Accelerating Syntax-Guided Program Synthesis by Optimizing Domain-Specific Languages

Published: 08 January 2026
Accepted: 06 November 2025
Received: 10 July 2025

Accelerating Syntax-Guided Program Synthesis by Optimizing Domain-Specific Languages

ZHENTAO YE, Peking University, China
RUYI JI, Peking University, China
YINGFEI XIONG, Peking University, China
XIN ZHANG*, Peking University, China

Syntax-guided program synthesis relies on domain-specific languages (DSLs) to constrain the search space and improve efficiency. However, manually designing optimal DSLs is challenging and often results in suboptimal performance. In this paper, we propose AMaze, a novel framework that automatically optimizes DSLs to accelerate synthesis. AMaze iteratively refines a DSL by identifying key program fragments, termed feature components, whose enumeration ranks correlate with synthesis time. Using a dynamic-programming-based algorithm to calculate enumeration ranks of feature components and a machine learning model based on them, AMaze estimates synthesis cost instead of directly invoking the synthesizer, which is impractical due to high computational cost. We evaluate AMaze on state-of-the-art synthesizers, including DryadSynth, Duet, Polygen, and EUsolver, across multiple domains. Empirical results demonstrate that AMaze achieves up to 4.35× speedup, effectively reducing synthesis time while maintaining expressiveness.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; **Automatic programming**; *Programming by example.*

Additional Key Words and Phrases: Program Synthesis, Domain-Specific Languages, Optimization

ACM Reference Format:
Zhentao Ye, Ruyi Ji, Yingfei Xiong, and Xin Zhang. 2026. Accelerating Syntax-Guided Program Synthesis by Optimizing Domain-Specific Languages. *Proc. ACM Program. Lang.* 10, POPL, Article 37 (January 2026), 28 pages. https://doi.org/10.1145/3776679

1 Introduction

Modern program synthesis [Alur et al. 2013] concerns the search problem of generating programs automatically from a set of syntactic and semantic constraints. While semantic constraints specify the intents of end users, syntactic constraints make the search problem tractable by limiting the space of possible programs and providing search biases (e.g., ordering programs by size). These syntactic constraints are expressed using domain-specific languages (DSLs), thus, the choice of DSLs may significantly affect the efficiency of synthesizers [Iyer et al. 2019; Ji et al. 2020; Padhi et al. 2019]. A too general DSL can lead to a large search space, making the synthesis task hard

*Corresponding author.

Authors' Contact Information: Zhentao Ye, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, ztye@pku.edu.cn; Ruyi Ji, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, jiruyi910387714@pku.edu.cn; Yingfei Xiong, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, xiongyf@pku.edu.cn; Xin Zhang, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, xin@pku.edu.cn.

ACM 2475-1421/2026/1-ART37
https://doi.org/10.1145/3776679

to solve, while an overly constrained DSL may include no viable program, making the synthesis task unrealizable. Typically, a DSL is manually designed by an expert by considering the target domain and sometimes the synthesis algorithm for speeding up specific synthesizers. In this paper, we aim to automate the process of designing DSLs that achieve high synthesis efficiency, focusing on optimizing DSLs for modern syntax-guided synthesizers.

An intuitive approach to search for a DSL with high synthesis efficiency is to start with a general and highly-expressive DSL and gradually refine it to reduce the search space while keeping sufficient expressiveness. The main challenge in the search process is how to evaluate the performance of a candidate DSL. A straightforward approach is to directly invoke the synthesizer on a representative training set of synthesis tasks with the new DSL and calculate the speedup against the initial DSL. However, it may take hundreds of years to evaluate all the new DSLs searched on some of our datasets! As a result, the key problem our paper addresses is how to efficiently evaluate the synthesis cost without invoking the synthesizer directly.

To address this problem, let us first consider the naive enumerative synthesizer, which enumerates programs from small to large until finding a program that satisfies all the constraints. Since the time to verify whether a program satisfies the constraints typically does not vary much (e.g., verifying whether a program satisfies all input-output examples), the enumeration time is mainly determined by the number of programs enumerated before finding a viable program, which we denote as the *rank* of this solution program [1]. As a result, the lowest rank of all viable programs can be applied to estimate the synthesis time of a task under a given DSL.

While state-of-the-art syntax-guided synthesizers have become significantly more complex, we find that the above estimation method can be extended to them within a unified framework. **The key observation is that these synthesizers all follow a similar workflow (for scalability), which breaks down the synthesis task of a large program into enumerating program fragments until they are able to compose a solution program using these fragments.** Here, enumeration is adopted as it is highly efficient for synthesizing small programs. As a result, the synthesizer's time consumption to synthesize the solution program is highly correlated with the time to find its largest program fragment, which can be estimated using the rank of the fragment. We refer to such a program fragment as the **feature component** of the solution program. Following this observation, to estimate the synthesizer's performance in solving a task under a given DSL, we can first find the solution program, then identify the feature component based on the synthesis algorithm, and finally estimate the synthesis time based on the rank of the feature component.

Based on the above analysis, we propose AMaze, a framework that automatically modifies a DSL to accelerate a synthesizer for a problem domain specified by a representative set of synthesis tasks. AMaze first invokes the synthesizer to generate solution programs on the problem set and identifies the feature component of each program based on the synthesis algorithm. Then it applies a search algorithm to iteratively modify the DSL to minimize the synthesis cost on the problem set, which is estimated using the ranks of the feature components.

To effectively implement AMaze, there are two technical challenges we need to solve. First, the solution program (e.g., the smallest viable program for the naive enumerative synthesizer) for a task can change as the DSL changes, so estimating the synthesis cost based on the feature component of the initial solution may become inaccurate as the search progresses. To solve this challenge, AMaze invokes the synthesizer periodically to generate a set of possible solution programs, and to estimate the synthesis cost of a new DSL, it first rewrites the programs using the new DSL if possible and then uses the smallest rank of the feature components extracted from the rewritten

[1]We denote the set of programs that satisfy the task specification as the set of *viable programs* for the task, and the viable program returned by the synthesizer as the *solution program* for the task.

programs to perform the estimation. The intuition is that the estimated synthesis times of the collected programs are upper-bounds of the real synthesis time to find the solution program. Since each modification to the DSL is typically local, the least upper-bound can serve as an effective approximation of the real synthesis time.

The second challenge is how to efficiently compute the rank of a feature component under a given DSL—we cannot simply enumerate all smaller programs, as the enumeration takes exponential time. To address this challenge, we introduce a novel dynamic programming algorithm whose time complexity is polynomial in the program size under the given DSL.

To evaluate the effectiveness of AMaze, we implemented it and applied it to optimize DSLs for state-of-the-art syntax-guided synthesizers in various domains, including DryadSynth [Ding and Qiu 2024] for bitvector programs, Duet [Lee 2021] for string manipulation programs, PolyGen [Ji et al. 2021] for conditional linear integer arithmetic programs, and EUsolver [Alur et al. 2017] for the aforementioned three types of programs considered together. As the result, AMaze helps these synthesizers achieve speedups of 1.09×, 3.62×, 1.53× and 4.35× on average, respectively.

We summarize our contributions below:

- We propose a framework for accelerating syntax-guided synthesizers by optimizing DSLs.
- We propose the concept of feature components, whose ranks in enumeration can effectively estimate the time cost of various state-of-the-art algorithms for a given synthesis task.
- We demonstrate the effectiveness of our approach empirically on representative syntax-guided program synthesis algorithms.

2 Overview

This section illustrates AMaze on a motivating example, where we optimize a DSL to accelerate one popular synthesizer, EUsolver [Alur et al. 2017], in the domain of string manipulation.

2.1 Setup

The input to AMaze comprises an initial DSL, a set of training synthesis tasks, a set of pre-defined DSL modifications, and a target synthesizer. AMaze optimizes the DSL by selecting a sequence of modifications from the pre-defined set, with the goal of accelerating the synthesizer on the training set; then, AMaze will return the modified DSL as the result, which is expected to speed up the synthesizer on similar unseen tasks. This subsection describes the input in our example below.

Initial DSL. The grammar of our initial DSL (denoted as $\mathcal{L}_0$) is as follows:

$$\begin{array}{rcl} S & := & x \mid \text{“ ”} \mid \text{“|”} \mid \text{ConCat}(S,S) \mid \text{Substr}(S,I,I) \mid \text{Ite}(B,S,S) \\ I & := & 0 \mid 1 \mid I+I \mid \text{Indexof}(S,S,I) \\ B & := & \text{true} \mid \text{false} \mid \text{Contains}(S,S) \end{array}$$

In this grammar, S is the start non-terminal representing string expressions, while I and B represent integer and boolean expressions, respectively. Among the operators, $\text{ConCat}(s_1,s_2)$ concatenates two strings, $\text{Substr}(s,i_1,i_2)$ returns the substring of s that begins at index i_1 with a length of i_2 (> 0), and $\text{Ite}(b,s_1,s_2)$ is the if-then-else operator with s_1 being the expression when b is true. The symbol $+$ is the arithmetic addition operator, $\text{Indexof}(s_1,s_2,i)$ returns the index of the first occurrence of s_2 in s_1 starting at the position i, and $\text{Contains}(s_1,s_2)$ returns whether s_1 contains s_2.

Training Set. For simplicity, our training set comprises only one task from SyGus-Comp [Padhi 2019]. Its semantic constraint is in the form of input-output examples, as listed below:

$$\begin{array}{rclcrcl} \{x = \text{“TL-18273982| 10M”}\} & \mapsto & \text{“TL-18273982”} & \quad & \{x = \text{“CT-576”}\} & \mapsto & \text{“CT-576”} \\ \{x = \text{“TL-288762| 76DK”}\} & \mapsto & \text{“TL-288762”} & & \{x = \text{“N/A”}\} & \mapsto & \text{“N/A”} \end{array}$$

These examples direct to a program that returns the substring before "| " if such a substring exists, otherwise returns the input itself. This task, just like most program synthesis tasks, has multiple viable programs in $\mathcal{L}_0$—below are two examples:

$$\begin{array}{ll} (p_1) & \text{Ite}(\text{Contains}(x, \text{" "}), \text{Substr}(x, 0, \text{Indexof}(x, \text{"|"}, 0)), x) \\ (p_2) & \text{Substr}(\text{ConCat}(x, \text{"|"}), 0, \text{Indexof}(x, \text{"|"}, 0)) \end{array} \tag{1}$$

Modifications. AMaze considers two kinds of operations to modify a DSL: deletions and additions. The deletion operation simply removes production rules, and the addition operation introduces new operators by composing existing rules. For example, the rules $I := I + I$ and $I := 1$ can be composed into $I := \text{inc}(I)$ where the operator $\text{inc}(a)$ is defined as $a + 1$. An addition operation can also enforce the equivalence between parameters. For example, $I := \text{double}(I)$ can be added by introducing $\text{double}(a)$ as $a + a$, which enforces the two operands of + to be the same.

AMaze collects a set of atomic modifications at the beginning. It discovers the set of deletions by taking all grammar rules in the original DSL $\mathcal{L}_0$, except those producing input and constants, as candidates; and discovers the set of additions by mining frequent patterns from the solution programs in the training set—in this section, we consider the following three additions for illustration:

$$f_0(a, b, c) = \text{Ite}(\text{Contains}(a, \text{" "}), b, c) \qquad f_1(a, b) = \text{Substr}(a, 0, b) \qquad f_2(a) = \text{Indexof}(a, \text{"|"}, 0)$$

In summary, AMaze considers 9 atomic modifications in our example task, including 6 deletions (deleting any production rule except ones producing constants and the input) and the 3 additions above. These modifications then yield a search space of 2^9 DSLs.

