mathcal{C}(\mathsf{c}, u)$ if and only if $C_{\mathsf{c}}(u)$. In other words, $\mathcal{L}(\mathsf{s}, u)$ is the syntactic sugar for $L_{\mathsf{s}}(u)$ and $\mathcal{C}(\mathsf{c}, u)$ is the syntactic sugar for $C_{\mathsf{c}}(u)$.

Now we are ready to encode configurations to MSO. We define a formula $Configuration(\mathcal{L}, \mathcal{C}, \mathsf{q}, v)$ below, which means $\mathcal{L}$ and $\mathcal{C}$ correctly represent a configuration with $(\mathsf{q}, v, \dots)$ as the current record, for some non-call block q:

$$
\begin{aligned}
&\mathit{Configuration}(\mathcal{L}, \mathcal{C}, \mathsf{q}, v) \equiv \mathcal{L}(\mathsf{main}, \mathit{root}) \\
&\quad \wedge\ \mathit{Current}(\mathcal{L}, \mathsf{q}, v) \wedge\ \forall u. \Big(u \neq v \rightarrow \bigwedge_{\mathsf{s} \in \mathsf{AllNonCalls}} \neg \mathcal{L}(\mathsf{s}, u) \Big) \\
&\quad \wedge\ \forall u. \bigwedge_{\mathsf{s} \in \mathsf{AllCalls}} \Big(\mathcal{L}(\mathsf{s}, u) \rightarrow \bigvee_{\mathsf{s} \lhd \mathsf{t}} \Big(\mathit{Next}(\mathcal{L}, \mathcal{C}, u, \mathsf{s}, \mathsf{t}) \\
&\qquad \wedge \bigwedge_{\mathsf{t} \sim \mathsf{t'}, \mathsf{t} \neq \mathsf{t'}} \neg \mathit{Next}(\mathcal{L}, \mathcal{C}, u, \mathsf{s}, \mathsf{t'}) \Big) \Big) \\
&\quad \wedge\ \forall u. \bigwedge_{\mathsf{t} \in \mathsf{AllCalls} \cup \mathsf{AllNonCalls}} \Big(\mathcal{L}(\mathsf{t}, u) \rightarrow \mathit{Prev}(\mathcal{L}, \mathcal{C}, u, \mathsf{t}) \Big) \\
&\quad \wedge\ \forall u. \bigvee_{\mathsf{C} \in \mathsf{ConsistentCondSet}} \Big(\bigwedge_{\mathsf{c} \in \mathsf{C}} \mathcal{C}(\mathsf{c}, u) \wedge \bigwedge_{\mathsf{c} \notin \mathsf{C}} \neg \mathcal{C}(\mathsf{c}, u) \Big)
\end{aligned}
$$

The first three lines claim that main is marked on the *root*, and q is the only non-call block marked on the tree, where $\mathit{Current}(\mathcal{L}, \mathsf{q}, v)$ is a subformula indicating that for the current node v, a record $(\mathsf{q}, v, \dots)$ is in the stack for exactly one non-call block q:

$$
\mathit{Current}(\mathcal{L}, \mathsf{q}, v) \equiv \mathcal{L}(\mathsf{q}, v) \wedge \bigwedge_{\mathsf{q'} \in \mathsf{AllNonCalls}, \mathsf{q'} \neq \mathsf{q}} \neg \mathcal{L}(\mathsf{q'}, v)
$$

The next two lines, intuitively, say that every record has a unique successor (and predecessor) that can reach to (and from). Predicates *Next* and *Prev* are defined as below:

$$
\mathit{Next}(\mathcal{L}, \mathcal{C}, u, \mathsf{s}, \mathsf{t}) \equiv \exists v. \Big(\mathcal{L}(\mathsf{t}, v) \wedge \overline{\mathit{PathCond}_{\mathsf{s},\mathsf{t}}}(u, v) \Big)
$$

$$
\begin{aligned}
\mathit{Prev}(\mathcal{L}, \mathcal{C}, u, \mathsf{t}) \equiv \exists v. \Bigg(\bigvee_{\mathsf{s} \lhd \mathsf{t}} \Big(& \mathcal{L}(\mathsf{s}, v) \wedge \overline{\mathit{PathCond}_{\mathsf{s},\mathsf{t}}}(v, u) \\
& \wedge \bigwedge_{\mathsf{s'} \lhd \mathsf{t}, s' \neq s} \neg \Big(\mathcal{L}(\mathsf{s'}, v) \wedge \overline{\mathit{PathCond}_{\mathsf{s},\mathsf{t}}}(v, u) \Big) \Big) \Bigg)
\end{aligned}
$$

$\mathit{Next}(\mathcal{L}, \mathcal{C}, u, \mathsf{s}, \mathsf{t})$ indicates that a record $(\mathsf{t}, v, \dots)$ exists and is reachable from record $(\mathsf{s}, u, \dots)$. $\mathit{Prev}(\mathcal{L}, \mathcal{C}, u, \mathsf{t})$ constrains that, for a record $(\mathsf{t}, u, \dots)$, there should exist one and only one record $(\mathsf{s}, v, \dots)$ that can reach $(\mathsf{t}, u, \dots)$.

The last line makes sure that for each node u, the set of satisfied conditions C is consistent, i.e., $\bigwedge_{\mathsf{c} \in \mathsf{C}} \mathit{WP}(\mathsf{c}, M)$ is satisfiable for every record (s, u, M). In other words, a consistent condition set for a node u represents a feasible conditional path from the root of the tree to reach node u. Notice that this is a linear integer arithmetic constraint and SMT-solvable. Hence

(a) A configuration running s3 on m

(b) A configuration running s7 on q

Figure 4.9. Examples of configuration

we can assume the set of all possible consistent condition set, denoted by ConsistentCondSet, has been computed a priori.

Example 4.4.2. Let us continue on Figure 4.7b. The labels on the tree show a valid instance of configuration for the running example in Figure 4.4. The root node r belongs to the second order variable L_{main}. Block s3 running on node w is the only non-call block marked on the tree and node w is the only node that is running a non-call block. Along with the execution path ($r \to p \to q \to w$), each record has a unique successor and predecessor. For example, node w labeled L_{s5} is the only successor of label L_{s1} running on node q and $\mathsf{s1} \lhd \mathsf{s5}$. In contrast, if the label L_{s5} on w is changed to L_{s2}, the whole model is no longer a configuration because s1 does not call s2 directly (hence the third line of the formula is violated).

4.4.2 Schedules and Dependences

The definition and encoding of configurations above have paved the way for reasoning about RETREET programs. Given two configurations, a basic query one would like to make is about their order in a possible execution: can the two configurations possibly coexist? If so, are they always ordered? Or can they occur in arbitrary order due to the parallelization between them? To answer these questions, intuitively, we need to pairwisely compare the

$$Ordered(\mathcal{L}_1, \mathcal{L}_2, \mathcal{C}_1, \mathcal{C}_2) \equiv \bigvee_{\substack{\mathsf{s},\mathsf{t}_1,\mathsf{t}_2 \\ \mathsf{s} \lhd \mathsf{t}_1, \mathsf{s} \lhd \mathsf{t}_2, \mathsf{t}_1 \prec \mathsf{t}_2}} Consistent_{\mathsf{s},\mathsf{t}_1,\mathsf{t}_2}(\mathcal{L}_1, \mathcal{L}_2, \mathcal{C}_1, \mathcal{C}_2)$$

$$Parallel(\mathcal{L}_1, \mathcal{L}_2, \mathcal{C}_1, \mathcal{C}_2) \equiv \bigvee_{\substack{\mathsf{s},\mathsf{t}_1,\mathsf{t}_2 \\ \mathsf{s} \lhd \mathsf{t}_1, \mathsf{s} \lhd \mathsf{t}_2, \mathsf{t}_1 \| \mathsf{t}_2}} Consistent_{\mathsf{s},\mathsf{t}_1,\mathsf{t}_2}(\mathcal{L}_1, \mathcal{L}_2, \mathcal{C}_1, \mathcal{C}_2)$$

Figure 4.10. Relations between consistent configurations

records in the two configurations from the beginning and find the place where they diverge. We define the following predicate:

$$\begin{aligned} Consistent_{\mathsf{s},\mathsf{t}_1,\mathsf{t}_2}(\mathcal{L}_1, \mathcal{L}_2, \mathcal{C}_1, \mathcal{C}_2) \equiv \exists z. \Big[& \\ \forall v. \Big(reach(v, z) \rightarrow \Big(\bigwedge_{\mathsf{s}} \big(\mathcal{L}_1(\mathsf{s}, v) \leftrightarrow \mathcal{L}_2(\mathsf{s}, v) \big) \wedge \bigwedge_{\mathsf{c}} \big(\mathcal{C}_1(\mathsf{c}, v) \leftrightarrow \mathcal{C}_2(\mathsf{c}, v) \big) \Big) \Big) & \\ \wedge \mathcal{L}_1(\mathsf{s}, z) \wedge \mathcal{L}_2(\mathsf{s}, z) \wedge Next(\mathcal{L}_1, \mathcal{C}_1, z, \mathsf{s}, \mathsf{t}_1) \wedge Next(\mathcal{L}_2, \mathcal{C}_2, z, \mathsf{s}, \mathsf{t}_2) \Big] & \end{aligned}$$

The predicate assumes that there are two sequences of records represented as $(\mathcal{L}_1, \mathcal{C}_1)$ and $(\mathcal{L}_2, \mathcal{C}_2)$, respectively, and indicates that there is a diverging record $(\mathsf{s}, z, \dots)$ in both sequences such that: 1) the two configurations match on all records prior to the diverging record; 2) the next records after the diverging one are $(\mathsf{t}_1, \dots)$ and $(\mathsf{t}_2, \dots)$, respectively, and they can be reached at the same time (i.e., $\mathcal{C}_1$ and $\mathcal{C}_2$ agree on the diverging node z).

