Improving Oracle-Guided Inductive Synthesis by Efficient Question Selection

RUYI JI, Peking University, China
CHAOZHE KONG, Peking University, China
YINGFEI XIONG*, Peking University, China
ZHENJIANG HU, Peking University, China

Oracle-guided inductive synthesis (OGIS) is a widely-used framework to apply program synthesis techniques in practice. The question selection problem aims at reducing the number of iterations in OGIS by selecting a proper input for each OGIS iteration. Theoretically, a question selector can generally improve the performance of OGIS solvers on both interactive and non-interactive tasks if it is not only effective for reducing iterations but also efficient. However, all existing effective question selectors fail in satisfying the requirement of efficiency. To ensure effectiveness, they convert the question selection problem into an optimization one, which is difficult to solve within a short time.

In this paper, we propose a novel question selector, named *LearnSy*. *LearnSy* is both efficient and effective and thus achieves general improvement for OGIS solvers for the first time. Since we notice that the optimization tasks in previous studies are difficult because of the complex behavior of operators, we estimate these behaviors in *LearnSy* as simple random events. Subsequently, we provide theoretical results for the precision of this estimation and design an efficient algorithm for its calculation.

According to our evaluation, when dealing with interactive tasks, *LearnSy* can offer competitive performance compared to existing selectors while being more efficient and more general. Moreover, when working on non-interactive tasks, *LearnSy* can generally reduce the time cost of existing CEGIS solvers by up to 43.0%.

CCS Concepts: • **Software and its engineering → General programming languages**.

Additional Key Words and Phrases: Oracle-Guided Inductive Synthesis, Question Selection Problem

ACM Reference Format:
Ruyi Ji, Chaozhe Kong, Yingfei Xiong, and Zhenjiang Hu. 2023. Improving Oracle-Guided Inductive Synthesis by Efficient Question Selection. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 103 (April 2023), 29 pages. https://doi.org/10.1145/3586055

1 INTRODUCTION

Oracle-guided inductive synthesis (OGIS) [Jha and Seshia 2017] is a widely-used framework to apply program synthesis techniques in practice. A typical implementation of OGIS synthesizes programs by iteratively invoking a solver for programming-by-example (PBE) [Shaw et al. 1975], a

*Corresponding author

Authors' addresses: Ruyi Ji, Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University); School of Computer Science, Peking University, Beijing, China, jiruyi910387714@pku.edu.cn; Chaozhe Kong, Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University); School of Computer Science, Peking University, Beijing, China, kcz@pku.edu.cn; Yingfei Xiong, Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University); School of Computer Science, Peking University, Beijing, China, xiongyf@pku.edu.cn; Zhenjiang Hu, Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University); School of Computer Science, Peking University, Beijing, China, huzj@pku.edu.cn.

2475-1421/2023/4-ART103
https://doi.org/10.1145/3586055

question selector, and an oracle[1]. In each iteration, the PBE solver generates a candidate program, the question selector either accepts the candidate program as the synthesis result or selects an input among all available inputs, and the oracle completes the selected input into an input-output example, which will be provided to the PBE solver in the next iteration.

There are two major types of OGIS tasks, which correspond to interactive and non-interactive application scenarios.

- In interactive tasks, the human user acts as the oracle. In every iteration, the question selector shows an input to the user and asks for the intended output on this input. For these tasks, the performance of an OGIS solver is usually measured by the user burden during synthesis.
- In non-interactive tasks, the oracle is an executable that provides the output of the target program. In this case, the performance of an OGIS solver is usually measured by the time cost for synthesis.

For both types of tasks, the number of iterations used for synthesis is an important indicator of the performance of OGIS solvers. For interactive tasks, the number of iterations directly reflects the user burden. Fewer iterations correspond to fewer questions asked to the user and hence less user burden. For non-interactive tasks, the number of iterations reveals how many times each component in the OGIS solver is invoked during synthesis, which in turn determines the time cost of the OGIS solver. Therefore, to improve the performance of OGIS solvers, the problem of reducing the number of OGIS iterations has received much attention [Ji et al. 2020a, 2021; Kalyan et al. 2018; Singh and Gulwani 2015; Tiwari et al. 2020].