Synthesizer. Let us consider one popular synthesizer, EUsolver, which synthesizes by enumerating programs and composing them together using a decision tree. Concretely, the enumerator of EUsolver enumerates programs from small to large—it runs as follows under our original DSL $\mathcal{L}_0$.

- Programs of size 1: x, " ", "|"
- Programs of size 3: $\text{ConCat}(x, x), \ldots$
- Programs of size 4: $\text{Substr}(x, 0, 0), \ldots$
- $\ldots$

EUsolver runs the enumerator to continuously generate straight-line program fragments (i.e., programs without if-then-else operators), and periodically checks whether these fragments can be composed into a solution program (using if-then-else operators) via a decision tree learning algorithm. For example, on our training synthesis task, EUsolver will generate the aforementioned viable program p_1 (Formula 1, restated below), as its solution program.

$$(p_1) \qquad \text{Ite}(\text{Contains}(x, \text{" "}), \text{Substr}(x, 0, \text{Indexof}(x, \text{"|"}, 0)), x)$$

In this program, the straight-line fragments $\text{Contains}(x, \text{" "})$, $\text{Substr}(x, 0, \text{Indexof}(x, \text{"|"}, 0))$, and x are discovered by enumeration. Note that although p_2 (Formula 1), the other viable program on this task, has a smaller size overall, EUsolver will generate p_1 first because p_2 itself is a straight-line program and is behind all straight-line program fragments in p_1 in the small-to-large enumeration.

2.2 Efficiently Evaluating the Performance of the Synthesizer under a DSL

The main challenge in searching for an appropriate DSL is how to efficiently evaluate a candidate DSL, that is, to evaluate the time costs of the synthesizer on the training set under this candidate DSL. In this section, we introduce our key insight, describe the two technical challenges to achieve efficient evaluation, and show our solution.

Feature Components. Our key insight is that, for many state-of-the-art synthesizers, their solution programs comprise special program fragments whose ranks in enumeration can effectively estimate the time costs of the synthesizers. We denote such fragments as *feature components*.

Let us take EUsolver as an example. EUsolver finds its solution program once its enumerator finds all necessary straight-line program fragments, so its running time is highly correlated with when the largest fragment is found, which in turn is reflected by its rank in the enumeration. Correspondingly, the feature component in the solution program of EUsolver can be defined as the largest straight-line fragment comprised. For example, consider EUsolver's solution program to our training task, restated below, with the straight-line program fragments underlined:

$$\text{Ite}\left(\underline{\text{Contains}(x, \text{`` ''})}, \underline{\text{Substr}(x, 0, \text{Indexof}(x, \text{``|''}, 0))}, \underline{x}\right)$$

There are three straight-line fragments in this program, and the feature component is the largest fragment $\text{Substr}(x, 0, \text{Indexof}(x, \text{``|''}, 0))$. The rank of this feature component is 3,158, which means EUsolver needs to enumerate 3,158 programs before composing the solution program through the decision tree learning algorithm; and clearly, a larger rank of the feature component leads to a greater time cost in the enumeration part, and usually greater total synthesis time.

As another example, let us now consider the following modified DSL, denoted as $\mathcal{L}_1$.

$$\begin{array}{lcl} S & := & x \mid \text{`` ''} \mid \text{``|''} \mid \text{Ite}(B, S, S) \mid f_1(S, I) \\ I & := & 0 \mid 1 \mid f_2(S) \\ B & := & \text{true} \mid \text{false} \mid \text{Contains}(S, S) \end{array} \qquad \textbf{where} \quad \begin{array}{lcl} f_1(a, b) & = & \text{Substr}(a, 0, b) \\ f_2(a) & = & \text{Indexof}(a, \text{``|''}, 0) \end{array}$$

Starting from the initial DSL, $\mathcal{L}_1$ is obtained by adding the two operators f_1 and f_2 and deleting the original operators Concat, Substr, +, and Indexof. Then, the solution program of EUsolver becomes as follows, still with the straight-line fragments underlined:

$$\text{Ite}(\underline{\text{Contains}(x, \text{`` ''})}, \underline{f_1(x, f_2(x))}, \underline{x})$$

The feature component of this program (i.e., the largest straight-line fragment) is now $f_1(x, f_2(x))$, whose rank is 58. This number is far smaller than the previous number 3,158 under $\mathcal{L}_0$, so that $\mathcal{L}_1$ can help EUsolver find the solution program much earlier, leading to a significant speedup.

We observe that feature components exist not only in EUsolver, but also in various popular synthesizers that follow a similar workflow of composing enumerated programs into a solution program. Example synthesizers include DryadSynth [Ding and Qiu 2024], Duet [Lee 2021], and PolyGen [Ji et al. 2021], which we discuss in detail in Section 4.

Next, we discuss two technical challenges and how we solve them in order to utilize the ranks of feature components to evaluate synthesizer performance under a given DSL.

How to Get Solution Programs. In order to identify feature components, we need to first get solution programs. But how can we do it without invoking the synthesizer? After all, the purpose of extracting feature components is to avoid frequent direct invocations to the synthesizer.

To see our solution, let us compare the two solution programs of EUsolver under $\mathcal{L}_0$ and $\mathcal{L}_1$:

$$\begin{array}{ll} (\text{under } \mathcal{L}_0) & \text{Ite}(\text{Contains}(x, \text{`` ''}), \text{Substr}(x, 0, \mathit{Indexof}(x, \text{``|''}, 0)), x) \\ (\text{under } \mathcal{L}_1) & \text{Ite}(\text{Contains}(x, \text{`` ''}), f_1 \qquad (x, \quad f_2(x)) \qquad\qquad , x) \end{array}$$

These two programs are essentially the same if we unfold f_1 and f_2 by their definitions! This comparison tells us the fact that, since $\mathcal{L}_1$ is modified from $\mathcal{L}_0$ rather than being created from scratch, most programs in $\mathcal{L}_1$ are closely related to those in $\mathcal{L}_0$. Therefore, whenever we obtain the solution program under the initial DSL $\mathcal{L}_0$, we can rewrite it into a viable program under $\mathcal{L}_1$, as long as we record how $\mathcal{L}_1$ is modified from $\mathcal{L}_0$.

Can we generalize this program rewriting idea to get solution programs under all modified DSLs? A pitfall is that the solution program generated by the synthesizer might be different for a modified DSL from the one under the initial DSL even if we apply rewriting. For example, consider another modified DSL $\mathcal{L}_2$ with the following production rule added.

$$S := f_3(S, I) \quad \textbf{where } f_3(a, b) = \text{Substr}(\text{ConCat}(a, \text{“|”}), b, \text{Indexof}(a, \text{“|”}, 0))$$

Then, the solution program under $\mathcal{L}_2$ will become $f_3(x, 0)$—it cannot be trivially rewritten from the previous solution program under $\mathcal{L}_0$ (e.g., by folding the matches of f_3).

However, even though the solution program under $\mathcal{L}_0$ is not the solution program under $\mathcal{L}_2$, it is still a viable program that meets the specification under $\mathcal{L}_2$. As a result, the feature component rank of the initial solution program is *always* an upper bound of the feature component rank of the real solution program under $\mathcal{L}_2$, so we can still take the former as an *approximation* of the latter. Note that, in general, we still need to rewrite the initial program into the modified DSLs, and there can be multiple candidate results of the rewrite. In such cases, we will always select the rewrite result that has the lowest feature component rank.

Besides, although this approximation may be imprecise initially, AMaze will keep improving its precision during the search. Specifically, AMaze periodically invokes the synthesizer under the optimal DSL found so far to solve training tasks and records all solution programs. When evaluating a new candidate DSL, AMaze will first rewrite all recorded programs to the new DSL, and then use the least rank among all feature components (on each training task) to estimate the synthesizer's performance. Generally speaking, the more solution programs collected, the more accurate the estimate of the synthesizer's performance under a new DSL will be.

Calculating Ranks Effectively. The second challenge is how to efficiently calculate the rank of the extracted feature component, i.e., the number of programs to be enumerated before reaching the feature component. To address this challenge, we further estimate this rank by the number of programs whose size is no larger than the size of the feature component—it is an upper approximation once the enumeration proceeds from small to large. This number can be efficiently calculated via a dynamic programming algorithm, which will be elaborated in Section 5.2.2.

2.3 Workflow of AMaze

Now we explain the workflow of AMaze. For simplicity, in this section, we assume the synthesis cost of EUsolver is strictly proportional to the rank of the feature component. Then, the task of optimizing a DSL for EUsolver turns out to be minimizing the rank of the feature component.

Preparing Stage. AMaze first invokes the synthesizer with the initial DSL on the training tasks and collects all solution programs. These solution programs are used in two ways.

First, AMaze applies a frequent pattern mining algorithm [Zaki 2002] to extract possible operators for addition. More specifically, although the scope of addition in theory covers all compositions of production rules, it is impractical to really consider all these possibilities—they will not only form a huge and intractable search space but also bring in the risk of overfitting. To avoid such issues, AMaze limits the scope of addition to only the frequent patterns mined from the solution programs. Since we only consider a single solution program p_1 in our example, for illustration purposes, we directly assume the following three functions as the considered operators for addition:

$$f_0(a, b, c) = \text{Ite}(\text{Contains}(a, \text{“ ”}), b, c) \qquad f_1(a, b) = \text{Substr}(a, 0, b) \qquad f_2(a) = \text{Indexof}(a, \text{“|”}, 0)$$

Second, the solution programs will be used in the rewriting process to estimate the feature component ranks under modified DSLs, and then will be used to estimate the synthesis cost.

Table 1. The sequence of modifications on the DSL by AMaze

Modifications on $\mathcal{L}_0$	**Feature Component Rank**
init	3158
add $I := f_2(S)$	327
add $S := f_1(S, I)$	122
delete $S :=$ ConCat(S, S)	104
delete $I :=$ Indexof(S, S, I)	86
delete $I := I + I$	70
delete $S :=$ Substr(S, I, I)	58

Main Search Loop. AMaze runs by iteratively searching for an optimal modification sequence. It can be combined with any off-the-shelf search algorithm (such as the hill climbing method or genetic algorithm). In each iteration, AMaze applies the search algorithm to find the best possible DSL under the current estimation. For example, when the hill climbing method is used, AMaze evaluates all DSLs that can be obtained by modifying the current DSL in one step (adding one rule or deleting one rule), and selects the best one in terms of the estimated synthesis cost. This process repeats until the current DSL cannot be improved by any single modification.

In this search procedure, to evaluate the synthesizer's performance, AMaze maintains a set of known solution programs, which initially contains the solution programs under the initial DSL. When a new DSL comes up, AMaze rewrites the known solution programs into the new DSL and selects the least rank of the feature components on each task.

At last, when the current best DSL is found, AMaze invokes the synthesizer (with the current best DSL) on the whole training set, collects all solution programs, translates these programs back to the initial DSL, and updates the set of known solution programs if any new solution program has been found—these new programs help improve our estimation in future iterations.

In our example, suppose we use the hill climbing method as the search algorithm, AMaze can finally find the aforementioned $\mathcal{L}_1$ as the final result, with 6 modifications applied. These modifications are illustrated step-by-step in Table 1.

Better Search Algorithms. In practice, it is generally impossible to traverse all modified DSLs (above 2^{60} DSLs), and the hill climbing method is prone to falling into local optima. This problem can be addressed by many well-known search algorithms, and we apply a genetic algorithm in our implementation. In every iteration, AMaze updates the best candidate DSL by the genetic algorithm and collects more solution programs (by invoking the synthesizer with the current DSL) to improve the estimation. AMaze terminates after 100 iterations or when no better DSL is found after 10 consecutive iterations, and returns the best candidate DSL.