Blocks t_1 and t_2 are obviously in the same function. If $\mathsf{t}_1 \neq \mathsf{t}_2$, there are two possible relations between them: a) if t_1 precedes t_2 (or symmetrically, t_2 precedes t_1), then configuration $(\mathcal{L}_1, \mathcal{C}_1)$ always precedes $(\mathcal{L}_2, \mathcal{C}_2)$ (or vice versa); b) otherwise, t_1 and t_2 must be two parallel blocks, then the two configurations occur in arbitrary order. Both relations can be described in MSO (see Figure 4.10).

Example 4.4.3. Let the configuration shown in Figure 4.7b be denoted as $(\mathcal{L}_3, \mathcal{C}_3)$. Consider another configuration $(\mathcal{L}_4, \mathcal{C}_4)$ shown in Figure 4.9a with execution path $r \rightarrow p \rightarrow q \rightarrow m$. Instead of labeling L_{s5} and L_{s3} on node w, L_{s6} and L_{s3} is labeled on node m. All the other labels on nodes r, p, q in $\mathcal{L}_4, \mathcal{C}_4$ are the same with the ones in $\mathcal{L}_3, \mathcal{C}_3$. In this case, $Consistent_{\mathsf{s1},\mathsf{s5},\mathsf{s6}}(\mathcal{L}_3, \mathcal{L}_4, \mathcal{C}_3, \mathcal{C}_4)$ and q is the node where two configurations diverge. Since $\mathsf{s1} \lhd \mathsf{s5}, \mathsf{s1} \lhd \mathsf{s6}$ and $\mathsf{s5} \prec \mathsf{s6}$, $Ordered(\mathcal{L}_3, \mathcal{L}_4, \mathcal{C}_3, \mathcal{C}_4)$. In other words, configurations $(\mathcal{L}_3, \mathcal{C}_3)$ and $(\mathcal{L}_4, \mathcal{C}_4)$ are ordered.

Another set of relations is necessary to describe the data dependences. Recall that we use a read&write analysis to compute the read set R_{s} and write set W_{s} for each non-call block s. These sets allow us to define two binary predicates: $\mathit{Write}_{\mathsf{s}}(u, v)$ if running s on u will write to v; $\mathit{ReadWrite}_{\mathsf{s}}(u, v)$ if running s on u will read or write to v. The following predicate describes two configurations $(\mathcal{L}_1, \mathcal{C}_1, \mathsf{s}, u)$ and $(\mathcal{L}_2, \mathcal{C}_2, \mathsf{t}, v)$ with data dependence if both last records $(\mathsf{s}, u, \dots)$ and $(\mathsf{t}, v, \dots)$ access the same node z and at least one of the accesses is a write:

$$\begin{aligned}\mathit{Dependence}_{\mathsf{s},\mathsf{t}}(u, v, \mathcal{L}_1, \mathcal{L}_2, \mathcal{C}_1, \mathcal{C}_2) \equiv \mathit{Configuration}(\mathcal{L}_1, \mathcal{C}_1, \mathsf{s}, u) \wedge \mathit{Configuration}(\mathcal{L}_2, \mathcal{C}_2, \mathsf{t}, v) \\ \wedge \exists z. \Big(\Big(\mathit{ReadWrite}_{\mathsf{s}}(u, z) \wedge \mathit{Write}_{\mathsf{t}}(v, z) \Big) \vee \Big(\mathit{Write}_{\mathsf{s}}(u, z) \wedge \mathit{ReadWrite}_{\mathsf{t}}(v, z) \Big) \Big)\end{aligned}$$

Example 4.4.4. Considering another configuration $(\mathcal{L}_5, \mathcal{C}_5)$ with execution path $r \to p \to q$ shown in Figure 4.9b. The labels in configuration $(\mathcal{L}_5, \mathcal{C}_5)$ on nodes r, p are the same with the ones in configuration $(\mathcal{L}_3, \mathcal{C}_3)$. Labels L_{s1} and L_{s7} are on node q. Thus $\mathit{Dependence}_{\mathsf{s3},\mathsf{s5}}$ $(w, q, \mathcal{L}_3, \mathcal{L}_5, \mathcal{C}_3, \mathcal{C}_5)$ is true since $s3$ is writing $\mathsf{n.Odd}$ on node w while $s7$ is reading $\mathsf{n.Odd}$ on w.

4.4.3 Data Race Detection and Equivalence Checking

Now we are ready to encode some common dependence analysis queries to MSO. A data race may occur in a RETREET program P if there exist two parallel configurations between which there is data dependence:

$$\begin{aligned}\mathit{DataRace}[\![P]\!] \equiv \bigvee_{\mathsf{q}_1, \mathsf{q}_2 \in \mathsf{AllNonCalls}} \exists x_1, x_2, \mathcal{L}_1, \mathcal{L}_2, \mathcal{C}_1, \mathcal{C}_2. \Big(\\ \mathit{Dependence}_{\mathsf{q}_1,\mathsf{q}_2}(x_1, x_2, \mathcal{L}_1, \mathcal{L}_2, \mathcal{C}_1, \mathcal{C}_2) \wedge \mathit{Parallel}(\mathcal{L}_1, \mathcal{L}_2, \mathcal{C}_1, \mathcal{C}_2) \Big)\end{aligned}$$

Theorem 4.4.1. *A RETREET program P is data-race-free if $\mathit{DataRace}[\![P]\!]$ is invalid.*

Proof. If P is not data-race-free, there must exist two iterations, represented as $(\mathcal{L}_1, \mathcal{C}_1)$ and $(\mathcal{L}_2, \mathcal{C}_2)$ and running blocks q_1 and q_2 on nodes x_1 and x_2, respectively, such that there is data dependence but no happens-before relation between them. This pair witnesses the

validity of the formula $\mathit{DataRace}[\![P]\!]$, as *Dependence* encodes data dependences and *Parallel* encodes the absence of happens-before. □

Besides data race detection, another critical query is the equivalence between two RETREET programs, which is common in program optimization. For example, when two sequential tree traversals A(); B() are fused into a single traversal AB(), one needs to check if this optimization is valid, i.e., if A(); B() is equivalent to AB(). Again, while the equivalence checking is a classical and extremely challenging problem, we focus on comparing programs that are built on the same set of straight-line blocks and simulate each other. The comparison is sufficient since the goal of the RETREET framework is to automate the verification of common program transformations such as fusion or parallelization, which only reorder the operations of a program.

Definition 4.4.1. *Two RETREET programs P and P' bisimulate if there exists a mapping between blocks $Sim : \mathsf{AllBlocks}(P) \to \mathsf{AllBlocks}(P')$ such that*

- *for any $\mathsf{q} \in \mathsf{AllNonCalls}$, q and $Sim(\mathsf{q})$ are identical (modulo variable renaming).*
- *Sim is a bijective mapping between $\mathsf{AllNonCalls}(P)$ and $\mathsf{AllNonCalls}(P')$.*
- *for any call $\mathsf{s} \in \mathsf{AllCalls}(P)$, s and $Sim(\mathsf{s})$ are calling the same node.*
- *if $\mathsf{s} \lhd \mathsf{t}$ in P, then $Sim(\mathsf{s}) \lhd Sim(\mathsf{t})$ in P'.*
- *if $\mathsf{s}' \lhd \mathsf{t}'$ in P' and $Sim(\mathsf{t}) = t'$, then there is a unique s such that $\mathsf{s} \lhd \mathsf{t}$ and $Sim(\mathsf{s}) = \mathsf{s}'$.*
- *for any nodes u, v, s, speculative values M, N, and any blocks $\mathsf{s}', \mathsf{t}, \mathsf{t}'$ such that $Sim(\mathsf{s}) = \mathsf{s}'$ and $Sim(\mathsf{t}) = t'$, the path conditions $\overline{\mathit{PathCond}_{\mathsf{s},\mathsf{t}}}(u, v, M, N)$ and $\overline{\mathit{PathCond}_{\mathsf{s}',\mathsf{t}'}}(u, v, M, N)$ are equivalent.*

Intuitively, P and P' bisimulate if any configuration for P can be converted to a corresponding configuration for P', and vice versa. It is not hard to develop a naive bisimulation-checking algorithm to check if two RETREET programs P and P' bisimulate: just enumerate all possible relations between P blocks and P' blocks, by brute force.