The *question selection problem* is proposed for a question selector that minimizes the number of iterations [Ji et al. 2020a; Tiwari et al. 2020]. It is motivated by the observation that the selection of inputs has a strong impact on the number of questions needed. For example, when only three programs $0, x, y$ are available, selecting the input as $\langle x = 1, y = 2\rangle$ is sufficient to help the synthesizer find the correct program because the output returned by the oracle must uniquely correspond to the target. On the contrary, selecting the input as $\langle x = 0, y = 0\rangle$ cannot bring any progress for synthesis as the output must be 0 whatever the correct program is.

Ji et al. [2020a] conducted a theoretical analysis of the question selection problem and reduced it to an optimization problem on the inputs. In this optimization problem, the objective function measures how an input distinguishes *remaining programs*, those programs satisfying all existing examples. In other words, the question selector should select an input where the outputs of the remaining programs vary as significantly as possible. For instance, when the remaining programs are $0, x, y$, the selector should prefer input $\langle x = 1, y = 2\rangle$, where all the remaining programs output differently, to input $\langle x = 0, y = 0\rangle$, where all the remaining programs output the same.

However, even reduced to an optimization problem, solving the question selection problem remains difficult, and even the evaluation of the objective function for a given input faces significant scalability challenges. A direct approach to measure how an input distinguishes the remaining programs needs to enumerate all of these programs, get the output for each program, and measure the diversity of all the outputs. It is rather impossible to finish this procedure within an acceptable timeframe since the program space in a synthesis task is usually extremely large.

To address the scalability issue, it is natural to consider evaluating the objective function approximately rather than precisely. Attempts have been made in this regard, but the results are still limited [Ji et al. 2020a; Tiwari et al. 2020]. These studies make approximations by sampling a small set of programs from all remaining ones and measuring the output diversity from this set only. This approximation could reduce the number of programs taken into consideration, but two main shortcomings remain:

[1]Here we use the triple-form OGIS studied by Ji et al. [2020a], where the selection of questions is in a separate component.

- **Low efficiency**. Theoretically, selecting a sample set from the remaining programs is a task even harder than PBE because this selection asks for multiple programs satisfying all examples, but PBE only needs to find one such program. Therefore, this approximation usually slows down OGIS solvers and thus can only be applied to interactive tasks.
  Meanwhile, such inefficiency would also lead to significant user inconvenience. To reduce the response time, Ji et al. [2020a] assume that the user needs a long time (2 minutes in their implementation) to answer each query, and then they use this period to perform background sampling. However, most existing interactive tasks include easy-to-answer queries. While processing these tasks, the user must wait until enough samples have been collected, resulting in an unacceptable response time.
- **Low generality**. Existing approaches [Ji et al. 2020a; Tiwari et al. 2020] rely on efficient witness functions, which only exist in specific synthesis domains, to perform sampling. Therefore, their usages are quite limited.

In this paper, we design an efficient approximation approach for the objective function and propose a question selector that achieves general improvement for OGIS solvers. Our approach has two remarkable advantages compared to existing approaches:

- **High Efficiency**. Our approach does not require extra remaining programs and thus does not involve extra synthesis tasks, which constitute the major time-consuming aspect of existing approaches. The time cost of our approach relates only to the size of the grammar specifying the program space and the number of iterations, both of which are usually small numbers. Consequently, our approach achieves general speedups for OGIS solvers on non-interactive tasks. It is the first question selector to realize this speedup to our knowledge.
- **High Generality**. Our approach uses only the semantics of programs and the grammar, making it applicable to all tasks under the syntax-guided synthesis framework [Alur et al. 2013] without relying on efficient witness functions.

To get an efficient approximation, we analyze the original objective function and find that the complex semantics of operators is the main challenge for evaluation. Based on this observation, our approximation is performed in three steps. Firstly, we propose the *unified-equivalence model* to approximate the behavior of operators with probabilities and estimate the likelihood of two programs outputting the same from a certain input. Secondly, we approximate the objective function with the model estimations. Finally, we find that the calculation of the approximated objective value can be decomposed into subtasks on sub-program spaces, thereby generating an efficient dynamic-programming algorithm.

We complete our approximation approach into a question selector named *LearnSy* in a generate-and-select manner. In each OGIS iteration, *LearnSy* generates a set of candidate inputs and selects the one with the best approximated objective value. We evaluate the performance of *LearnSy* on both interactive and non-interactive OGIS tasks.