3 Problem Definition

In this section, we formalize the problem studied in this paper, the *DSL optimization problem.*

3.1 Preliminaries

A *domain-specific language (DSL)* comprises a program space specified by a context-free grammar and an interpreter that interprets each program as a function from inputs to outputs.

Definition 3.1 (Domain-Specific Language). A DSL is a pair $(G, [\![\cdot]\!])$ of a context-free grammar G and an interpreter $[\![\cdot]\!]$ that maps each program p in G to a function $[\![p]\!]$ from inputs to outputs.

A program synthesizer takes a DSL and a *synthesis task* as the input. Its goal is to find a program (in the DSL) that meets the requirements of the task. This paper does not care about the concrete form of synthesis tasks, so we simply formalize it as a predicate that judges whether the semantics of a program is viable for the task.

Definition 3.2 (Synthesizer). A synthesizer $\mathcal{S}$ takes a DSL $\mathcal{L} = (G, [\![\cdot]\!])$ and a synthesis task φ as the input. Its output $\mathcal{S}(\mathcal{L}, \varphi)$ is either a program in G or a reserved symbol $\perp$ representing a synthesis failure. The synthesizer should be sound in the sense that its solution program must be viable for the task, as shown below.

$$(p^* \in G) \to \varphi([\![p^*]\!]) \qquad \textbf{where } p^* = \mathcal{S}(\mathcal{L}, \varphi)$$

There are multiple possible reasons for synthesis failures; for example, the synthesizer may fail to find a viable program within the resource limit, or there may be no viable program at all.

3.2 DSL Optimization Problem

The input of the *DSL optimization problem* comprises a synthesizer, an initial DSL, a training set of synthesis tasks, and a set of *DSL modifications*. Its goal is to generate a new DSL by applying the modifications so that the performance of the synthesizer is optimized on the training set. In this paper, we consider a special class of DSL modifications that not only modify a DSL but also keep the correspondence between programs in its input and output DSLs.

Definition 3.3 (DSL Modification). A DSL modification m comprises a modification function *mod*, a forward translator *forward*, and a backward translator *backward*. Given an input DSL $\mathcal{L}$, the modification result $mod(\mathcal{L})$ is a new DSL, whose programs are connected with their original versions in $\mathcal{L}$ via two functions $forward_{\mathcal{L}}$ and $backward_{\mathcal{L}}$—$forward_{\mathcal{L}}$ maps a program in $\mathcal{L}$ forward to a set of related programs in $mod(\mathcal{L})$, and $backward_{\mathcal{L}}$ maps a program in $mod(\mathcal{L})$ backward to a set of related programs in $\mathcal{L}$.

The translators are required to be sound in the sense of preserving program semantics. Specifically, we call two programs p and p' in different DSLs *semantically equivalent* if $[\![p]\!]$ and $[\![p']\!]'$ have the same input-output behavior (where $[\![\cdot]\!]$ and $[\![\cdot]\!]'$ are the interpreter of the DSLs of p and p', respectively). Then, we require all programs in the output set of $forward_{\mathcal{L}}$ and $backward_{\mathcal{L}}$ to be semantically equivalent to the input program.

Intuitively, the translators capture program equivalence established through syntactic rewrites. Note that the translators are not limited to one-to-one mappings, since a DSL modification may make a program inexpressible or lead to multiple ways to rewrite a program. For example, suppose there is a program $p_0 = f_1(f_2(x))$ in the original DSL $\mathcal{L}$. If the modification removes f_1, p_0 will no longer exist in the new DSL, making $forward_{\mathcal{L}}(p_0)$ an empty set; on the other hand, if the modification composes f_1 and f_2 into a new function f_3, p_0 will have two different implementations in the DSL, i.e., p_0 and $p_1 = f_3(x)$, making $forward_{\mathcal{L}}(p_0)$ a set with two elements.

In this paper, we consider two kinds of modifications: additions and deletions. An addition is akin to defining a function, which adds a new production rule by composing several rules together; and a deletion is akin to removing useless functions, which deletes a production rule from the grammar. We implement the translators of these modifications in a straightforward way.

- For addition, *forward* generates all possible ways to replace code fragments with the new function, while *backward* unfolds the new functions back to the original code fragment.
- For deletion, *forward* maps a program to an empty set if the deleted rule is used, and maps to a singleton set containing the program otherwise; in contrast, *backward* always returns a singleton set containing that program.

Example 3.4. In our motivating example, we apply a modification add $S := f_1(S, I)$ to introduce a production rule $S := f_1(S, I)$ with a new function $f_1(a, b) = \text{Substr}(a, 0, b)$. Let $\mathcal{L}_0$ denotes the original DSL, and let p be the program $\text{Ite}(\text{Contains}(x, \text{“ ”}), \text{Substr}(x, 0, \text{Indexof}(x, \text{“|”}, 0)), x))$ in $\mathcal{L}_0$. Then, the forward translator of this modification maps p to two programs, as shown below; they respectively correspond to the choices of using the new function f_1 or not.

$$\mathit{forward}_{\mathcal{L}_0}(p) = \{p', p\} \qquad \textbf{where } p' = \text{Ite}(\text{Contains}(x, \text{“ ”}), f_1(x, \text{Indexof}(x, \text{“|”}, 0)), x)$$

In the other direction, the backward translator $\mathit{backward}_{\mathcal{L}_0}$ maps the new program p' back to the singleton set $\{p\}$, as the new function f_1 must be unfolded in the original DSL.

As an extension, we introduce three auxiliary functions Mod, Forward, and Backward to apply multiple modifications. Given an initial DSL $\mathcal{L}$ and a sequence $\overline{m} = [m_1, \dots, m_n]$ of modifications, these functions compose the corresponding functions for each modification in order, as follows.

- $\textsc{Mod}(\mathcal{L}, \overline{m})$ represents the resulting DSL of applying $m_1, \dots, m_n$ to $\mathcal{L}$ in order.
- For any $p \in \mathcal{L}$, $\textsc{Forward}(\mathcal{L}, \overline{m}, p)$ returns a set of equivalent programs in $\textsc{Mod}(\mathcal{L}, \overline{m})$. It first applies the forward translator of m_1 to p, then applies the forward translator of m_2 to each program in the resulting set and takes the union, and so on.
- Similarly, for any $p \in \textsc{Mod}(\mathcal{L}, \overline{m})$, $\textsc{Backward}(\mathcal{L}, \overline{m}, p)$ returns a set of equivalent programs in $\mathcal{L}$ obtained by applying the backward translators of $m_n, \dots, m_1$ in order to p.

With all the above notations, we are ready to define the *DSL optimization problem.*

Definition 3.5 (DSL Optimization Problem). Given a synthesizer $\mathcal{S}$, an initial DSL $\mathcal{L}$, a training set T of synthesis tasks, and a set M of DSL modifications, the DSL optimization problem aims to find a sequence $\overline{m}$ of DSL modifications such that the following objective function is maximized.

$$\frac{\sum_{\varphi \in T} \left[p^*_\varphi \neq \bot \vee p_\varphi \neq \bot\right] \left(\left[p_\varphi \neq \bot\right] \mathit{time}(\mathcal{S}, \mathcal{L}, \varphi) + \left[p_\varphi = \bot\right] \mathit{timeout}\right)}{\sum_{\varphi \in T} \left[p^*_\varphi \neq \bot \vee p_\varphi \neq \bot\right] \left(\left[p^*_\varphi \neq \bot\right] \mathit{time}(\mathcal{S}, \mathcal{L}^*, \varphi) + \left[p^*_\varphi = \bot\right] \mathit{timeout}\right)}$$

where $\mathcal{L}^*$ denotes the modification result $\textsc{Mod}(\mathcal{L}, \overline{m})$, p_φ and p^*_φ respectively denote the solution programs under the original DSL and the modified DSL, i.e., $\mathcal{S}(\mathcal{L}, \varphi)$ and $\mathcal{S}(\mathcal{L}^*, \varphi)$. Then, $[\phi]$ is a 01 function that takes 1 when ϕ is true, $\mathit{time}(\mathcal{S}, \mathcal{L}, \varphi)$ represents the time cost for $\mathcal{S}$ to synthesize a program under DSL $\mathcal{L}$ for task φ, and *timeout* is the time limit for $\mathcal{S}$ to solve one task.

This cost function calculates the average acceleration ratio achieved under the modified DSL on the training set, ignoring tasks where $\mathcal{L}$ and $\mathcal{L}^*$ both fail.

4 Feature Components and Their Identification

In this section, we describe our main innovation: what a feature component is and how we identify the feature component from a solution program. This concept helps us estimate the time cost of synthesis, given a synthesizer and a solution program.

4.1 Composition-Based Synthesizers and Feature Components

In this paper, we consider a composition-based framework of program synthesis, which is widely used in many state-of-the-art syntax-guided synthesizers. As shown in Figure 1, this framework comprises an *enumerator* and a *composer*. In each iteration, the enumerator first generates a certain number of small components according to a certain order (usually by size), and then the composer composes these components into a candidate solution program. This candidate will be returned as the result if it is verified to be correct; otherwise, the framework will increase the number of components and start the next iteration.

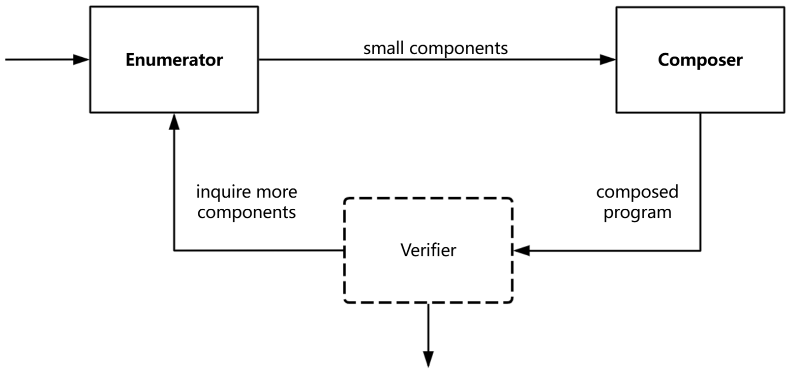

Fig. 1. The composition-based synthesis framework.

Definition 4.1 (Composition-Based Synthesizer). A *composition-based synthesizer* is a pair of an *enumerator* and a *composer.*

- The **enumerator** $\mathcal{E}_r$ is parameterized by a ranking function r, which takes a DSL $\mathcal{L}$ and maps each program in $\mathcal{L}$ to a distinct positive integer. Given the ranking function, $\mathcal{E}_r$ takes a DSL $\mathcal{L}$ and a positive integer n as the inputs, and returns the set of all programs in $\mathcal{L}$ whose rank is no larger than n:

$$\mathcal{E}_r(\mathcal{L}, n) = \{p \in \mathcal{L} \mid r(\mathcal{L}, p) \leq n\}$$

- The **composer** C composes a set of small components into a viable program for a synthesis task. Given a DSL $\mathcal{L}$, a set of components $P \subseteq \mathcal{L}$, and a synthesis task, the composer C explores a subspace $\mathcal{L}_C(P) \subseteq \mathcal{L}$ (i.e., the set of programs composed from P) and will return either a viable program in $\mathcal{L}_C(P)$, or the failure symbol $\perp$ if no viable program exists.