The correspondence between configurations can be extended to executions, i.e., every execution of P corresponds to an execution of P' that runs exactly the same blocks of code on the same nodes, and vice versa. To guarantee the equivalence, it suffices to make sure that the correspondence does not swap any pair of ordered configurations with data dependences. [5] In the following formula, the predicates $Dependence^{P}_{\mathsf{q}_1,\mathsf{q}_2}$ and $Dependence^{P'}_{\mathsf{q}_1,\mathsf{q}_2}$ guarantee four configurations, two on P and two on P', and pair-wisely bisimulating (as they end with the same blocks).

$$
\begin{aligned}
Conflict[\![P, P']\!] \equiv \bigvee_{\mathsf{q}_1,\mathsf{q}_2 \in \mathsf{AllNonCalls}} \exists x_1, x_2, \mathcal{L}_1, \mathcal{L}_2, \mathcal{C}_1, \mathcal{C}_2, \mathcal{L}'_1, \mathcal{L}'_2, \mathcal{C}'_1, \mathcal{C}'_2. \Big(\\
Dependence^{P}_{\mathsf{q}_1,\mathsf{q}_2}(x_1, x_2, \mathcal{L}_1, \mathcal{L}_2, \mathcal{C}_1, \mathcal{C}_2) \ \wedge \ Dependence^{P'}_{\mathsf{q}_1,\mathsf{q}_2}(x_1, x_2, \mathcal{L}'_1, \mathcal{L}'_2, \mathcal{C}'_1, \mathcal{C}'_2) \\
\wedge \ Ordered^{P}(\mathcal{L}_1, \mathcal{L}_2, \mathcal{C}_1, \mathcal{C}_2) \ \wedge \ Ordered^{P'}(\mathcal{L}'_2, \mathcal{L}'_1, \mathcal{C}'_2, \mathcal{C}'_1) \Big)
\end{aligned}
$$

Theorem 4.4.2. *For any two data-race-free* RETREET *programs P and P' that bisimulate, they are equivalent if $Conflict[\![P, P']\!]$ is invalid.*

Proof. According to Definition 4.4.1, it can be proved by recursion that there is a one-to-one correspondence between the configurations for P and the configurations for P' such that the corresponding configurations are running the same block of code. Therefore for any execution of P, P' can run exactly the same set of iterations, and vice versa. Furthermore, as $Conflict[\![P, P']\!]$ is invalid, the corresponding executions keep the same order for all pairs of dependent iterations. Therefore the two executions are equivalent. The correctness of the formula encoding can be verified by readers. □

Theorem 4.4.3. *The MSO encoding for Theorems 4.4.1 and 4.4.2 is incomplete.*

Proof. Since the outputs of speculative execution are arbitrary, the precision of the path conditions is lost. Consider a function f as shown in Figure 4.11 where height and size recursively compute the height and size of the tree, respectively. Due to speculative execution, the call to f(n.l) is considered reachable since arbitrary h and s values are legal. However,

[5] ↑ We assume both programs are free of data races; otherwise the equivalence between them is undefined.

```
f(n)
  ...
  h = height(n.l)
  s = size(n.l)
  if (h == 5 && s == 3)
    f(n.l)
  ...
```

Figure 4.11. Example of incompleteness

f(n.l) is unreachable during real computation since height of a tree can never be greater the size of the tree. □

4.5 Evaluation

We prototyped the Retreet framework, which implements all techniques presented above and also incorporates other existing MSO-based analysis techniques. We evaluated the effectiveness and efficiency of the framework through four case studies: mutually recursive size-counting traversals, CSS minification, cycletree routing, and list sum-and-shift traversals. For the first two case studies, we synthesized provably-correct optimizations (parallelizing a traversal and/or fusing multiple traversals) using MSO encoding. More concretely, our prototype constructed a candidate fused program by heuristically enumerating possible mappings that establish the bisimulation relation between the original and fused programs, and finally checked their data-race-freeness and equivalence using the MSO encoding presented in this paper. For cycletree routing, our prototype automatically verified some manually-crafted optimizations. The list sum-and-shift traversals, our prototype verified known optimizations using a combination of configuration-based abstraction presented in this paper and the Streaming Register Transducer (SRT) techniques for streaming list traversals [123]. To the best of our knowledge, none of these verification tasks can be automatically done by existing techniques before Retreet.

Our framework leverages Mona [119], a state-of-the-art WS2S (weak MSO with two successors) logic solver as our back-end constraint solver. All experiments were run on a server with a 40-core, 2.2GHz CPU and 128GB memory running Fedora 26. The bisimulation

```
Fused(n)
  if (n == nil) return (0, 0)
  else
    (ls, lv) = Fused(n.l)
    (rs, rv) = Fused(n.r)
    return (ls + rs + 1, lv + rv)
```

(a) A valid fusion

```
Fused(n)
  if (n == nil) return (0, 0)
  else
    (ret1, ret2) = (ls + rs + 1, lv + rv)
    (ls, lv) = Fused(n.l)
    (rs, rv) = Fused(n.r)
    return (ret1, ret2)
```

(b) An invalid fusion

Figure 4.12. Fusing two mutually recursive traversals

checking step is currently done by hand but can be automated in the future. The time spent on program construction and encoding is negligible. Remember our MSO encodings of data-race-freeness and equivalence are sound but not complete, the negative answers could be spurious. To this end, whenever MONA returned a counterexample, we manually investigated if it corresponds to a real evidence of violation.

Mutually Recursive Size-Counting This is our running example presented in Figure 4.4. We synthesized a fused traversal shown in Figure 4.12a and verified that the mutually recursive traversals Odd and Even can be fused to the single traversal (solved by MONA in 0.14s). This simple synthesis and verification task, to our knowledge, is already beyond the capability of all existing approaches. We also designed an invalid fused traversal (shown in Figure 4.12b) and encode the fusibility to MSO. MONA returned a counterexample in 0.14s that illustrates how the data dependence is violated. Basically, the read-after-write dependence between a child and its parent in traversal Even is violated after the fusion. We manually verified that the counterexample is a true positive.

We also checked the data-race-freeness of the original program. The two parallel traversals Odd(n) and Even(n) in the main function are independent because in every layer of the tree there is exactly one Odd call and one Even call and they belong to different traversals on each layer of the tree. The data-race-freeness was checked in 0.02s.

CSS Minification Cascading Style Sheets (CSS) are a widely-used style sheet language for web pages. In order to lessen the page loading time, many minification techniques are adapted to reduce the size of CSS document so that the time spent on delivering CSS doc-

```
ConvertValues(n)
  if (n == nil) return 0
  else
    for each child p: ConvertValues(n.p)
    if (n.type == ''word'' || n.type == ''func'')
      n.value = TransValue(n.value)
MinifyFont(n)
  if (n == nil) return 0
  else
    for each child p: MinifyFont(n.p)
    if (n.prop == ''font–weight'')
      n.value = MinifyWeight(n.value)
ReduceInit(n)
  if (n == nil) return 0
  else
    for each child p: ReduceInit(n.p)
    if (length(n.value) < initialLength)
      n.value = ReduceInitial(n.value)
```

```
Main(n)
  ConvertValues(n)
  MinifyFont(n)
  ReduceInit(n)
```

Figure 4.13. CSS minification traversals

uments can be reduced [124]–[128]. When minifying the CSS file, the Abstract Syntax Tree (AST) of the CSS code is traversed several times to perform different kinds of minifications, such as shortening identifiers, reducing whitespaces, etc. In the case that the same AST is traversed multiple times, fusing the traversals together would be desirable to enhance the performance of minification process.

Hence, we consider fusing three CSS minification traversals. Traversal ConvertValues converts values to use different units when conversion result in smaller CSS size. For instance, 100ms will be represented as .1s. Traversal MinifyFont will try to minimize the font weight in the code. For example, font-weight: normal will be rewritten to font-weight: 400. Traversal ReduceInit reduces the CSS size by converting the keyword initial to corresponding value when keyword initial is longer than the property value. For example, min-width: initial will be converted to min-width: 0. Notice that these programs involve conditions on string which are not supported by Retreet. Nonetheless, since the traversals in Figure 4.13 only manipulate the local fields of the AST, these conditions can be replaced by some simple arithmetic conditions. Moreover, as the ASTs of CSS programs are typically not binary

trees and cannot be handled by MONA directly, we converted the ASTs to left-child right-sibling binary trees and then simplify the traversals to match RETREET syntax. The three minification traversals are fused and their fusibility was checked in 6.88s.

We believe RETREET is the first framework to synthesize and verify these CSS traversal fusions. The CSS minification technique proposed by Hague *et al.* [129] also aims to generate minimized CSS file with the original semantics of the file preserved. However, they focus on one type of CSS minification method, called rule-merging, only, while RETREET can reason about the fusibility of different kinds of CSS minification methods.

Cycletree Routing Our most challenging case study is about Cycletrees [130], a special class of binary trees with an additional set of edges. These additional edges serve the purpose of constructing a Hamiltonian cycle. Hence, cycletrees are especially useful when it comes to different communication patterns in parallel and distributed computation. For instance, a broadcast can be efficiently processed by the tree structure while the cycle order is suitable for point-to-point communication. Cycletrees are proven to be an efficient network topology in terms of degree and number of communication links [130]–[132].