- For interactive tasks, we apply *LearnSy* to the interactive synthesis framework proposed by Ji et al. [2020a] and evaluate it on 166 tasks collected by previous studies. The results show that *LearnSy* achieves competitive performance with *SampleSy* [Ji et al. 2020a], a state-of-the-art question selector for interactive synthesis, while requiring a shorter sampling period and not asking for witness functions of related operators.
- For non-interactive tasks, we apply *LearnSy* to the counter-example guided inductive synthesis (CEGIS) framework [Solar-Lezama et al. 2006] and compare it with (1) the default selector in CEGIS, and (2) two manually-designed domain-specific heuristics on input selection discussed in previous studies on PBE solvers [Jha et al. 2010; Padhi et al. 2018]. We combine these selectors with three state-of-the-art PBE solvers and evaluate them on 755 tasks in 3

different domains. The results show that *LearnSy* can reduce the time cost of all considered CEGIS solvers by up to 43.0%.

2 OVERVIEW

In this section, we will explain the main idea of our approach with a motivating example. This example will also be used throughout this paper.

2.1 The Motivating Example

This example includes a sample synthesis task, an oracle, and a PBE solver. In this example, we assume the question selector always forms an OGIS solver with the given oracle and the given PBE solver, and the OGIS solver is applied to solve the sample task.

Sample task. We consider the task of synthesizing $x + y$ from the following grammar G_e.

$$S \coloneqq 1 \mid x \mid y \mid T + T \quad T \coloneqq 1 \mid x \mid y$$

There are 12 programs in this grammar. We list them below in ascending order by size.

$$1, x, y, 1 + 1, 1 + x, 1 + y, x + 1, x + x, x + y, y + 1, y + x, y + y \tag{1}$$

To simplify the task, we assume that variables x and y only take values within the set $\{0, 1, 2\}$. Therefore, there are only 9 possible inputs.

Oracle. Given any input, the oracle provides the corresponding output of the target program $x + y$.

PBE solver. In this example, we consider an enumeration-based PBE solver. When given a set of input-output examples, this PBE solver will enumerate programs in G_e following Order 1 and return the first program that satisfies all given examples.

Finally, we assume that the question selector includes a verifier as a component and accepts the PBE result as final when it is equivalent to $x + y$.

2.2 The Min-Pair Strategy for Question Selection

The question selector is a significant factor to affect the number of iterations needed for synthesis. A naive way to select the input is to find a counterexample where the PBE result gives an incorrect output and thus ensure at least one incorrect program is excluded after each iteration. However, this counterexample may overfit the PBE result, resulting in a number of iterations as large as the size of the program space. Fortunately, appropriate input selection can help reduce the number of iterations. In our sample synthesis task, the target program $x + y$ will be found within three iterations if input $\langle x = 0, y = 2\rangle$ and $\langle x = 1, y = 0\rangle$ are selected in the first two iterations. This number of iterations (3) is much smaller than the size of the program space (12).

The question selection problem is about minimizing the number of iterations by properly selecting the inputs. Ji et al. [2020a] connect this problem to the optimal decision tree problem and prove the effectiveness of a series of greedy strategies in solving the question selection problem. In this paper, we consider one of these strategies, namely *min-pair* [Adler and Heeringa 2012; Chakaravarthy et al. 2011]. In this section, we would only explain a simplified version of the min-pair strategy and leave the detailed discussion in Section 3.2. In the simplified min-pair strategy, the objective value of an input is the number of pairs in the remaining programs (those satisfying all existing examples) that provide the same output from the given input. In each iteration, this strategy always selects the input with the smallest objective value.

Table 1 shows the synthesis procedure with the min-pair strategy, where $\langle a, b\rangle[w]$ represents an input $\langle x = a, y = b\rangle$ with an objective value of w. For example, in the second iteration, only four programs $y, 1 + 1, x + y$ and $y + x$ satisfy the previous example $\langle 0, 2\rangle \mapsto 2$, and their outputs on input $\langle 1, 0\rangle$ are $0, 2, 1$ and 1 respectively. Therefore, the objective value of input $\langle 1, 0\rangle$ in the

Table 1. The synthesis procedure with the min-pair strategy.