Given a DSL $\mathcal{L}$ and a synthesis task φ, the composition-based synthesizer explores the following sub-spaces in order, and returns the first viable program returned by the composer.

$$\mathcal{L}_C\big(\mathcal{E}_r(\mathcal{L}, 1)\big), \quad \mathcal{L}_C\big(\mathcal{E}_r(\mathcal{L}, 2)\big), \quad \ldots, \quad \mathcal{L}_C\big(\mathcal{E}_r(\mathcal{L}, n)\big), \quad \ldots$$

The time cost of composition-based synthesizers can be estimated with the concept of *feature components.* According to the above definition, the time cost depends heavily on the number of iterations used. The more iterations there are, the more times the enumerator and the composer are invoked, resulting in a higher time cost. Moreover, given the target program p, the number of needed iterations is determined by the highest rank among the components in p that must be generated by the enumerator. We refer to such components as *enumerated components*, and refer to the enumerated component with the highest rank as the *feature component.*

Definition 4.2 (Enumerated Components). Given a composition-based synthesizer $(\mathcal{E}_r, C)$, a DSL $\mathcal{L}$, a synthesis task φ, and a target program p, the *enumerated components* $\mathrm{EC}_{\mathcal{S},\mathcal{L},\varphi}(p)$ are defined as follows, where $\mathrm{Sub}(p)$ denotes the set of program fragments in p.

$$\mathrm{EC}_{\mathcal{S},\mathcal{L},\varphi}(p) = \mathrm{Sub}(p) \cap \mathcal{E}_r(\mathcal{L}, n^*) \qquad \textbf{where } n^* = \min(\{n \in \mathbb{N} \mid p \in \mathcal{L}_C\big(\mathcal{E}_r(\mathcal{L}, n)\big)\})$$

Definition 4.3 (Feature Component). Given a composition-based synthesizer $(\mathcal{E}_r, C)$ with ranking function r, a DSL $\mathcal{L}$, a synthesis task φ, a target program p, and a set of enumerated components

$\mathrm{EC}_{\mathcal{S},\mathcal{L},\varphi}(p)$, the feature component of p, denoted as $\mathrm{FC}_{\mathcal{S},\mathcal{L},\varphi}(p)$, is defined as follows.

$$\mathrm{FC}_{\mathcal{S},\mathcal{L},\varphi}(p) = \operatorname*{argmax}_{ec} r(\mathcal{L}, ec) \qquad \textbf{for } ec \in \mathrm{EC}_{\mathcal{S},\mathcal{L},\varphi}(p)$$

For simplicity, we abbreviate the enumerated components as $\mathrm{EC}(p)$ and the feature component as $\mathrm{FC}(p)$ if the synthesizer, the DSL, and the synthesis task are clear from the context.

Although $\mathrm{FC}(p)$ is defined over the synthesis procedure, it can usually be computed efficiently with only the syntax of the target program p, without actually running the synthesizer. In the remainder of this section, we consider four state-of-the-art composition-based synthesizers, including DryadSynth, Duet, PolyGen, and EUsolver, and show how to efficiently compute the feature components for these synthesizers.

4.2 Feature Component Identification

This section focuses on how to find all enumerated components—it is straightforward to identify the feature component from the enumerated components once we can compute the rank of each component, which will be elaborated in Section 5.2.2.

We find enumerated components by decomposing the solution program recursively, purely based on the syntax. Algorithm 1 outlines our algorithm. It is parameterized by a decision function **DF** that depends on the specific target synthesizer. Function **DF** takes two inputs, a program p' and a component *sub* of p', and returns a boolean value indicating whether *sub* needs to be further decomposed or it is already an enumerated component. Using the **DF** function, Algorithm 1 first checks whether the input program p itself is an enumerated component, and returns a singleton set containing it if it is (Line 1). Otherwise, Algorithm 1 invokes the **Decompose** function (Line 2), a recursive function that collects all enumerated components of its input program p'. It iterates through all components that are direct children of the AST root (Line 4-11). For each component, if it can be further decomposed, the function will recursively collect all enumerated components inside (Line 8). Otherwise, the component itself is an enumerated component and is added to the set EC (Line 10). Finally, the function returns all collected components (Line 11).

Next, we explain how to instantiate **DF** for various state-of-the-art synthesizers—they are all straightforwardly derived from the composition mechanism of each synthesizer.

EUsolver. EUsolver [Alur et al. 2017] uses if-then-else to compose straight-line enumerated components into a larger solution program. The **DF** function for EUsolver is quite simple:

$$\mathbf{DF}(p, sub) = \begin{cases} true & \text{if the root node of } sub \text{ is if-then-else operator} \\ false & \text{otherwise} \end{cases} \tag{2}$$

With the above **DF** function, for a solution program generated by EUsolver, Algorithm 1 traverses the decision tree of the solution program in a depth-first order and returns all the leaf programs.

PolyGen. Similar to EUsolver, PolyGen [Ji et al. 2021] also defines several "backbone" operators for decomposing synthesis problems. Apart from the if-then-else operator, PolyGen further allows the composition with Boolean operators to efficiently generate Boolean conditions. Therefore, we define a special operators set $sop_{\textsc{PolyGen}} = \{\text{ite}, \text{and}, \text{or}, \text{not}\}$ and the **DF** function can be defined as:

$$\mathbf{DF}(p, sub) = \begin{cases} true & \text{if the root node of } sub \in sop_{\textsc{PolyGen}} \\ false & \text{otherwise} \end{cases} \tag{3}$$

Algorithm 1 The decomposition procedure for identifying enumerated components

```
Require: A program $p$.
Hyperparameter: A decision function DF that specifies, for a program $s$ and a program fragment
    $sub$, whether $sub$ should be further decomposed.
Ensure: The set of enumerated components in $p$.
if not DF($p$, $p$) then return {$p$};
return DECOMPOSE($p$);

function DECOMPOSE(program $p'$):
    EC ← ∅;  $fragments$ ← SUBNODES($p'$);
    for all program $sub$ in $fragments$ do
        if DF($p'$,$sub$) then                      ▷ $sub$ should be further decomposed
            EC.APPENDALL(DECOMPOSE($sub$));
        else                                        ▷ $sub$ is an enumerated program fragment
            EC.APPEND($sub$);
    return EC;
```

$$\begin{aligned}
\mathit{ntString} :=\ & (\text{str.++ } \mathit{ntString}\ \mathit{ntString}) \mid (\text{str.replace } \mathit{ntString}\ \mathit{ntString}\ \mathit{ntString}) \mid \\
& (\text{str.at } \underline{\mathit{ntString}}\ \mathit{ntInt}) \mid (\text{str.substr } \underline{\mathit{ntString}}\ \mathit{ntInt}\ \mathit{ntInt}) \mid \\
& (\text{ite } \underline{\mathit{ntBool}}\ \mathit{ntString}\ \mathit{ntString}) \mid (\text{int.to.str } \underline{\mathit{ntString}}) \\
\mathit{ntInt} :=\ & (+\ \underline{\mathit{ntInt}}\ \mathit{ntInt}) \mid (-\ \underline{\mathit{ntInt}}\ \mathit{ntInt}) \mid (\text{str.len } \underline{\mathit{ntString}}) \mid (\text{str.to.int } \mathit{ntString}) \mid \\
& (\text{ite } \underline{\mathit{ntBool}}\ \mathit{ntInt}\ \mathit{ntInt}) \mid (\text{str.indexof } \underline{\mathit{ntString}}\ \underline{\mathit{ntString}}\ \underline{\mathit{ntInt}}) \\
\mathit{ntBool} :=\ & (=\ \underline{\mathit{ntInt}}\ \underline{\mathit{ntInt}}) \mid (\text{str.prefixof } \underline{\mathit{ntString}}\ \underline{\mathit{ntString}}) \mid \\
& (\text{str.suffixof } \underline{\mathit{ntString}}\ \underline{\mathit{ntString}}) \mid (\text{str.contains } \underline{\mathit{ntString}}\ \underline{\mathit{ntString}})
\end{aligned}$$

Fig. 2. Production rules used by DUET in the domain of string manipulation; non-terminals corresponding to program fragments that need to come directly from the enumeration part are underlined.

DUET. DUET [Lee 2021] generates small program fragments via enumeration and puts them together into the desired program using an algorithm called top-down propagation (TDP) [Polozov and Gulwani 2015]. TDP itself is a complete top-down synthesis algorithm that tries to solve the synthesis problem by decomposing the problem into smaller sub-problems. However, it suffers from the issue that there are too many ways to decompose the synthesis problem. To avoid this issue, DUET restricts some program fragments to be enumerated components and stops decomposition once these fragments are reached. This restriction is specified as annotations on production rules. For example, Figure 2 shows the annotated production rules in the domain of string manipulation, where underlined non-terminals correspond to program fragments that must come from enumeration. Based on these rules, the **DF** function can be defined as:

$$\mathbf{DF}(p, sub) = \begin{cases} \mathit{false} & \text{if } sub \text{ is expanded from an underlined non-terminal Figure 2} \\ \mathit{true} & \text{otherwise} \end{cases} \tag{4}$$

DRYADSYNTH. DRYADSYNTH [Ding and Qiu 2024] is a highly efficient synthesizer for bitvector problems. It composes enumerated components using templates like bvxor(*hole1*, *hole2*) and

Algorithm 2 Search Framework

```
Require: A DSL optimization problem specified by a synthesizer $\mathcal{S}$, an initial DSL $\mathcal{L}_0$, a training
    set $T$ of synthesis tasks, and a set $M$ of available DSL modifications.
Ensure: An optimized DSL $\mathcal{L}^*$.
1:  $bestRatio, bestDSL \leftarrow -\infty, \text{NULL}$;
2:  $evaluatedTasks \leftarrow \emptyset$;
3:  $solutionSet[t] \leftarrow \emptyset$ for each training task $t \in T$;
4:  $noImprovementCount \leftarrow 0$;
5:  while $noImprovementCount <$ a pre-defined limit or max number of iterations reached do
6:      $timePredictModel \leftarrow \textbf{Train}(T, evaluatedTasks, solutionSet)$;
7:      $\overline{m} \leftarrow \textbf{SearchWithModel}(T, timePredictModel, solutionSet, M)$;
8:      $\mathcal{L} \leftarrow \textsc{Mod}(\mathcal{L}_0, \overline{m})$;
9:      for all training task $t \in T$ do
10:         $p, time \leftarrow \textsc{SynthesisWithTime}(\mathcal{S}, \mathcal{L}, t)$;
11:         $evaluatedTasks.\textsc{Append}(\overline{m}, \mathcal{L}, t, time)$;
12:         $solutionSet[t].\textsc{AppendAll}(\textsc{Backward}(\mathcal{L}, \overline{m}, p))$;
13:     $ratio \leftarrow \textsc{EvaluateAccelerateRatio}(\mathcal{L}_0, \mathcal{L}, \mathcal{S}, T)$;
14:     if $ratio > bestRatio$ then
15:         $bestRatio, bestDSL \leftarrow ratio, \mathcal{L}$;
16:         $noImprovementCount \leftarrow 0$;        ▷ Reset counter since we find a better DSL
17:     else
18:         $noImprovementCount \leftarrow noImprovementCount + 1$;
19: return $bestDSL$;
```

bvand(*hole1*, bvand(*hole2*, *hole3*)). Therefore, the **DF** function of DryadSynth can be defined as:

$$\mathbf{DF}(p, sub) = \begin{cases} true & \text{if } p \text{ fits into a template } t \text{ and } sub \text{ is a hole program according to } t \\ false & \text{otherwise} \end{cases} \tag{5}$$

Note that, to apply the above **DF** function, we need to slightly modify the **Decompose** function in Algorithm 1 so that the whole solution program p is always used as the first parameter of **DF**.

5 Design of AMaze

AMaze solves the DSL optimization problem through search-based optimization. During the search, it efficiently estimates the synthesis time cost of each task under candidate DSLs based on the feature component ranks of solution programs. Although the synthesis time highly correlates with the feature component rank for a given task, typically it is impossible to find a DSL that minimizes feature component ranks of all tasks and the relationship between feature component ranks and synthesis times is complex. As a result, instead of directly using ranks to estimate synthesis time, AMaze trains a machine learning model on the fly that predicts synthesis time based on feature component ranks. In this section, we introduce the detailed design of AMaze.