We consider two traversals regarding cycletrees. A traversal, called RootMode, is a mutually recursive traversal that constructs the cyclic order on a binary tree to transform the binary tree to a cycletree. Another traversal ComputeRouting computes the router data of each node which are essential for an efficient cycletree routing algorithm presented in [130]. In the event of cyclic order traversal and routing had to be performed repeatedly—in case of link failures—it would be useful to think about ways we can optimize these procedures by fusion or parallelization. Figure 4.14 shows the code for these two traversals.

We first consider checking the fusibility of these two traversals RootMode and ComputeRouting. Since the mapping relation between the unfused traversals and expected fused one is very subtle and does not satisfy the bisimulation relation defined in Definition 4.4.1, we designed the fused traversal manually and apply RETREET to verify the correctness of the fusion. The total time spent to verify the fusibility of these two traversals was 490.55s.

We then considered whether the two traversals can run in parallel. This time MONA spent 0.95s and returned a counterexample which allows us to discover a data race that

```
RootMode(n, number)
  if (n == nil) return
  else
    n.num = number
    number = number+1
    PreMode(n.l, number)
    PostMode(n.r, number)
PreMode(n, number)
  if (n == nil) return
  else
    n.num = number
    number = number+1
    PreMode(n.l, number)
    InMode(n.r, number)

InMode(n, number)
  if (n == nil) return
  else
    PostMode(n.l, number)
    n.num = number
    number = number+1
    PreMode(n.r, number)
PostMode(n, number)
  if (n == nil) return
  else
    InMode(n.l, number)
    PostMode(n.r, number)
    n.num = number
    number = number+1

ComputeRouting(n)
  if (n == nil) return
  else
    ComputeRouting(n.l)
    ComputeRouting(n.r)
    n.lmin = n.l.min
    n.rmin = n.r.min
    n.lmax = n.l.max
    n.rmax = n.r.max
    n.max = MAX(n.lmax, n.rmax, n.num)
    n.min = MIN(n.lmin, n.rmin, n.num)
Main(n)
  RootMode(n, 0)
  ComputeRouting(n)
```

Figure 4.14. Ordered cycletree construction and routing data computation

violates a read-after-write dependence. We manually verified that the counterexample is indeed a true positive.

List Sum and Shift In this case study, we show how RETREET integrates other MSO-based techniques and enables optimizations not possible with any of the techniques alone. Consider the two list traversals discussed in [133] (as shown in Figure 4.15a). Traversal Sum updates the local fields v in the list to the aggregation of values v in the list. Traversal Shift shifts the element in the list to the left and sets the last element in the list to be 0. A

```
Sum(n)
  if (n == nil) return
  else
    Sum(n.next)
    nv = n.next ? 0 : n.next.v
    n.v = nv + n.v
Shift(n)
  if (n == nil) return
  else
    nv = n.next ? 0 : n.next.v
    n.v = nv
    Shift(n.next)
```

(a) Traversals on list

```
Fused(n)
  if (n == nil) return
  else
    nv = n.next ? 0 : n.next.v
    n.v = nv
    Fused(n.next)
    nv = n.next ? 0 : n.next.v
    n.v = nv + n.v
```

(b) Single fused traversal (swapped order)

Figure 4.15. Two functions traversing a list

program invokes Sum followed by Shift. Sakka [133] shows the two traversals can be fused at the cost of an extra field for each node. However, if one swaps the order of the two traversals (step 1, from Sum(n);Shift(n) to Shift(n);Sum(n)), they can be fused without introducing the extra field and form the optimal program (step 2, from Shift(n);Sum(n) to Figure 4.15b). While the core RETREET can verify step 2, unfortunately, it is not sufficient to verify step 1, since there does not exist a relation between the original and the swapped traversals that preserves all data dependences in the original program.

Nonetheless, we extended RETREET to support other existing MSO-based analysis techniques. For example, both Sum and Shift can be described by streaming register transducer (SRT) [123], an automaton-based machine model for what they call *streaming transformations* with additive operations, which are essentially list traversals. It is also shown in [123] that these traversals are closed under composition and can be defined in MSO. The crux of the proof is: for every node y of the output list, there exists a set of nodes $N(y)$ from the input list such that the data value stored in y is the sum of values stored in $N(y)$. Following their encoding, we can define two MSO predicates:

$$sum(x, y) \equiv x \leq y$$

$$shift(x, y) \equiv x.next = y$$

such that $sum(x, y)$ (resp. $shift(x, y)$) means x belongs to the set $N(y)$ for traversal Sum (resp. Shift). We can further encode similar predicates for "sum then shift" and "shift then sum", respectively:

$$sum\_shift(x, y) \equiv \exists z.shift(x, z) \wedge sum(z, y)$$

$$shift\_sum(x, y) \equiv \exists z.sum(x, z) \wedge shift(z, y)$$

Then RETREET verifies the validity of step 1 by checking the validity of the following formula:

$$shift\_sum(x, y) \leftrightarrow sum\_shift(x, y)$$

Furthermore, RETREET verifies the validity of step 2. The whole chain of optimization was verified automatically, for the first time, in 0.11s.

4.6 Related Work

There has been much prior work on program dependence analysis for tree data structures. Using shape analyses [134], Ghiya *et al.* [135] detect function calls that access disjoint subtrees for parallel computation in programs with recursive data structures. Rugina *et al.* [136] extract symbolic lower and upper bounds for the regions of memory that a program accesses. Instead of providing a framework that describes dependences in programs, these works only focus on detecting the data races and the potential of parallel computing so that is not able to handle fusion or other transformations.

Amiranoff *et al.* [120] propose instance-wise analysis to perform dependence analysis for recursive programs involving trees. This framework represents each dynamic instance of a statement by an execution trace, and then abstracts the execution trace to a finitely-presented control word. Nonetheless, the framework does not support applications other than parallelization and they cannot handle programs with tree mutation. Weijiang *et al.* [121] also present a tree dependence analysis framework that reason the legality of point blocking,

traversal slicing and parallelization of traversals with the assumption that all traversals are identical preorder traversals. Their framework allows restricted tree mutations including nullifying or creating a subtree but the traversals that they consider are also single node traversals like RETREET. Deforestation [137]–[141] is a technique widely applied to fusion, but it either does not support fusion over arbitrary tree traversals, or does not handle reasoning about imperative programs.

The last decade has seen significant efforts on reasoning transformations over recursive tree traversals. Meyerovich *et al.* [115], [116] focus on fusing tree traversals over ASTs of CSS files. They specify tree traversals as attribute grammars and present a synthesizer that automatically fuses and parallelizes the attribute grammars. Their framework only supports traversals that can be written as attribute grammars, basically layout traversals. Rajbhandari *et al.* [112] provide a domain specific fusion compiler that fuses traversals of k-d trees in computational simulations. Both frameworks are ad hoc, designed to serve specific applications. The tree traversals they can handle are less general than RETREET.

Most recently, TreeFuser [117] is an automatic framework that fuses tree traversals written in a general language. TreeFuser supports code motion and partial fusion, i.e., parts of a traversal (left subtree or right subtree) can be fused together when possible, even if the traversals cannot be fully fused. Their approach cannot handle transformations other than fusion. In other words, parallelization of traversals is beyond the scope of TreeFuser. Besides, TreeFuser also suffers from the restrictions that RETREET has, i.e. no tree mutation and single node traversal. PolyRec [118] is a framework that can handle schedule transformations for nested recursive programs only. PolyRec targets a limited class of tree traversals, called perfectly nested recursive programs, hence the framework is not able to handle arbitrary recursive tree traversals. Also PolyRec does not handle dependence analysis and suffers from the restriction that no tree mutation is allowed. The transformations that they handle are interchange, inlining and code motion rather than fusion and parallelization. Another deforestation transformation proposed by Sakka [133] combines fusion and tupling to optimize functional programming. Their framework focuses on runtime complexity and termination guarantees, hence they do not handle dependence analysis either. None of the dependence analysis in the frameworks above is expressive enough to handle mutual recursion.

4.7 Conclusion

We introduced Retreet, a general tree-traversal-describing language, and developed a stack-based, fine-grained representation of dynamic instances in a tree traversal. Based on the new language and new representation, we presented a MSO encoding that can check data-race-freeness and transformation correctness automatically. Our approach is more general than existing approaches and allows us to efficiently reason about traversals with sophisticated mutual recursion on real-world data structures such as CSS and cycletrees, and synthesize provably-correct optimizations. We also show our approach can be integrated with other MSO-based analysis techniques.

5. SUMMARY AND FUTURE DIRECTIONS

Program synthesis is a well-unknown technique to automate programming and have the potential to make life easier for both programmers and end-users without programming expertise. However, the inherent challenges in program synthesis keep it from being used more widely. In this thesis, we present three techniques towards more scalable and practical program synthesis from three perspectives: one comparative synthesis framework to capture optimization targets for quantitative program synthesis; one cooperative synthesis technique for pushing the scalability limit of syntax-guided synthesis; one reasoning approach for provably-correct tree traversal optimizations.