Turn	PBE Result	Remaining Programs	Inputs	Example
1	1	all programs in G_e	$\langle 0,0\rangle$[62], $\langle 0,1\rangle$[56], $\langle 0,2\rangle$[34] $\langle 1,0\rangle$[56], $\langle 1,1\rangle$[90], $\langle 1,2\rangle$[46] $\langle 2,0\rangle$[34], $\langle 2,1\rangle$[46], $\langle 2,2\rangle$[42]	$\langle 0,2\rangle \mapsto 2$
2	y	$y, 1+1, x+y, y+x$	$\langle 0,0\rangle$[10], $\langle 0,1\rangle$[10], $\langle 0,2\rangle$[16] $\langle 1,0\rangle$[6], $\langle 1,1\rangle$[10], $\langle 1,2\rangle$[8] $\langle 2,0\rangle$[10], $\langle 2,1\rangle$[6], $\langle 2,2\rangle$[8]	$\langle 1,0\rangle \mapsto 1$
3	$x+y$			

second iteration is 6, representing four pairs of the same program and two pairs of $x+y$ and $y+x$ in different orders. According to Table 1, the min-pair strategy can finish the synthesis with merely three iterations and thus achieves the optimal selection.

2.3 The Approximation of the Objective Function in the Min-Pair Strategy

Despite that the min-pair strategy is effective in reducing iterations, its practical application is difficult. The direct evaluation of its objective function needs all programs in the program space to be enumerated and to have their outputs calculated. Such a procedure is so time-consuming that makes it is practically not feasible because the size of the program space is usually extremely large. Therefore, we intend to propose an efficient approximation of the objective function of the min-pair strategy.

Main idea. In this approximation, we assume the program space is specified by a grammar. Such a program space is *compositional*. Each non-terminal in the grammar specifies a sub-program space, and these sub-program spaces make up the whole program space according to the grammar rules.

To enable an efficient calculation on such a program space, our approximation should be compositional, too; In other words, its results should be calculable based on the results of sub-program spaces. In this case, we need only to process each sub-program space once and do not have to consider their combinations. This procedure is efficient since the number of sub-program spaces (i.e., the grammar size) is usually small in practice.

The objective function of the min-pair strategy is not compositional due to the complex behavior of operators. To see this point, let us consider a special case, where (1) there is no existing example, and (2) only the 9 programs expanded from $S \to T + T$ are considered. In this case, the objective value of an input is the number of pairs of programs expanded from $S \to T + T$ that output the same on the given input, and the task on the sub-program space is counting the number of pairs of sub-programs expanded from (T, T) that output the same. However, this sub-result is not enough to calculate the objective value. For those pairs whose sub-programs give different outputs, their final outputs may still be the same as different integer pairs may have an identical sum.

To enable a compositional approximation, our approach estimates the complex behavior of operators with probabilities. In the special case discussed above, our approach will approximate $+$ as a random function and assume that, for any two pairs of different integers, the probability for the random $+$ to output the same is always a fixed constant c. Subsequently, the objective value can be estimated as the expected number of program pairs that output the same under the approximated operator. This approximation is compositional since the subtask here is to calculate the expected number of sub-program pairs that output the same. When the subtask results in w, the approximation can be calculated as $w + c(81 - w)$, as (1) there are 81 program pairs in total, and (2) each pair where sub-programs output differently has a probability of c to give the same final output.

Table 2. A unified equivalence model on grammar G_e.

(a) Assignments to *self*.

Rule	Value
$S \to T + T$	5/36
$S \to x$	0
...	...

(b) Assignments to *cross*

Rule	$S \to T + T$	$S \to x$	...
$S \to T + T$		1/9	
$S \to x$	1/9		
...	...		...

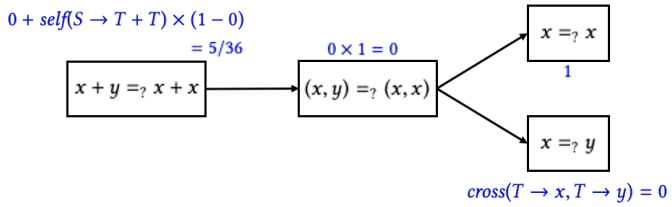

Fig. 1. The procedure for the model in Table 2 to estimate the check between $x + y$ and $x + x$.

Dealing with examples. Our approximation is straightforward when dealing with more general cases. When examples are available, the objective function will only consider programs satisfying all these examples. The approach to process the special case cannot deal with such constraint because it only distinguishes whether the outputs of two programs are the same without considering the concrete outputs of programs.