5.1 Search Framework

Algorithm 2 shows the framework of AMaze. It searches iteratively and collects two growing sets of data: *evaluatedTasks* (Line 2) records all invocations to the synthesizer (each of which includes the sequence of modifications, the modified DSL, the synthesis task) and the corresponding synthesis time costs, which will be used for training a synthesis time prediction model based on the ranks

Algorithm 3 Model Training

```
1:  function GetFeatureRank(task $t$, modification sequence $\overline{m}$, candidate DSL $\mathcal{L}$, solutionSet):
2:      $solutions \leftarrow \bigcup_p$ Forward($\mathcal{L}, \overline{m}, p$) for $p \in solutionSet[t]$;
3:      $featureComponents \leftarrow \{$GetFeatureComponent($\mathcal{L}, \overline{m}, s$) for each $s \in solutions\}$;
4:      $featureComponent^* \leftarrow$ the smallest feature component in featureComponents;
5:      $rank \leftarrow$ CalRank($\mathcal{L}, featureSubcomponent^*$);
6:      return rank;
7:  function Train(training set $T$, evaluatedTasks, solutionSet):
8:      $trainingInput \leftarrow \emptyset$; $trainingOutput \leftarrow \emptyset$;
9:      for all (modification sequence $\overline{m}$, DSL $\mathcal{L}$, task $t$, and time cost time) $\in$ evaluatedTasks do
10:         $rank \leftarrow$ GetFeatureRank($t, \overline{m}, \mathcal{L}$, solutionSet);
11:         trainingInput.Append(rank); trainingOutput.Append(time);
12:     return TrainModel(trainingInput, trainingOutput);
```

of feature components; and for each training task t, *solutionSet*$[t]$ (Line 3) records its all known solution programs in the initial DSL space, which will be used for estimating the ranks of feature components of the solution programs under a newly discovered DSL.

In every iteration, AMaze first uses the collected data to train a synthesis time prediction model *timePredictModel* (Line 6, details in 5.2.1), which predicts the synthesis time cost of the synthesizer for a synthesis task under a candidate DSL *without invoking the synthesizer*. AMaze uses the predictions of this model to quickly estimate the objective value of a new DSL. Under the guidance of this model, AMaze searches among possible modifications and quickly generates a candidate modification sequence that is effective under the prediction of the model (Line 7, details in 5.3). Then, for the modification sequence $\overline{m}$, the framework computes its corresponding candidate DSL (Line 8) and invokes the synthesizer with this DSL for every training task (Line 10). The input information and running time of these invocations will be added to *evaluateTasks* for training a more accurate model (Line 11). The synthesized program will be backward translated to the initial DSL and recorded in *solutionSet* for better estimating the ranks of feature components of the solution programs under later-generated DSLs (Line 12, details in Section 5.2.1). Finally, AMaze calculates the current objective value (Line 13). If a better DSL has been found, it updates the result (Line 15) and resets the failure counter (Line 16); otherwise, it increments the counter (Line 18).

The loop terminates when AMaze fails to find a better DSL for a fixed number (10 in our implementation) of consecutive iterations or when it reaches the maximum number (100 in our implementation) of iterations (line 5), and it returns the DSL with the largest objective value as the final result (Line 19).

In the rest of the section, we explain how AMaze builds the time prediction model and how it searches DSLs with the model in detail.

5.2 Building the Prediction Model

Since evaluating the objective function (i.e., the acceleration ratio on solving all training tasks) by invoking the synthesizer directly is time-consuming, AMaze trains a prediction model to predict synthesis time cost for each task based on feature component ranks, which are then used to compute the objective function value.

5.2.1 Training Procedure. Algorithm 3 shows the training procedure **Train** (Line 6 in Algorithm 2) of this model. The model takes the estimated feature component rank of the solution program for a task under a DSL (calculated by **GetFeatureRank**, Lines 1-6) as the input and predicts the corresponding synthesis time.

One problem here is how to estimate the rank without exactly knowing the accurate solution program since we would like to use the prediction model for new candidate DSLs without invoking the synthesizer with them. To address this problem, AMaze estimates the rank corresponding to the accurate solution program using the known solution programs in *solutionSet* (Lines 2-5). Specifically, AMaze forward translates every program in *sotluionSet* into the current DSL space (Line 2). For each solution s obtained through this forward translation, AMaze invokes the decomposition procedure introduced in Algorithm 1 and decomposes s into a set of enumerated components and gets the feature component, which is the largest component among them (Line 3). Then, AMaze takes the smallest feature component among them (Line 4), and calculates its rank using a dynamic-programming-based approach (details in Section 5.2.2), which is used to estimate the rank of the feature component of the accurate solution program (Line 5). Note that since *solutionSet* will grow larger during the iterations (Line 12 in Algorithm 2), this estimation will become more and more accurate in the search procedure.

The training procedure of the model is straightforward (Lines 7-12). For each piece of synthesis information recorded in *evaluatedTasks*, AMaze calculates the rank of the feature component of each solution program for task t under each DSL (Line 10) and takes the rank along with the actual time cost as training data of the model (Line 11). AMaze can be instantiated with any off-the-shelf prediction model, and in our implementation, we use a gradient boosted decision tree implemented in XGBoost [Chen and Guestrin 2016] as the prediction model.

5.2.2 Calculating the Rank. AMaze calculates an upper approximation of the rank of a program in enumeration by dynamic programming. Specifically, let p^* be a program, let s be the start non-terminal of the grammar, and suppose that p^* is obtained by applying num^* production rules to s. Then, the rank of p^* can be effectively approximated by the number of programs that are not larger than p^*, which is the number of programs that can be obtained by applying up to num^* production rules to s. [2]

Let **Count**(n, num) be the number of programs that can be obtained by applying *exactly num* production rules to a non-terminal n. This function can be efficiently calculated by dynamic programming showed in Algorithm 4. It uses two global caches to speed up the computation: $CCache[n, num]$ records the number of programs that can be produced by applying *exactly num* production rules to a non-terminal n (Line 2), and $RCache[r, k, num]$ records the number of (partial) programs generated by expanding the rule r *exactly num* times, considering *only* the first k non-terminals in rule r (Line 3). If $num = 1$, **Count** sets $CCache[n, num]$ to the number of rules that start with n and only produce terminals (Line 10). Otherwise, **Count** iterates over all production rules that start with n and produce at least one non-terminal, and sets $CCache[n, num]$ to the sum of the numbers of programs obtainable by applying each such rule followed by $num - 1$ additional rules. This sum is computed using the **CountPartial** function (Line 14). **Count** returns $CCache[n, num]$ as the final result (Line 15).

CountPartial is a recursive function to calculate the number of (partial) programs generated by expanding a rule r *exactly num* times, considering *only* the first k non-terminals in rule r. In addition to the boundary condition checks (Lines 18–21), the core logic is that this number equals the sum, over all t from 1 to $num - 1$, of the numbers of programs produced by expanding the first

[2]Here we present the rank calculation method under the condition of enumeration according to the size of syntax tree nodes, which is consistent with the principle of enumeration part of most program synthesizers. An exception is DryadSynth, which enumerates programs according to the order of term graph sizes, and our tool also implements the rank calculation under this condition.

Algorithm 4 Counting the number of programs of a given size

```
Require: A non-terminal $n$ and an integer $num$
Ensure: The number of programs that can be produced by applying exactly $num$ rules to $n$
 1: global variables
 2:     $CCache \leftarrow \emptyset$;   ▷ $CCache[n, num]$ records the number of programs that can be produced by applying exactly $num$ production rules to a non-terminal $n$
 3:     $RCache \leftarrow \emptyset$;   ▷ $RCache[r, k, num]$ records the number of (partial) programs generated by expanding the rule $r$ exactly $num$ times, considering only the first $k$ non-terminals of rule $r$
 4: end global variables
 5:
 6: function Count(non-terminal $n$, int $num$):
 7:     if $CCache[n, num]$ is not NULL then
 8:         return $CCache[n, num]$;
 9:     if $num = 1$ then
10:         $CCache[n, num] \leftarrow$ number of rules that start with $n$ and only produce terminals;
11:     else
12:         $CCache[n, num] \leftarrow 0$;
13:         for all rule $r$ that starts with $n$ and produces at least one non-terminal do
14:             $CCache[n, num] \leftarrow CCache[n, num]$ + CountPartial($r$, the arity of $r$, $num - 1$);
15:     return $CCache[n, num]$;
16:
17: function CountPartial(rule $r$, int $k$, int $num$):
18:     if $num < k$ then                ▷ Each non-terminal requires at least one expansion
19:         return 0;
20:     if $k = 1$ then                  ▷ All expansions are used by the first non-terminal
21:         return Count(GetNonterminals($r$)[0],$num$);
22:     if $RCache[r, k, num]$ is not NULL then
23:         return $RCache[r, k, num]$;
24:     if $k = num$ then                ▷ Each non-terminal corresponds to exactly one expansion
25:         $tmp \leftarrow 1$;
26:         for all non-terminal $n$ in $r$ do
27:             $tmp \leftarrow tmp \times$ Count($n$, 1);
28:         $RCache[r, k, num] \leftarrow tmp$;
29:     else
30:         $RCache[r, k, num] \leftarrow 0$;
31:         for $t = 1$ to $num - 1$ do  ▷ Assume the last non-terminal uses $num - t$ expansions
32:             $RCache[r, k, num] \leftarrow RCache[r, k, num]$ + Count(GetNonterminals($r$)[$k - 1$], $num - t$) $\times$ CountPartial($r$, $k - 1$, $t$);
33:     return $RCache[r, k, num]$;
```

$k - 1$ non-terminals in r exactly t times and the last non-terminal exactly $num - t$ times (Lines 22–32). This function is efficiently implemented based on the two caches *CCache* and *RCache*.

By invoking **Count**, the rank of the program p^* can be approximated as $\sum_{i=1}^{num^*}$ **Count**(s, i).[3] [4]

5.3 Model-Guided Search

We first describe how AMaze generates the space of DSL modifications, and then how it finds an adequate sequence of modifications within the space using the aforementioned model.

5.3.1 Generating the Search Space. The choice of modifications (i.e., set M in the input of Algorithm 2) is crucial for the performance of AMaze because it defines the search space. As discussed in Section 3.2, in this paper, we consider only two basic types of modifications, *addition* and *deletion*, but we believe the performance of AMaze can be further improved if more advanced or domain-specific modifications are considered.

One problem here is that there may be an exponential number of different possible addition modifications, which will induce a search space too large for effective exploration. To address this problem, AMaze considers only those composed rules that frequently appear in solution programs. Specifically, given the initial DSL, the synthesizer, and the training set, AMaze first collects a set of solution programs by invoking the synthesizer on each training task under the initial DSL. Then, AMaze will decompose these solution programs into a family of enumerated components and discover common patterns (i.e., compositions of production rules) by applying a frequent pattern mining algorithm based on the SLEUTH algorithm [Zaki 2002] and will consider only those addition modifications corresponding to the mined patterns. The candidate rules generated in this way are generalizable and avoid overfitting that may be caused by directly introducing the complete solution program for a specific task.

The candidate space for the deletion modifications is the set of initial production rules except for rules that produce constants or inputs.

5.3.2 Search Algorithm. AMaze can be combined with any off-the-shelf search algorithm to find effective modification sequences guided by the prediction model (Line 7 in Algorithm 2). In our implementation, we adopt the genetic algorithm, which is a popular search algorithm inspired by the concept of natural selection. Briefly, it searches by iterating through a set of modification sequences, which are referred to as a population of genes. In each iteration, it produces the next generation of the population by (1) selecting a set of modification sequences as parents to participate in the next step of reproduction (i.e., *selection*), (2) generating a set of candidate modification sequences by mixing the selected parents (i.e., *crossover*), and then (3) randomly changing one of the modifications with a given probability for each candidate modification sequence (i.e., *mutation*).