In the case of comparative synthesis, we have two key observations: 1) users find it much easier to provide their preferences on concrete programs, when it is hard to specify their optimization targets in the form of either closed-form functions or input-output examples. 2) program search and objective learning are closely related and better be done together. Utilizing observation 1, we propose an interactive synthesis framework that learns near optimal programs through comparative queries. Given observation 2, we define a unified search space for programs and objectives. Based on the unified search space, we design the first algorithm that combines program search and objective learning for comparative synthesis.

In the case of cooperative synthesis, the key observations are that enumerative synthesis and deductive synthesis have their own merits and limitations and better be combined together, and subproblems that are decomposed from a syntax-guided synthesis problem are usually more solvable than the original problem. We therefore propose a set of divide-and-conquer rules to split the synthesis problem into subproblems and solve each subproblem either by enumerative synthesis or by deductive synthesis.

In the case of Retreet, we observe that finding the right level of abstraction of program is key for automating fine-grained tree traversal reasoning. We therefore propose the stack-based representation for iterations in tree traversals which allows us to describe dependences between iterations of fine granularity, which were not possible before, yet still amenable for automated reasoning.

Looking to the future, we believe that this thesis opens up several further research directions.

Human-computer interaction In Chapter 2, we propose a novel human-computer interaction paradigm that is based on comparative queries. We believe that our framework can be extended with other query types besides COMPARE and VALIDATE. Proposing single program to the user and asking for a counterexample or some revision suggestions can be a typical kind of query to incorporate. Another interesting direction could be presenting partial programs, which require less efforts to generate, and still have the potential to discover user intent from their feedback.

Robustness Existing synthesizers, which expect informal specifications, rely heavily on quality of feedback provided by users. However, as we discussed in Chapter 2, users are very likely to provide inconsistent answers. We envision that multiple ways could be utilized to improve robustness. For example, a future direction is to investigate synthesis techniques that are less sensitive to inconsistent feedback. Another direction could be identifying conflicting inputs by formal methods techniques, such as minimal unsat core, MaxSat, etc., then discard conflicts when synthesizing programs.

Solution Quality Existing works which focuses on syntax-guided synthesis, including the one we present in Chapter 3, focuses on the functional correctness of generated program and pay less attention to the quality of solutions. Although the cooperative synthesis framework allows us to automatically discover complicated functions that were considered intractable before, the synthesized program satisfies the functional specification but may be too complicated to be readable. One promising future direction would be combining the qualitative synthesis algorithm with other quantitative synthesis tools, e.g. the comparative synthesis framework presented in Chapter 2, in order to synthesize high-quality programs. Furthermore, the programs generated by syntax-guided synthesizers are usually logical expressions. How to further generate user level code is also an interesting future work.

Automated program optimization Our work presented in Chapter 4 considers the problem of verifying the validity of program transformations. In many real-world scenarios, there may be multiple transformations applicable to the same program. These transformations may not be compatible with each other and applying transformations in different order

may impose completely different quality of transformed program. How to generate optimal program transformation automatically remains an open question.

REFERENCES

[1] J. Gottschlich, A. Solar-Lezama, N. Tatbul, *et al.*, "The three pillars of machine programming," *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages - MAPL 2018*, 2018. DOI: 10.1145/3211346.3211355. [Online]. Available: http://dx.doi.org/10.1145/3211346.3211355.

[2] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11, Austin, Texas, USA: ACM, 2011, pp. 317–330, ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926423. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926423.

[3] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS'06*, San Jose, California, USA: ACM, 2006, pp. 404–415.

[4] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling enumerative program synthesis via divide and conquer," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Legay and T. Margaria, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 319–336, ISBN: 978-3-662-54577-5.

[5] M. Puschel, J. Moura, J. Johnson, *et al.*, "Spiral: Code generation for dsp transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, Feb. 2005, ISSN: 1558-2256. DOI: 10.1109/JPROC.2004.840306.

[6] B. Delaware, C. Pit–Claudel, J. Gross, and A. Chlipala, "Fiat: Deductive synthesis of abstract data types in a proof assistant," in *POPL'15*, ACM, 2015, pp. 689–700.

[7] M. Vechev and E. Yahav, "Deriving linearizable fine-grained concurrent objects," in *PLDI'08*, ACM, 2008, pp. 125–135.

[8] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," *CoRR*, vol. abs/1611.01989, 2016. arXiv: 1611.01989. [Online]. Available: http://arxiv.org/abs/1611.01989.

[9] J. Ansel, S. Kamil, K. Veeramachaneni, *et al.*, "Opentuner: An extensible framework for program autotuning," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2014, pp. 303–315. DOI: 10.1145/2628071.2628092.

[10] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "Taso: Optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19, Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 47–62, ISBN: 9781450368735. DOI: 10.1145/3341301.3359630. [Online]. Available: https://doi.org/10.1145/3341301.3359630.

[11] P. M. Phothilimthana, A. S. Elliott, A. Wang, *et al.*, "Swizzle inventor: Data movement synthesis for gpu kernels," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 65–78, ISBN: 9781450362405. DOI: 10.1145/3297858.3304059. [Online]. Available: https://doi.org/10.1145/3297858.3304059.

[12] Y. Wang, C. Jiang, X. Qiu, and S. G. Rao, "Learning network design objectives using a program synthesis approach," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, ser. HotNets '19, Princeton, NJ, USA: Association for Computing Machinery, 2019, pp. 69–76, ISBN: 9781450370202. DOI: 10.1145/3365609.3365861. [Online]. Available: https://doi.org/10.1145/3365609.3365861.

[13] K. Huang, X. Qiu, P. Shen, and Y. Wang, "Reconciling enumerative and deductive program synthesis," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 1159–1174, ISBN: 9781450376136. DOI: 10.1145/3385412.3386027. [Online]. Available: https://doi.org/10.1145/3385412.3386027.

[14] Y. Wang, J. Liu, D. Zhang, and X. Qiu, "Reasoning about recursive tree traversals," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 47–61, ISBN: 9781450382946. DOI: 10.1145/3437801.3441617. [Online]. Available: https://doi.org/10.1145/3437801.3441617.

[15] B. Fortz and M. Thorup, "Internet traffic engineering by optimizing ospf weights," in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 2000, pp. 519–528.

[16] C.-Y. Hong, S. Kandula, R. Mahajan, *et al.*, "Achieving high utilization with software-driven wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13, Hong Kong, China: ACM, 2013, pp. 15–26, ISBN: 978-1-4503-2056-6. DOI: 10.1145/2486001.2486012. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486012.

[17] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter, "Traffic engineering with forward fault correction," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14, Chicago, Illinois, USA: Association for Computing Machinery, 2014, pp. 527–538, ISBN: 9781450328364. DOI: 10.1145/2619239.2626314. [Online]. Available: https://doi.org/10.1145/2619239.2626314.

[18] S. Jain, A. Kumar, S. Mandal, *et al.*, "B4: Experience with a globally-deployed software defined wan," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, Aug. 2013, ISSN: 0146-4833. DOI: 10.1145/2534169.2486019. [Online]. Available: https://doi.org/10.1145/2534169.2486019.

[19] A. Kumar, S. Jain, U. Naik, *et al.*, "Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 1–14, Aug. 2015, ISSN: 0146-4833. DOI: 10.1145/2829988.2787478. [Online]. Available: https://doi.org/10.1145/2829988.2787478.

[20] C. Jiang, S. Rao, and M. Tawarmalani, "Pcf: Provably resilient flexible routing," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 139–153, ISBN: 9781450379557. DOI: 10.1145/3387514.3405858. [Online]. Available: https://doi.org/10.1145/3387514.3405858.

[21] Y. Wang, H. Wang, A. Mahimkar, *et al.*, "R3: Resilient routing reconfiguration," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10, New Delhi, India: Association for Computing Machinery, 2010, pp. 291–302, ISBN: 9781450302012. DOI: 10.1145/1851182.1851218. [Online]. Available: https://doi.org/10.1145/1851182.1851218.

[22] R. Srikant, *The Mathematics of Internet Congestion Control (Systems and Control: Foundations and Applications)*. SpringerVerlag, 2004, ISBN: 0817632271. [Online]. Available: https://doi.org/10.1007/978-0-8176-8216-3.

[23] Y. Wang, C. Jiang, X. Qiu, and S. G. Rao, "Learning network design objectives using a program synthesis approach," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, ser. HotNets '19, Princeton, NJ, USA: ACM, 2019, pp. 69–76, ISBN: 978-1-4503-7020-2. DOI: 10.1145/3365609.3365861. [Online]. Available: http://doi.acm.org/10.1145/3365609.3365861.

[24] S. Gulwani, K. Pathak, A. Radhakrishna, A. Tiwari, and A. Udupa, *Quantitative programming by examples*, 2019. arXiv: 1909.05964 [cs.PL].

[25] Q. Hu and L. D'Antoni, "Syntax-guided synthesis with quantitative syntactic objectives," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds., Cham: Springer International Publishing, 2018, pp. 386–403, ISBN: 978-3-319-96145-3.