The following describes more details of the adopted genetic algorithm in AMaze.

- The fitness function is set to the estimated objective function value (under the prediction of our model).
- AMaze determines the parents using roulette wheel selection [Golberg 1989] based on ranking to avoid the “elite monopoly" problem caused by excessive disparities in fitness values.
- The crossover probability is set to 0.8. If crossover occurs, AMaze performs crossover on two modification sequences by randomly selecting positions in the first sequence and replacing the modifications at those positions with their counterparts from the second sequence.

[3]Some synthesizers (e.g., Duet) need to enumerate programs from multiple non-terminals simultaneously, where each non-terminal expands num^* times. To handle this, we apply the same calculation to each non-terminal until its expansion count reaches num^*.

[4]Synthesizers like EUsolver and DryadSynth apply optimization like observational equivalence to improve enumeration efficiency. Our calculation of ranks ignores these optimizations, but the results are still highly correlated with the real values. Our machine learning predictor captures this correlation and reduces the noise caused by ignoring such optimizations.

- The mutation probability is set to 0.01. AMaze mutates a modification sequence by randomly selecting one of two operations: (1) inserting an available modification at a random position, or (2) removing a randomly selected modification.
- AMaze maintains a population of 100 modification sequences per iteration and runs for 100 iterations. It then returns the best sequence as the final candidate.

6 Empirical Evaluation

We have implemented AMaze in Python. To evaluate the effectiveness of AMaze, we try to answer the following research questions:

(1) Can AMaze accelerate state-of-the-art synthesizers across different domains by optimizing DSLs?
(2) How does AMaze compare to existing DSL modification methods that are not designed for accelerating synthesizers?
(3) Ablation Study: Of the two innovations to estimate synthesis performance (i.e., using feature components and using ranks) in AMaze, what is the independent contribution of each technique to the final result? Does removing any of them lead to a significant change in performance?
(4) Case Study: As informal evidence to reflect whether the rules introduced in the final DSLs are generalizable, are they human-readable?
(5) How is the time cost of the experiments?

In the rest of the section, we describe our experiment results by answering these questions.

6.1 Experiment Setup

Before describing our evaluation results, we explain the setup first.

Synthesis Tasks and DSLs. For synthesis tasks, we consider three application domains from SyGus-Comp 2019, the latest Syntax-Guided Synthesis Competition [Padhi 2019]:

- The string manipulation benchmark suite (String or S in short) contains 209 tasks from the SLIA (strings with linear integer arithmetic) track of the competition.
- The bitvector (BV or B) benchmark suite contains 750 problems from the BV (bit-vectors) track. The benchmarks are motivated by program deobfuscation [Jha et al. 2010].
- The conditional linear integer arithmetic (CLIA or C) benchmark suite contains 82 problems from the General track[5]. In addition, following the evaluation setup of PolyGen [Ji et al. 2021], we include 18 tasks for synthesizing combinators in divide-and-conquer algorithms, which are collected by Farzan and Nicolet [2017], resulting in 100 tasks in total.

For each domain, we adopt a ten-fold cross-validation approach to evaluate the performance of our approach and report the overall synthesizer performance on the test sets across the ten folds.

Figure 3 shows the initial DSLs of the three domains. We omit constants and inputs because they vary across tasks, and our method does not delete any rule producing a constant or an input.

Synthesizers. To evaluate the effectiveness of our approach, we consider four synthesizers: DryadSynth [Ding and Qiu 2024] for bitvector programs, Duet [Lee 2021] for string manipulation programs, PolyGen [Ji et al. 2021] for conditional linear integer arithmetic programs, and EUsolver [Alur et al. 2017] for the aforementioned three types of programs considered together.

[5]There is also a CLIA Track in SyGus-Comp 2019. We chose to use the dataset from the general track of the competition because (1) all benchmarks from the CLIA track are included in the general track, and (2) the general track also contains additional benchmarks from various domains that can be solved using programs written in the CLIA syntax.

$$\begin{aligned}
\textit{ntBV} := \quad & (\text{bvand } \textit{ntBV}\ \textit{ntBV}) \mid (\text{bvor } \textit{ntBV}\ \textit{ntBV}) \mid (\text{bvxor } \textit{ntBV}\ \textit{ntBV}) \mid \\
& (\text{bvadd } \textit{ntBV}\ \textit{ntBV}) \mid (\text{bvsub } \textit{ntBV}\ \textit{ntBV}) \mid (\text{bvmul } \textit{ntBV}\ \textit{ntBV}) \mid \\
& (\text{bvlshr } \textit{ntBV}\ \textit{ntBV}) \mid (\text{bvshl } \textit{ntBV}\ \textit{ntBV}) \mid (\text{im } \textit{ntBV}\ \textit{ntBV}\ \textit{ntBV})
\end{aligned}$$

(a)

$$\begin{aligned}
\textit{ntString} := \quad & (\text{str.++ } \textit{ntString}\ \textit{ntString}) \mid (\text{str.replace } \textit{ntString}\ \textit{ntString}\ \textit{ntString}) \mid \\
& (\text{str.at } \textit{ntString}\ \textit{ntInt}) \mid (\text{str.substr } \textit{ntString}\ \textit{ntInt}\ \textit{ntInt}) \mid \\
& (\text{ite } \textit{ntBool}\ \textit{ntString}\ \textit{ntString}) \mid (\text{int.to.str } \textit{ntString}) \\
\textit{ntInt} := \quad & (+\ \textit{ntInt}\ \textit{ntInt}) \mid (-\ \textit{ntInt}\ \textit{ntInt}) \mid (\text{str.len } \textit{ntString}) \mid (\text{str.to.int } \textit{ntString}) \mid \\
& (\text{ite } \textit{ntBool}\ \textit{ntInt}\ \textit{ntInt}) \mid (\text{str.indexof } \textit{ntString}\ \textit{ntString}\ \textit{ntInt}) \\
\textit{ntBool} := \quad & (=\ \textit{ntInt}\ \textit{ntInt}) \mid (\text{str.prefixof } \textit{ntString}\ \textit{ntString}) \mid \\
& (\text{str.suffixof } \textit{ntString}\ \textit{ntString}) \mid (\text{str.contains } \textit{ntString}\ \textit{ntString})
\end{aligned}$$

(b)

$$\begin{aligned}
\textit{Start} := \quad & (+\ \textit{Start}\ \textit{Start}) \mid (-\ \textit{Start}\ \textit{Start}) \mid (\text{ite } \textit{StartBool}\ \textit{Start}\ \textit{Start}) \\
\textit{StartBool} := \quad & (\text{and } \textit{StartBool}\ \textit{StartBool}) \mid (\text{or } \textit{StartBool}\ \textit{StartBool}) \mid (\text{not } \textit{StartBool}) \mid \\
& (<\ \textit{Start}\ \textit{Start}) \mid (=\ \textit{Start}\ \textit{Start}) \mid (\leq\ \textit{Start}\ \textit{Start}) \mid
\end{aligned}$$

(c)

Fig. 3. The initial DSLs used in synthesizing (a) BV programs, (b) String programs and (c) CLIA programs.

Table 2. Improvement in performance of different synthesizers using AMaze. The timeout is set to five minutes.

Synthesizer	Domain	# Tasks	# Solved Problems		Average Time Cost (Second)		
			Initial DSL	New DSL	Initial DSL	New DSL	Speedup
DryadSynth	BV	750	750	750	0.53	**0.49**	1.09×
Duet	String	209	208	208	1.96	**0.54**	3.62×
PolyGen	CLIA	100	81	84	30.14	**19.72**	1.53×
EUsolver	B+S+C	1059	806	935	55.43	**12.74**	4.35×

As far as we know, DryadSynth, Duet, and PolyGen are the strongest synthesizers in the domains of bitvector, string manipulation, and conditional linear integer arithmetic, respectively. And EUsolver is a popular synthesizer which works on all the three domains and solves more tasks than the previous three solvers when considering the domains together. The four synthesizers all meet the definition of a composition-based synthesizer we defined in Definition 4.1 .

Software and Hardware Environment. We ran all experiments using Python 3.11.5 and XGBoost 2.0.0 on a Ubuntu 22.04 server equipped with a 22-core, 44-thread Intel(R) Xeon(R) Gold 6238 CPU running at 2.10GHz and 256 GB of RAM.

6.2 Experiment Results

Speedups for State-of-the-Art Synthesizers. Table 2 shows the performance improvement of different synthesizers by optimizing DSLs using AMaze in terms of the number of solved problems and average synthesis time. The average synthesis time and average speedup are calculated after excluding tasks where the synthesizer fails to find a solution under both DSLs.

Table 3. Improvement in performance of different synthesizers using a modified version of AMaze that simulates Stitch. The time out is set to five minutes.

Synthesizer	Domain	# Tasks	# Solved Problems		Average Time Cost (Second)		
			Initial DSL	New DSL	Initial DSL	New DSL	Speedup
DryadSynth	BV	750	750	750	0.53	**0.49**	1.09×
Duet	String	209	208	208	1.96	**0.39**	5.03×
PolyGen	CLIA	100	81	79	**20.61**	32.51	0.64×
EUsolver	B+S+C	1059	806	276	**23.78**	207.06	0.11×

While the absolute numbers vary, AMaze accelerates all synthesizers in their excelled domains despite that the majority of the synthesizers are already highly optimized for the corresponding domains. DryadSynth is carefully designed for bitvector problems and can already solve all 750 tasks in only 0.53 seconds on average with the initial DSL. However, with the help of AMaze, DryadSynth can further reduce the average synthesis time to 0.49 seconds, showing an average speedup of 1.09×. The speedup effect is even more evident for the other synthesizers. For Duet on the string domain, it also solves almost all tasks (208 out of 209 tasks) with the initial DSL in 1.96 seconds on average. While with the optimized DSL, the time cost is reduced to 0.54 seconds, yielding an average speedup of 3.62×. As for PolyGen, AMaze not only helps it solve 3 more tasks, but also decreases the average time cost from 30.14 seconds to 19.72 seconds, resulting in an average speedup of 1.53×. At last, EUsolver, which we evaluate in all the above three domains, receives the most significant performance boost. With the optimized DSLs, EUsolver is able to solve 129 more tasks. The percentage of solved tasks increases from 76.11%(806/1059) to 88.29%(935/1059). The average synthesis time is reduced from 55.43 seconds to 12.74 seconds, yielding an average speedup of 4.35×.

In addition, AMaze improves all synthesizers consistently across different train-test splits. Across all 60 held-out folds for test (30 for EUsolver and 10 each for the other solvers), the modified DSLs enable solving more tasks on 24 folds, the same number on 31 folds, and fewer on only 5 folds. Moreover, the synthesizer solves only 1-2 fewer tasks on each of the 5 under-performing folds. All these folds are in the CLIA domain (2 for PolyGen and 3 for EUsolver), which has significantly fewer tasks than other domains.

In summary, by optimizing the DSL, AMaze has enabled DryadSynth and Duet to achieve an average speedup of 1.09× on the bitvector domain and 3.62× on the string domain, respectively. For the CLIA tasks, AMaze helps PolyGen solve 3 more tasks and achieve an average speedup of 1.53×. Finally, AMaze enables EUsolver to solve 129 more tasks and achieve an average speedup of 4.35×. This shows that AMaze is effective in speeding up state-of-the-art syntax-guided synthesizers.