[26] G. Ammons, R. Bodık, and J. R. Larus, "Mining specifications," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '02, Portland, Oregon: Association for Computing Machinery, 2002, pp. 4–16, ISBN: 1581134509. DOI: 10.1145/503272.503275. [Online]. Available: https://doi.org/10.1145/503272.503275.

[27] K. Subramanian, A. Abhashkumar, L. D'Antoni, and A. Akella, "Detecting network load violations for distributed control planes," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 974–988, ISBN: 9781450376136. DOI: 10.1145/3385412.3385976. [Online]. Available: https://doi.org/10.1145/3385412.3385976.

[28] P. Kumar, Y. Yuan, C. Yu, *et al.*, "Semi-oblivious traffic engineering: The road not taken," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA: USENIX Association, Apr. 2018, pp. 157–170, ISBN: 978-1-939133-01-4. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/kumar.

[29] E. Danna, S. Mandal, and A. Singh, "A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering," in *2012 Proceedings IEEE INFOCOM*, 2012, pp. 846–854. DOI: 10.1109/INFCOM.2012.6195833.

[30] Y. Chang, S. Rao, and M. Tawarmalani, "Robust validation of network designs under uncertain demands and failures," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 347–362.

[31] J. Bogle, N. Bhatia, M. Ghobadi, *et al.*, "Teavar: Striking the right utilization-availability balance in wan traffic engineering," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19, Beijing, China: Association for Computing Machinery, 2019, pp. 29–43, ISBN: 9781450359566. DOI: 10.1145/3341302.3342069. [Online]. Available: https://doi.org/10.1145/3341302.3342069.

[32] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, Oct. 2011, ISSN: 1558-0008. DOI: 10.1109/JSAC.2011.111002.

[33] A. Solar-Lezama, *The sketch programmers manual*, Version 1.7.2, 2016.

[34] P. Černý and T. A. Henzinger, "From boolean to quantitative synthesis," in *Proceedings of the Ninth ACM International Conference on Embedded Software*, ser. EMSOFT '11, Taipei, Taiwan: Association for Computing Machinery, 2011, pp. 149–154, ISBN: 9781450307147. DOI: 10.1145/2038642.2038666. [Online]. Available: https://doi.org/10.1145/2038642.2038666.

[35] J. Bornholt, E. Torlak, D. Grossman, and L. Ceze, "Optimizing synthesis with metasketches," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16, St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pp. 775–788, ISBN: 9781450335492. DOI: 10.1145/2837614.2837666. [Online]. Available: https://doi.org/10.1145/2837614.2837666.

[36] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, Houston, Texas, USA: ACM, 2013, pp. 305–316, ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451150. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451150.

[37] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, Edinburgh, United Kingdom: ACM, 2014, pp. 53–64, ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594302. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594302.

[38] S. Chaudhuri, M. Clochard, and A. Solar-Lezama, "Bridging boolean and quantitative synthesis using smoothed proof search," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14, San Diego, California, USA: Association for Computing Machinery, 2014, pp. 207–220, ISBN: 9781450325448. DOI: 10.1145/2535838.2535859. [Online]. Available: https://doi.org/10.1145/2535838.2535859.

[39] G. J. Chaitin, "A theory of program size formally identical to information theory," *J. ACM*, vol. 22, no. 3, pp. 329–340, Jul. 1975, ISSN: 0004-5411. DOI: 10.1145/321892.321894. [Online]. Available: https://doi.org/10.1145/321892.321894.

[40] S. Jha and S. A. Seshia, "A theory of formal synthesis via inductive learning," *Acta Informatica*, vol. 54, no. 7, pp. 693–726, Feb. 2017, ISSN: 1432-0525. DOI: 10.1007/s00236-017-0294-5. [Online]. Available: http://dx.doi.org/10.1007/s00236-017-0294-5.

[41] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS'08*, 2008, pp. 337–340. DOI: . [Online]. Available: .

[42] L. Gurobi Optimization, *Gurobi optimizer reference manual*, 2020. [Online]. Available: http://www.gurobi.com.

[43] W. Gautschi, "Numerical analysis: An introduction," in Birkhäuser, 1997, ch. 4, p. 215.

[44] S. Boyd and L. Vandenberghe, *Convex Optimization*. USA: Cambridge University Press, 2004, ISBN: 0521833787.

[45] A. Ghosh, S. Ha, E. Crabbe, and J. Rexford, "Scalable multi-class traffic management in data center backbone networks," *IEEE Journal on Selected Areas in Communications*, vol. 31, pp. 2673–2684, 2013.

[46] N. Gvozdiev, S. Vissicchio, B. Karp, and M. Handley, "On low-latency-capable topologies, and their impact on the design of intra-domain routing," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18, Budapest, Hungary: Association for Computing Machinery, 2018, pp. 88–102, ISBN: 9781450355674. DOI: 10.1145/3230543.3230575. [Online]. Available: https://doi.org/10.1145/3230543.3230575.

[47] T. G. Dietterich, "Ensemble methods in machine learning," *Lecture Notes in Computer Science*, pp. 1–15, 2000, ISSN: 0302-9743. DOI: . [Online]. Available: .

[48] K. Subramanian, L. D'Antoni, and A. Akella, "Genesis: Synthesizing forwarding tables in multi-tenant networks," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017, Paris, France: ACM, 2017, pp. 572–585, ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009845. [Online]. Available: http://doi.acm.org/10.1145/3009837.3009845.

[49] S. Saha, S. Prabhu, and P. Madhusudan, "Netgen: Synthesizing data-plane configurations for network policies," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15, Santa Clara, California: ACM, 2015, 17:1–17:6, ISBN: 978-1-4503-3451-8. DOI: 10.1145/2774993.2775006. [Online]. Available: http://doi.acm.org/10.1145/2774993.2775006.

[50] R. Soulé, S. Basu, P. J. Marandi, *et al.*, "Merlin: A language for provisioning network resources," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14, Sydney, Australia: ACM, 2014, pp. 213–226, ISBN: 978-1-4503-3279-8. DOI: 10.1145/2674005.2674989. [Online]. Available: http://doi.acm.org/10.1145/2674005.2674989.

[51] L. Ryzhyk, N. Bjørner, M. Canini, *et al.*, "Correct by construction networks using stepwise refinement," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA: USENIX Association, 2017, pp. 683–698, ISBN: 978-1-931971-37-9. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/ryzhyk.

[52] Y. Yuan, D. Lin, R. Alur, and B. T. Loo, "Scenario-based programming for sdn policies," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '15, Heidelberg, Germany: ACM, 2015, 34:1–34:13, ISBN: 978-1-4503-3412-9. DOI: 10.1145/2716281.2836119. [Online]. Available: http://doi.acm.org/10.1145/2716281.2836119.

[53] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Network-wide configuration synthesis," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds., Cham: Springer International Publishing, 2017, pp. 261–281, ISBN: 978-3-319-63390-9.

[54] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Netcomplete: Practical network-wide configuration synthesis with autocompletion," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA: USENIX Association, 2018, pp. 579–594, ISBN: 978-1-931971-43-0. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/el-hassany.

[55] J. McClurg, H. Hojjat, P. Černý, and N. Foster, "Efficient synthesis of network updates," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15, Portland, OR, USA: Association for Computing Machinery, 2015, pp. 196–207, ISBN: 9781450334686. DOI: 10.1145/2737924.2737980. [Online]. Available: https://doi.org/10.1145/2737924.2737980.

[56] J. McClurg, H. Hojjat, N. Foster, and P. Černý, "Event-driven network programming," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 369–385, ISBN: 9781450342612. DOI: 10.1145/2908080.2908097. [Online]. Available: https://doi.org/10.1145/2908080.2908097.

[57] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Abstract interpretation of distributed network control planes," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019. DOI: 10.1145/3371110. [Online]. Available: https://doi.org/10.1145/3371110.

[58] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev, "Probabilistic verification of network configurations," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 750–764, ISBN: 9781450379557. DOI: 10.1145/3387514.3405900. [Online]. Available: https://doi.org/10.1145/3387514.3405900.

[59] R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev, "Config2spec: Mining network specifications from network configurations," in *Proceedings of 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, 2020.

[60] A. Sivaraman, A. Cheung, M. Budiu, *et al.*, "Packet transactions: High-level programming for line-rate switches," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 15–28, ISBN: 9781450341936. DOI: 10.1145/2934872.2934900. [Online]. Available: https://doi.org/10.1145/2934872.2934900.

[61] X. Gao, T. Kim, A. K. Varma, A. Sivaraman, and S. Narayana, "Autogenerating fast packet-processing code using program synthesis," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, ser. HotNets '19, Princeton, NJ, USA: Association for Computing Machinery, 2019, pp. 150–160, ISBN: 9781450370202. DOI: 10.1145/3365609.3365858. [Online]. Available: https://doi.org/10.1145/3365609.3365858.