Comparison with Existing DSL Modification Techniques. As far as we know, there is no existing technique that modifies the DSL specifically to accelerate various state-of-the-art syntax-guided synthesizers in different domains. However, there exists a rich body of literature in modifying DSLs for library learning [Bowers et al. 2023; Cao et al. 2023; Ellis et al. 2021], which all try to minimize the sizes of a set of training programs by composing existing DSL operators into new operators (i.e., library functions). The question is whether we can achieve a similar speedup effect following the idea of compressing programs by adding new library functions. To answer this question, we try to compare our approach with Stitch [Bowers et al. 2023], a state-of-the-art library learning method. However, we cannot apply Stitch to accelerate the aforementioned synthesizers out-of-the-box as DSLs generated by it involve higher-order functions, which are not supported by the synthesizers. Instead, we modify AMaze in the following aspects to simulate Stitch for enabling a fair comparison:

- We modify the pattern mining algorithm that discovers possible additions by allowing it to discover patterns on the whole solution programs instead of on enumerated components.
- We replace the optimization objective in the search process with minimizing

$$\mathit{sizeof}(\textsc{rewrite}(\mathit{training\ programs}), \mathcal{L}) + \mathit{penalty}(\mathcal{L})$$

where $\mathit{sizeof}(\textsc{rewrite}(\mathit{training\ programs}), \mathcal{L})$ is the sum of the sizes of the training programs rewritten in the modified DSL $\mathcal{L}$ and $\mathit{penalty}(\mathcal{L})$ is a normalization term that penalizes DSLs for introducing too many library functions, thereby preventing overfitting.

Table 3 shows the results of using the modified AMaze to accelerate the four synthesizers. Different from our approach, which accelerates all four synthesizers, the modified version only accelerates DryadSynth and Duet but slows down PolyGen and EUsolver. Particularly, with the DSL generated by the modified version, PolyGen solves 2 fewer tasks and the average time cost increases from 26.42 seconds to 41.22 seconds, yielding an average speedup of 0.64×. The results of EUsolver are even worse: using the DSLs generated by the modified version, it solves 530 fewer tasks and the average time cost has increased by almost 200 seconds (23.78 seconds to 207.06 seconds), yielding an average speedup of 0.11×. There are two reasons for this phenomenon: 1) with program sizes as the objective, the modified version has no incentive to delete production rules, which degrades its performance in reducing feature component ranks, and 2) more importantly, by minimizing the sizes of whole solution programs, the modified version introduces library functions that are not used in enumerated components, which often delays discovering them.

For DryadSynth, the modified version achieves a similar speedup as our approach. The reason is that it has a highly optimized enumerator and composer for the bitvector domains, and therefore is resilient to introducing useless library functions. As for Duet, the modified version achieves a better speedup. After inspection, we found on the string dataset, while some of the library functions the modified version introduces are useless in speeding up solving the training tasks as they span across program components found through enumeration, they happen to be able to speed up solving some of the testing tasks. Moreover, similar to DryadSynth, Duet is also resilient to introducing useless library functions. As a result, the modified version gets "lucky" due to the difference between the testing set and the training set, although it learns the wrong libraries for the training set. However, we do not believe this phenomenon is typical – As shown by the results, AMaze consistently improves all synthesizers in all domains while the modified version only improves a subset and leads to slowdowns for the others.

In summary, the library learning approach represented by Stitch is not as broadly applicable as our approach to accelerate various composition-based synthesizers because it fails to capture the synthesizer costs precisely, and can lead to severe slowdowns for certain synthesizers. In contrast, AMaze achieves consistent acceleration of different synthesizers on different domains.

Ablation Study. To test the effectiveness of the two technical innovations in estimating synthesizer performance (using feature components and using rank) separately, we designed an ablation experiment. In brief, we examined the results when only using feature components (denoted as AMaze$_c$) and the results when only using rank (denoted as AMaze$_r$), and compared them with the results from AMaze. More concretely, AMaze$_c$ predicts time cost based on the *size* (syntax tree size) of the feature components, while AMaze$_r$ uses the *overall* ranks of the solution programs as the prediction indicator. We run this experiment on the most improved synthesizer, EUsolver, on all domains with the same ten-fold partitions and report the overall performance. The results are showed in Table 4.

Table 4. Improvement in performance of synthesizer(EUSOLVER) using different method. The time out is set to five minutes.

Domain	# Tasks	Method	# Solved Problems		Average Time Cost (Second)		
			Initial DSL	New DSL	Initial DSL	New DSL	Speedup
B+S+C	1059	AMAZE$_c$	806	858	42.65	32.18	1.33×
		AMAZE$_r$	806	215	20.95	224.52	0.09×
		AMAZE	806	**935**	55.43	**12.74**	**4.35×**

We first discuss the result of AMAZE$_c$. With the optimized DSLs produced by AMAZE$_c$, compared to using the initial DSLs, *EUsolver* solves 52 more task, and the average synthesis time is reduced from 42.65 seconds to 32.18 seconds, yielding an average speedup of 1.33×. This shows that when only using feature components, AMAZE$_c$ can find new DSLs that are better than the initial DSLs, but the speedup is worse compared to AMAZE. While applying feature components precisely captures the program fragments identified through enumeration, *ranks* provide a better estimation of enumeration time than *sizes*. Compared with *ranks*, the values of *sizes* are more concentrated. For example, the sizes of most components are around 3 to 10. As a result, the time cost prediction model based on the size values is actually a function with only a few discrete values, which leads to inaccurate time estimation. Furthermore, because deleting useless productions will not affect solution program sizes, AMAZE$_c$ cannot estimate the improvement caused by deletion operations.

On the other hand, the result of AMAZE$_r$ is much worse. When using the "optimized" DSLs, compared to using the initial DSL, *EUsolver* solves 73.3% less tasks and the average synthesis time has increased from 20.95 seconds to 224.52 seconds, yielding an average speedup of 0.09×. Guided by the model based on the ranks of the overall programs, AMAZE$_r$ could hardly find any efficient DSL. This is because AMAZE$_r$ adds rules containing the if-then-else operator in an attempt to effectively reduce the rank values of the overall programs. However, in fact, the introduction of these rules does not help EUSOLVER to enumerate the commonly used components faster as these components are straight-line programs, but instead greatly increases the enumeration cost.

In summary, the ablation experiment shows that both using feature components and using ranks contribute significantly in AMAZE's performance. While applying feature components alone can still improve the synthesizer performance, although worse, applying ranks alone can even lead to slowdowns.

Case Study. We manually inspect the rules introduced in the final DSLs to investigate how human-readable these rules are and whether they contain some interpretable insights. The intuition is that the more the synthesized libraries resemble ones that humans would write, the more likely they would generalize. Since human readability reflects how closely a synthesized library resembles one written by a human, we use it as a proxy signal to informally evaluate the generalization ability of the final DSLs produced by AMAZE. Across different folds, there are about 10 to 20 candidate rules to add for CLIA, 30 to 60 for String, and 40 to 70 for BV, and the final DSL usually applies about half of them. These rules are frequent patterns abstracted from the components of the initial solution programs and are generally not too large. Here we give some examples of interpretable rules for each domain.

In the String domain, a simple but useful rule is $fs_1(a) = \text{str.++}(\text{" "}, a)$, which just adds a space before the input string. A more complex example is $fs_2(a) = \text{str.substr}(a, 0, \text{str.indexof}(a, \text{" "}, 0))$, which composes two rules str.substr and str.indexof and fixes several parameters to 0 or a space. It returns a substring of the input string before the first space.

In the BV domain[6], a pattern $fb_1(a) = \text{bvand}(a, \#x0001)$ sets all bits except the last bit to 0. It is often used together with if-then-else to perform conditional branching based on whether the last bit is 0. A more interesting example pattern is $fb_2(a) = \text{bvxor}(\text{bvsub}(\#0000, \#0001), a)$, which is equal to the bitwise not. Note that bvnot is not included in our initial BV DSL, but our approach can discover this rule from the components.

In the CLIA domain, since most solution programs consist of a large number of branch conditions and each branch program is relatively small, the number of patterns that can be mined is the least and the length is the shortest. A typical example is $\text{fc}_1(a) = +(a, +(a, a))$, which is equal to $3a$. Another example is $\text{fc}_2(a) = -(a, 1)$, which is self-explanatory. Moreover, one interesting finding is that, after introducing this rule, the subtraction operation in the initial DSL is deleted. This is because any binary subtraction expression can be replaced with an expression involving addition and fc_2, and a significant portion in the tasks are minus-one expressions.

In summary, through our manual inspection, we find that in each domain, the rules introduced in the final DSLs are interpretable, indicating good generalizability.

Time Costs. We conclude the experimental section by presenting the time efficiency of AMaze. In our experimental setting, one run of the genetic algorithm performs 100 rounds with a population size of 100 and returns the best candidate DSL. This entire search process is repeated for up to 100 runs. The total runtime is approximately 20 hours for CLIA, 30 hours for String, and 100 hours for BV (due to its larger dataset). On average, the search terminates after 20 runs, evaluating 10,000 candidate DSLs per run.

7 Discussion

Effect of Deleting Rules While our framework can be instantiated with different modification operations, it currently supports adding or deleting production rules. While the addition operation does not change the set of the expressible programs, the deletion operation can produce DSLs that are less expressive than the original DSL. We made this design choice as deleting redundant rules can speed up the enumeration process significantly in some cases. Besides deleting rules that are unused in the training set, an interesting scenario that we observed empirically, is deleting some rules after composing them into a new rule. This can be seen as replacing low-level abstractions with high-level abstractions or a process of specialization. Moreover, in our empirical evaluation, compared to addition, deletion happens much more rarely. Most of the generated DSLs in our cross-validation experiment do not involve any deletion, and the average number of deleted rules is 1.15. Most of the deletions come from the domain of conditional linear integer arithmetic (CLIA), where the DSL has redundant rules for the domain. On the other hand, all generated DSLs involve addition, and the average number of added rules is 19.03 per DSL. Finally, the overall experiment results show that our approach produces DSLs that generalize well on the testing sets, even though deletion comes at the risk of reducing DSL expressiveness.

Applying to Other Synthesizers Besides the synthesizers discussed previously, another notable synthesizer is CVC5 [Barbosa et al. 2022] whose predecessor, CVC4 [Barrett et al. 2011], performed well in recent SyGus competitions [Alur et al. 2019]. While CVC5 also involves an enumeration process, it is integrated into a constraint-solving process. As a result, it is unclear how to identify feature components in its synthesized programs, which requires opening the box of the constraint-solving process. It would be an interesting direction to explore in the future.

Another interesting family of synthesizers is based on genetic programming [Gulwani et al. 2017]. However, to our knowledge, these synthesizers do not involve a process of enumeration, so our approach cannot be applied to them. We did not focus on them, as they are not the best performers

[6]For convenience, a 16-digit hexadecimal value is displayed using only its 4-digit effective part (the last 4 digits).

in the domains we consider (i.e., ones in the SyGuS competition). We focus on synthesizers involving enumerators instead, as all synthesizers leading in these domains fall into this category, showing that it is standard to use enumerators to generate small program fragments and more sophisticated algorithms to compose them into large programs for scalability. It would be interesting to explore how to combine genetic programming-based synthesizers with enumerators, which in turn can incorporate our technique.

8 Related Work

We first discuss works that apply probabilistic models to speedup synthesizers, which is another paradigm to bias the synthesis search besides our approach. Then we talk about works that have discussed the impact of DSL design choices on synthesizer performance. Finally, we describe previous works on modifying DSLs.

Accelerating Program Synthesis Using Probabilistic Models. Various works have applied trained probabilistic models to guide the search process for better performance. Compared to modifying the DSL, they offer more fine-grained ways to bias the search process but are also tightly coupled with the synthesis algorithm. Menon et al. [2013] apply probabilistic context-free grammars to guide an enumeration-based approach to synthesize string-manipulating programs from examples. Lee et al. [2018] further improve it by using probabilistic higher-order grammars on a wider range of applications. Balog et al. [2017] use a recurrent neural network to guide the search for straight-line programs that manipulate strings and lists. Ji et al. [2020] apply structural probabilities to speedup a VSA-based algorithm on programs that manipulate strings or matrices. Kalyan et al. [2018] propose a hybrid synthesis technique that combines the advantages of both symbolic logic techniques and statistical models to optimize two neural synthesis algorithms.

Besides the works above, there are other synthesizers that integrate neural networks, and fall into the category of guiding the synthesis process using a probabilistic model in a broader sense. In addition to being tightly coupled with the synthesis algorithm and the domain, their information to guide the synthesis algorithm is very different compared to that used in our approach. Shah et al. [2020] apply a neural network to approximate the semantics of a program fragment under a non-terminal to enable end-to-end gradient-based synthesis. In some sense, their guiding information is only the input-output specification. Ye et al. [2021] apply a neural network to guide the synthesis process by encoding information from natural language constraints. Yang et al. [2021] use a neural network to predict the world, which is specific to the reinforcement learning setting. Our work, on the other hand, encodes the syntactic and semantic similarities of programs in a domain through DSLs. It would be interesting to explore combining our work with these works to enable the incorporation of different sources of guiding information.

The Impact of DSLs on Synthesizer Performance. Multiple papers have discussed that choosing the right DSL is vital to a synthesizer's performance. They motivated our work. Ji et al. [2020] show that on string manipulating programs, using a more expressive DSL allows a synthesizer to solve more tasks but also run longer to solve some tasks. Iyer et al. [2019] show that there is a trade-off in DSL design similar to the Bias-Variance trade-off in machine learning: the expressivness of the DSL affects how well the synthesizer can generalize from training examples. Padhi et al. [2019] discuss this problem more systematically and show that one can improve a synthesizer's performance by running with multiple DSLs in parallel.

Modifying DSLs. There exist previous works that modify DSLs in particular ways for various purposes. Compared to them, our work is the first to systematically study how to accelerate different synthesizers across different domains.

There is a long line of works in learning libraries, most of which are not for accelerating modern synthesizers. The earliest in this line can go back to Predicate Invention [Lin et al. 2014; Muggleton and Buntine 1988; Muggleton et al. 2015] in inductive logic programming and Automatically Defined Functions in genetic programming [O'Neill 2009]. The former adds new predicates to a logic program to generate more compact and interpretable programs. The latter forms modules to enable generating large programs. The idea of abstracting common patterns into modules in order to gradually solve more complex tasks has also been explored in curriculum learning [Dechter et al. 2013; Henderson 2013]. To compress programs, Bowers et al. [2023] propose a library learning algorithm based on a corpus of training programs. Further, the approach by Cao et al. [2023] maintains better library quality by additionally taking as input an equational theory for a given problem domain. While our approach to generating libraries is similar to these approaches at a high level, the library functions generated by these approaches can not be directly added to a DSL, and they are not for accelerating modern synthesizers. We extracted their core idea of optimizing DSLs to compress programs and conducted comparative experiments by modifying our framework to use the objective to minimize program sizes, showing that our method is more consistent and effective in accelerating synthesis algorithms. A recent work [Ellis et al. 2021] accelerates a neural-network-guided, enumeration-style synthesizer by both iteratively learning libraries and improving the neural network on the fly. While our approach is applicable to any independent program synthesizers as long as they meet our definition of composition-based synthesizers, this approach can only generate DSLs for its specific synthesizer because the synthesis algorithm needs to be modified accordingly. Moreover, it selects candidate DSLs based on compressing the overall sizes of the solution programs, which we have shown in the comparative experiment that for modern synthesis algorithms, this objective is not effective for modifying DSLs to accelerate synthesizers.

Morton et al. [2020] propose an approach to speedup a synthesizer by removing production rules in the initial DSL. While deletion alone may help reduce the width of the search tree at each level, addition can be used to change the depth of the search tree of the solution program. The latter is an exponential change, so our approach considers both addition and deletion. Moreover, they only tried to modify one synthesizer on a single domain, while we proposed a general framework applicable to multiple synthesizers across multiple domains. They use a neural network to predict whether a deletion is beneficial. It is interesting to use similar approaches to model complex relationships between DSL designs and synthesizer performance.

Chan et al. [2020] describe an approach to generate a DSL that accelerates a synthesizer by searching within a parametric DSL. However, how to define an adequate parametric DSL remains an open problem.

9 Conclusion

We have proposed AMaze, a framework to accelerate state-of-the-art syntax-guided program synthesizers by optimizing the DSL. AMaze searches for a sequence of modifications to optimize the synthesis efficiency on a given set of training tasks. To make the search process efficient, it uses the ranks of *feature components* of solution programs to estimate the synthesis time cost. Our empirical evaluation shows that our approach significantly accelerates different state-of-the-art syntax-guided synthesizers in different domains.

Finally, large language models (LLMs) recently have received significant attention in program synthesis [Hüttel 2025] by demonstrating their effectiveness in generating programs in general-purpose languages from natural languages. However, based on our experience, due to the lack of training data, they are much less effective in generating programs in DSLs compared to existing synthesizers. Even though general-purpose languages are more expressive, they cannot replace DSLs in many domains (e.g., domains relying on specialized hardware). As a result, our approach is complementary to the LLM-based synthesis at a high level.

Acknowledgements

This work was sponsored by the National Key Research and Development Program of China under Grant No. 2022YFB4501902, the National Natural Science Foundation of China (NSFC) under Grant No. 62172017, W2411051.

Data-Availability Statement

The artifact of our paper is available at Zenodo [Ye et al. 2025]. It includes the main source code, scripts, data and statistics in our experiments. All results in our experiments can be reproduced.

References

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 1–8. doi:10.1109/fmcad.2013.6679385

Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. 2019. Sygus-comp 2018: Results and analysis. *arXiv preprint arXiv:1904.07146* (2019). https://arxiv.org/abs/1904.07146

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10205)*, Axel Legay and Tiziana Margaria (Eds.). 319–336. doi:10.1007/978-3-662-54577-5_18

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://arxiv.org/abs/1611.01989

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. doi:10.1007/978-3-642-22110-1_14

Matthew Bowers, Theo X Olausson, Lionel Wong, Gabriel Grand, Joshua B Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. 2023. Top-down synthesis for library learning. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1182–1213. doi:10.1145/3571234

David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. babble: Learning Better Abstractions with E-Graphs and Anti-unification. *Proc. ACM Program. Lang.* 7, POPL (2023), 396–424. doi:10.1145/3571207

Nicolas Chan, Elizabeth Polgreen, and Sanjit A. Seshia. 2020. Gradient Descent over Metagrammars for Syntax-Guided Synthesis. *CoRR* abs/2007.06677 (2020). arXiv:2007.06677 https://arxiv.org/abs/2007.06677

Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794. doi:10.1145/2939672.2939785

Eyal Dechter, Jonathan Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. 2013. Bootstrap Learning via Modular Concept Discovery. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, Francesca Rossi (Ed.). IJCAI/AAAI, 1302–1309.

Yuantian Ding and Xiaokang Qiu. 2024. Enhanced enumeration techniques for syntax-guided synthesis of bit-vector manipulations. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 2129–2159. doi:10.1145/3632913

Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Lucas Morales, Luke B. Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 835–850. doi:10.1145/3453483.3454080

Azadeh Farzan and Victor Nicolet. 2017. Synthesis of divide and conquer parallelism for loops. *ACM SIGPLAN Notices* 52, 6 (2017), 540–555. doi:10.1145/3140587.3062355

David E Golberg. 1989. Genetic algorithms in search, optimization, and machine learning. *Addion wesley* 1989, 102 (1989), 36. doi:10.5860/choice.27-0936

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. doi:10.1561/2500000010

Robert John Henderson. 2013. *Cumulative learning in the lambda calculus.* Ph. D. Dissertation. Imperial College London, UK.

Hans Hüttel. 2025. On Program Synthesis and Large Language Models. *Commun. ACM* 68, 1 (2025), 33–35. doi:10.1145/3680410

Arun Shankar Iyer, Manohar Jonnalagedda, Suresh Parthasarathy, Arjun Radhakrishna, and Sriram K. Rajamani. 2019. Synthesis and machine learning for heterogeneous extraction. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 301–315. doi:10.1145/3314221.3322485

Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1.* 215–224. doi:10.1145/1806799.1806833

Ruyi Ji, Yican Sun, Yingfei Xiong, and Zhenjiang Hu. 2020. Guiding dynamic programing via structural probability for accelerating programming by example. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 224:1–224:29. doi:10.1145/3428292

Ruyi Ji, Jingtao Xia, Yingfei Xiong, and Zhenjiang Hu. 2021. Generalizable synthesis through unification. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–28. doi:10.1145/3485544

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186* (2018). https://arxiv.org/abs/1804.01186

Woosuk Lee. 2021. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28. doi:10.1145/3434335

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 436–449. doi:10.1145/3296979.3192410

Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenenbaum, and Stephen Muggleton. 2014. Bias reformulation for one-shot function induction. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014) (Frontiers in Artificial Intelligence and Applications, Vol. 263)*, Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan (Eds.). IOS Press, 525–530. doi:10.3233/978-1-61499-419-0-525

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

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

Stephen Muggleton and Wray L. Buntine. 1988. Machine Invention of First Order Predicates by Inverting Resolution. In *Machine Learning, Proceedings of the Fifth International Conference on Machine Learning, Ann Arbor, Michigan, USA, June 12-14, 1988*, John E. Laird (Ed.). Morgan Kaufmann, 339–352. doi:10.1016/b978-0-934613-64-4.50040-2

Stephen H. Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. 2015. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Mach. Learn.* 100, 1 (2015), 49–73. doi:10.1007/s10994-014-5471-y

Michael O'Neill. 2009. Riccardo Poli, William B. Langdon, Nicholas F. McPhee: A Field Guide to Genetic Programming. *Genet. Program. Evolvable Mach.* 10, 2 (2009), 229–230. doi:10.1007/s10710-008-9073-y

Saswat Padhi. 2019. SyGuS-Comp 2019. https://sygus.org/comp/2019/. Accessed: 2021-07-09.

Saswat Padhi, Todd D. Millstein, Aditya V. Nori, and Rahul Sharma. 2019. Overfitting in Synthesis: Theory and Practice. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 315–334. doi:10.1007/978-3-030-25540-4_17

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.).

ACM, 107–126. doi:10.1145/2814270.2814310

Ameesh Shah, Eric Zhan, Jennifer J. Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. 2020. Learning Differentiable Programs with Admissible Neural Heuristics. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/342285bb2a8cadef22f667eeb6a63732-Abstract.html

Yichen Yang, Jeevana Priya Inala, Osbert Bastani, Yewen Pu, Armando Solar-Lezama, and Martin C. Rinard. 2021. Program Synthesis Guided Reinforcement Learning for Partially Observed Environments. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). 29669–29683. https://proceedings.neurips.cc/paper/2021/hash/f7e2b2b75b04175610e5a00c1e221ebb-Abstract.html

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2021. Optimal Neural Program Synthesis from Multimodal Specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 1691–1704. doi:10.18653/V1/2021.FINDINGS-EMNLP.146

Zhentao Ye, Ruyi Ji, Xin Zhang, and Yingfei Xiong. 2025. *DSL Optimization Artifact for POPL 2026: Accelerating Syntax-Guided Program Synthesis by Optimizing Domain Specific Languages.* doi:10.5281/zenodo.17345861

Mohammed J Zaki. 2002. Efficiently mining frequent trees in a forest. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. 71–80. doi:10.1145/775047.775058

Received 2025-07-10; accepted 2025-11-06