[62] L. Shi, Y. Li, B. T. Loo, and R. Alur, "Network traffic classification by program synthesis," in *Tools and Algorithms for the Construction and Analysis of Systems*, J. F. Groote and K. G. Larsen, Eds., Cham: Springer International Publishing, 2021, pp. 430–448, ISBN: 978-3-030-72016-2.

[63] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik, "Symbolic optimization with smt solvers," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14, San Diego, California, USA: Association for Computing Machinery, 2014, pp. 607–618, ISBN: 9781450325448. DOI: 10.1145/2535838.2535857. [Online]. Available: https://doi.org/10.1145/2535838.2535857.

[64] I. Drosos, T. Barik, P. J. Guo, R. DeLine, and S. Gulwani, "Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, ser. CHI '20, Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–12, ISBN: 9781450367080. DOI: 10.1145/3313831.3376442. [Online]. Available: https://doi.org/10.1145/3313831.3376442.

[65] M. Mayer, G. Soares, M. Grechkin, *et al.*, "User interaction models for disambiguation in programming by example," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ser. UIST '15, Charlotte, NC, USA: Association for Computing Machinery, 2015, pp. 291–301, ISBN: 9781450337793. DOI: 10.1145/2807442.2807459. [Online]. Available: https://doi.org/10.1145/2807442.2807459.

[66] D. Drachsler-Cohen, S. Shoham, and E. Yahav, "Synthesis with abstract examples," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds., Cham: Springer International Publishing, 2017, pp. 254–278, ISBN: 978-3-319-63387-9.

[67] H. Peleg, S. Shoham, and E. Yahav, "Programming not only by example," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 1114–1124, ISBN: 9781450356381. DOI: 10.1145/3180155.3180189. [Online]. Available: https://doi.org/10.1145/3180155.3180189.

[68] K. Miettinen, F. Ruiz, and A. P. Wierzbicki, "Introduction to multiobjective optimization: Interactive approaches," in *Multiobjective Optimization: Interactive and Evolutionary Approaches*, J. Branke, K. Deb, K. Miettinen, and R. Słowiński, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 27–57, ISBN: 978-3-540-88908-3. DOI: . [Online]. Available: .

[69] Y. Wang, I. Avramopoulos, and J. Rexford, "Design for configurability: Rethinking interdomain routing policies from the ground up," *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 3, pp. 336–348, 2009. DOI: 10.1109/JSAC.2009.090409.

[70] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10, Cape Town, South Africa: Association for Computing Machinery, 2010, pp. 215–224, ISBN: 9781605587196. DOI: 10.1145/1806799.1806833. [Online]. Available: https://doi.org/10.1145/1806799.1806833.

[71] R. Ji, J. Liang, Y. Xiong, L. Zhang, and Z. Hu, "Question selection for interactive program synthesis," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 1143–1158, ISBN: 9781450376136. DOI: 10.1145/3385412.3386025. [Online]. Available: https://doi.org/10.1145/3385412.3386025.

[72] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A robust framework for learning invariants," in *CAV'14*, 2014, pp. 69–87. DOI: . [Online]. Available: .

[73] B. Settles, "Active learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 6, no. 1, pp. 1–114, 2012. DOI: 10.2200/S00429ED1V01Y201207AIM018. eprint: https://doi.org/10.2200/S00429ED1V01Y201207AIM018. [Online]. Available: https://doi.org/10.2200/S00429ED1V01Y201207AIM018.

[74] D. Angluin, "Queries revisited," *Theoretical Computer Science*, vol. 313, no. 2, pp. 175–194, 2004, Algorithmic Learning Theory, ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2003.11.004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S030439750300608X.

[75] H. S. Seung, M. Opper, and H. Sompolinsky, "Query by committee," in *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, ser. COLT '92, Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1992, pp. 287–294, ISBN: 089791497X. DOI: 10.1145/130385.130417. [Online]. Available: https://doi.org/10.1145/130385.130417.

[76] A. Solar-Lezama, "Program Synthesis By Sketching," Ph.D. dissertation, EECS Dept., UC Berkeley, 2008.

[77] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet data manipulation using examples," *Commun. ACM*, vol. 55, no. 8, pp. 97–105, Aug. 2012, ISSN: 0001-0782. DOI: 10.1145/2240236.2240260. [Online]. Available: http://doi.acm.org/10.1145/2240236.2240260.

[78] R. Alur, R. Bodık, G. Juniwal, *et al.*, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, 2013, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/6679385/.

[79] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama, "Search-based program synthesis," *Communications of the ACM*, vol. 61, no. 12, pp. 84–93, Nov. 2018, ISSN: 0001-0782. DOI: 10.1145/3208071. [Online]. Available: http://dx.doi.org/10.1145/3208071.

[80] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Foundations and TrendsÂ® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017, ISSN: 2325-1107. DOI: 10.1561/2500000010. [Online]. Available: http://dx.doi.org/10.1561/2500000010.

[81] A. Solar-Lezama, *Introduction to program synthesis*, 2018. [Online]. Available: https://people.csail.mit.edu/asolar/SynthesisCourse/TOC.htm.

[82] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama, "Sygus-comp 2016: Results and analysis," *https://arxiv.org/abs/1611.07627*, 2016. eprint: 1611.07627.

[83] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama, "Sygus-comp 2017: Results and analysis," Nov. 2017. eprint: 1711.11438. [Online]. Available: https://arxiv.org/abs/1711.11438.

[84] J. Jeon, X. Qiu, A. Solar-Lezama, and J. S. Foster, "Adaptive Concretization for Parallel Program Synthesis," in *Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, vol. 9207, Jul. 2015, pp. 377–394.

[85] J. Jeon, X. Qiu, A. Solar-Lezama, and J. S. Foster, "An empirical study of adaptive concretization for parallel program synthesis," *Formal Methods in System Design*, vol. 50, no. 1, pp. 75–95, Mar. 2017, ISSN: 1572-8102. DOI: 10.1007/s10703-017-0269-8. [Online]. Available: https://doi.org/10.1007/s10703-017-0269-8.

[86] N. Polikarpova, I. Kuraj, and A. Solar-Lezama, "Program synthesis from polymorphic refinement types," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, Santa Barbara, CA, USA: ACM, 2016, pp. 522–538, ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908093. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908093.

[87] N. Yaghmazadeh, C. Klinger, I. Dillig, and S. Chaudhuri, "Synthesizing transformations on hierarchically structured data," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, Santa Barbara, CA, USA: ACM, 2016, pp. 508–521, ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908088. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908088.

[88] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri, "Component-based synthesis of table consolidation and transformation tasks from examples," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, Barcelona, Spain: ACM, 2017, pp. 422–436, ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062351. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062351.

[89] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *POPL'10*, Madrid, Spain: ACM, 2010, pp. 313–326.

[90] A. Solar-Lezama, "Program sketching," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5, pp. 475–495, Oct. 2013, ISSN: 1433-2787. DOI: 10.1007/s10009-012-0249-7. [Online]. Available: https://doi.org/10.1007/s10009-012-0249-7.

[91] V. Murali, S. Chaudhuri, and C. Jermaine, "Bayesian sketch learning for program synthesis," *CoRR*, vol. abs/1703.05698, 2017. arXiv: 1703.05698. [Online]. Available: http://arxiv.org/abs/1703.05698.

[92] S. Padhi and T. Millstein, "Data-driven loop invariant inference with automatic feature synthesis," Jul. 2017. eprint: 1707.02029. [Online]. Available: https://arxiv.org/abs/1707.02029.

[93] R. Alur, D. Fisman, S. Padhi, R. Singh, and A. Solar-Lezama, "Sygus-comp 2018: Results and analysis," *https://sygus.org/comp/2018/report.pdf*, 2018.

[94] Z. Manna and R. Waldinger, "Synthesis: Dreams =¿ programs," *IEEE Transactions on Software Engineering*, vol. 5, no. 4, pp. 294–328, 1979.

[95] R. M. Burstall and J. Darlington, "A transformation system for developing recursive programs," *Journal of the ACM*, vol. 24, no. 1, pp. 44–67, Jan. 1977, ISSN: 0004-5411. DOI: 10.1145/321992.321996. [Online]. Available: http://dx.doi.org/10.1145/321992.321996.

[96] N. Polikarpova and I. Sergey, "Structuring the synthesis of heap-manipulating programs," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, Jan. 2019, ISSN: 2475-1421. DOI: 10.1145/3290385. [Online]. Available: http://dx.doi.org/10.1145/3290385.

[97] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. W. Barrett, "Counterexample-guided quantifier instantiation for synthesis in SMT," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, 2015, pp. 198–216. DOI: . [Online]. Available: .

[98] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples," in *PLDI'15*, Portland, OR, USA: ACM, 2015, pp. 229–239.

[99] O. Polozov and S. Gulwani, "Flashmeta: A framework for inductive program synthesis," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015, Pittsburgh, PA, USA: ACM, 2015, pp. 107–126, ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814310. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814310.

[100] Y. Feng, R. Martins, O. Bastani, and I. Dillig, "Program synthesis using conflict-driven learning," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018, Philadelphia, PA, USA: ACM, 2018, pp. 420–435, ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192382. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192382.

[101] W. Lee, K. Heo, R. Alur, and M. Naik, "Accelerating search-based program synthesis using learned probabilistic models," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018, Philadelphia, PA, USA: ACM, 2018, pp. 436–449, ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192410. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192410.

[102] A. Stump, G. Sutcliffe, and C. Tinelli, "Starexec: A cross-community infrastructure for logic solving," in *Automated Reasoning*, S. Demri, D. Kapur, and C. Weidenbach, Eds., Cham: Springer International Publishing, 2014, pp. 367–373, ISBN: 978-3-319-08587-6.

[103] S. Saha, P. Garg, and P. Madhusudan, "Alchemist: Learning guarded affine functions," in *CAV'15*, 2015, pp. 440–446. DOI: . [Online]. Available: .

[104] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *POPL'16*, ser. POPL '16, St. Petersburg, FL, USA: ACM, 2016, pp. 499–512, ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837664. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837664.

[105] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "TRANSIT: Specifying Protocols with Concolic Snippets," in *PLDI*, 2013, pp. 287–296.

[106] R. Alur, P. Cerný, and A. Radhakrishna, "Synthesis through unification," in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 9207, Springer, 2015, pp. 163–179.

[107] B. Caulfield, M. N. Rabe, S. A. Seshia, and S. Tripakis, "What's decidable about syntax-guided synthesis?," Oct. 2015. eprint: 1510.08393. [Online]. Available: https://arxiv.org/abs/1510.08393.

[108] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, San Jose, California, USA: ACM, 2011, pp. 62–73, ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993506. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993506.

[109] B. Li, I. Dillig, T. Dillig, K. McMillan, and M. Sagiv, "Synthesis of circular compositional program proofs via abduction," in *Tools and Algorithms for the Construction and Analysis of Systems*, N. Piterman and S. A. Smolka, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 370–384, ISBN: 978-3-642-36742-7.

[110] D. Petrashko, O. Lhoták, and M. Odersky, "Miniphases: Compilation using modular and efficient tree transformations," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, Barcelona, Spain: ACM, 2017, pp. 201–216, ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062346. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062346.

[111] S. Rajbhandari, J. Kim, S. Krishnamoorthy, *et al.*, "On fusing recursive traversals of kd trees," in *Proceedings of the 25th International Conference on Compiler Construction*, ACM, 2016, pp. 152–162.

[112] S. Rajbhandari, J. Kim, S. Krishnamoorthy, *et al.*, "A domain-specific compiler for a parallel multiresolution adaptive numerical simulation environment," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, Salt Lake City, Utah: IEEE Press, 2016, 40:1–40:12, ISBN: 978-1-4673-8815-3. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014958.

[113] Y. Jo and M. Kulkarni, "Enhancing locality for recursive traversals of recursive structures," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '11, Portland, Oregon, USA: ACM, 2011, pp. 463–482, ISBN: 978-1-4503-0940-0. DOI: http://doi.acm.org/10.1145/2048066.2048104. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048104.

[114] Y. Jo and M. Kulkarni, "Automatically enhancing locality for tree traversals with traversal splicing," in *Proceedings of the 2012 ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '12, New York, NY, USA: ACM, 2012.

[115] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodik, "Parallel schedule synthesis for attribute grammars," ser. PPoPP '13, 2013.

[116] L. A. Meyerovich and R. Bodik, "Fast and parallel webpage layout," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10, Raleigh, North Carolina, USA: ACM, 2010, pp. 711–720, ISBN: 978-1-60558-799-8. DOI: 10.1145/1772690.1772763. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772763.

[117] L. Sakka, K. Sundararajah, and M. Kulkarni, "Treefuser: A framework for analyzing and fusing general recursive tree traversals," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 76:1–76:30, Oct. 2017, ISSN: 2475-1421. DOI: 10.1145/3133900. [Online]. Available: http://doi.acm.org/10.1145/3133900.

[118] K. Sundararajah and M. Kulkarni, "Composable, sound transformations of nested recursion and loops," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, Phoenix, AZ, USA: ACM, 2019, pp. 902–917, ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314592. [Online]. Available: http://doi.acm.org/10.1145/3314221.3314592.

[119] J. Elgaard, N. Klarlund, and A. Møller, "Mona 1.x: New techniques for ws1s and ws2s," in *Computer Aided Verification*, A. J. Hu and M. Y. Vardi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 516–520, ISBN: 978-3-540-69339-0.

[120] P. Amiranoff, A. Cohen, and P. Feautrier, "Beyond iteration vectors: Instancewise relational abstract domains," in *Proceedings of the 13th International Conference on Static Analysis*, ser. SAS'06, Seoul, Korea: Springer-Verlag, 2006, pp. 161–180, ISBN: 3-540-37756-5, 978-3-540-37756-6. DOI: 10.1007/11823230\textunderscore11. [Online]. Available: http://dx.doi.org/10.1007/11823230%5Ctextunderscore11.

[121] Y. Weijiang, S. Balakrishna, J. Liu, and M. Kulkarni, "Tree dependence analysis," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2015, Portland, OR, USA: ACM, 2015, pp. 314–325, ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737972. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737972.

[122] M. L. Minsky, *Computation: Finite and Infinite Machines*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1967, ISBN: 0-13-165563-9.

[123] X. Qiu, *Streaming transformations of infinite ordered-data words*, 2020. arXiv: 2001.06952 [cs.FL]. [Online]. Available: https://arxiv.org/abs/2001.06952.

[124] J. Bleuzen, *Cssmin*, 2015. [Online]. Available: https://www.npmjs.com/package/cssmin.

[125] B. Briggs, *Cssnano*, 2015. [Online]. Available: https://cssnano.co/.

[126] S. Clay, *Minify*, 2007. [Online]. Available: https://github.com/mrclay/minify.

[127] R. Dvornov, *Csso*, 2011. [Online]. Available: https://github.com/css/csso.

[128] J. Pawlowicz, *Clean-css*, 2011. [Online]. Available: https://github.com/jakubpawlowicz/clean-css.

[129] M. Hague, A. W. Lin, and C.-D. Hong, "Css minification via constraint solving," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 2, Jun. 2019, ISSN: 0164-0925. DOI: 10.1145/3310337. [Online]. Available: https://doi.org/10.1145/3310337.

[130] M. Veanes and J. Barklund, "Natural cycletrees: Flexible interconnection graphs," *J. Parallel Distrib. Comput.*, vol. 33, pp. 44–54, Feb. 1996. DOI: 10.1006/jpdc.1996.0023.

[131] M. Veanes and J. Barklund, "Construction of natural cycletrees," *Inf. Process. Lett.*, vol. 60, no. 6, pp. 313–318, 1996. DOI: 10.1016/S0020-0190(96)00179-2. [Online]. Available: https://doi.org/10.1016/S0020-0190(96)00179-2.

[132] M. Veanes and J. Barklund, "On the number of edges in cycletrees," *Inf. Process. Lett.*, vol. 57, no. 4, pp. 225–229, 1996. DOI: 10.1016/0020-0190(95)00183-2. [Online]. Available: https://doi.org/10.1016/0020-0190(95)00183-2.

[133] L. Sakka, "Techniques for automatic fusion of general tree traversals," Ph.D. dissertation, Purdue University, 2020.

[134] N. D. Jones and S. S. Muchnick, "A flexible approach to interprocedural data flow analysis and programs with recursive data structures," in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '82, Albuquerque, New Mexico: ACM, 1982, pp. 66–74, ISBN: 0-89791-065-6. DOI: 10.1145/582153.582161. [Online]. Available: http://doi.acm.org/10.1145/582153.582161.

[135] R. Ghiya, L. J. Hendren, and Y. Zhu, "Detecting parallelism in c programs with recursive darta structures," in *Proceedings of the 7th International Conference on Compiler Construction*, ser. CC '98, London, UK, UK: Springer-Verlag, 1998, pp. 159–173, ISBN: 3-540-64304-4. [Online]. Available: http://dl.acm.org/citation.cfm?id=647474.727598.

[136] R. Rugina and M. C. Rinard, "Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 2, pp. 185–235, Mar. 2005.

[137] P. Wadler, "Deforestation: Transforming programs to eliminate trees," *Theoretical computer science*, vol. 73, no. 2, pp. 231–248, 1990.

[138] A. Gill, J. Launchbury, and S. L. Peyton Jones, "A short cut to deforestation," in *Proceedings of the conference on Functional programming languages and computer architecture*, ACM, 1993, pp. 223–232.

[139] M. Martınez and A. Pardo, "A shortcut fusion approach to accumulations," *Science of Computer Programming*, vol. 78, no. 8, pp. 1121–1136, 2013.

[140] T. Rompf, A. K. Sujeeth, N. Amin, *et al.*, "Optimizing data structures in high-level programs: New directions for extensible compilers based on staging," in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '13, Rome, Italy: ACM, 2013, pp. 497–510, ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429128. [Online]. Available: http://doi.acm.org/10.1145/2429069.2429128.

[141] L. D'Antoni, M. Veanes, B. Livshits, and D. Molnar, "Fast: A transducer-based language for tree manipulation," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014, pp. 384–394.