To go through this issue, we make the following two observations.

- For any two concrete programs, their probability to output the same can be estimated after estimating the behavior of operators with probabilities.
- A PBE result, a program satisfying all examples, is available in each OGIS iteration.

Therefore, our approach calculates the probability for each program to satisfy all examples as the probability for the program to output the same as the PBE result on all examples (using a similar approach as in the special case). In this way, we get a compositional approximation of the objective value for general cases.

Unified-equivalence model. Next, we will elaborate on how our approach estimates the probability for two programs to output the same. This estimation is accomplished via a *unified-equivalence model* and a compositional method similar to the method to process the special case.

A unified-equivalence model is bound to a grammar and a concrete input; in other words, it is specific for estimating the probability for two programs in this grammar to output the same under this input. Table 2 demonstrates a unified-equivalence model for grammar G_e and input $\langle 0, 2 \rangle$. This model includes two series of parameters.

- Series *self* deals with the case we met in the previous special case. For each grammar rule r, $self(r)$ estimates the probability for two programs expanded from r to output the same when their sub-programs output differently.
- Series *cross* deal with the other case where two programs are expanded from different grammar rules. For each pair of different rules (r, r'), parameter $cross(r, r')$ estimates the probability for two programs (one expanded from r, the other from r') to output the same.

This unified-equivalence model can be used to recursively estimate whether two concrete programs output the same. Figure 1 shows the procedure to estimate the probability for $x + y$ and $x + x$ to output the same, where each rectangle represents a subtask, $p_1 =_? p_2$ represents an

estimation task for programs p_1 and p_2, and blue expressions describe the calculation. To estimate for $x + y$ and $x + x$, the model first estimates the probability for their sub-programs, (x, y) and (x, x), to output the same, where the result is 0. In the other case, the model regards the probability for $x + y$ and $x + x$ to output the same as $\mathit{self}(S \to T + T) = 5/36$. Thus, the model estimation becomes $0 + 5/36 \times (1 - 0) = 5/36$.

Based on the unified-equivalence model, our approximation of the objective value in the min-pair strategy is defined in three steps. First, all pairs of concrete programs in the program space are enumerated. Then, unified-equivalence models are used to estimate the probability for each pair to satisfy all examples and to output the same. Finally, the estimations for all pairs are added up, and the sum is taken as the result. This estimation is compositional, as shall be shown in Section 5.3, and intuitively, it is because (1) the program space is compositional, and (2) the estimation of unified-equivalence models proceeds recursively according to the structure of programs.

Learning unified-equivalence models. Estimations provided by unified-equivalence models rely on the parameters, hence a crucial problem is how to learn proper assignments so that high-quality estimations are enabled. Therefore, we first define a series of ideal models whose precisions are guaranteed in theory, namely *natural unified-equivalence models*. Following that, we design a sampling-based learning algorithm to efficiently generate a model close to the ideal one.

Every unified-equivalence model in our approach is specific for estimating the equivalence between programs in a fixed grammar under a fixed input. With a certain grammar and an input, our learning algorithm learns a corresponding model in three steps. Firstly, it samples several program pairs from the program space. Secondly, it evaluates these program pairs on the given input and checks whether the outputs are the same for each pair. Finally, it calculates proper parameters from the check results. This algorithm is efficient, and thus we learn models on demand while selecting inputs, i.e., our approach learns a new model whenever a new input is involved.

It is worth noting that the unified-equivalence model is not designed to estimate whether two programs output the same precisely; instead, its design purpose is to simplify the behavior of operators and thus enable a compositional approximation. Even so, we prove that the learned unified-equivalence model is precise in estimating the overall equivalence, providing an unbiased estimation of the number of program pairs that output the same (Theorem 4.8). Moreover, we provide an operator named *flattening* to further improve the precision of our model on every single estimation (Theorem 4.11).

2.4 Generation and Selection in *LearnSy*

In this paper, we propose a question selector named *LearnSy* that runs in a generate-then-select manner based on our approximation (Algorithm 1). It generates a set of candidate inputs (Line 1) and selects the best input among them with the approximated objective values (Lines 2-9).

The generation process depends on the synthesis task. When a verifier is available, the CEGIS framework [Solar-Lezama et al. 2006] can be applied to generate inputs where p_c outputs incorrectly. If such a verifier is unavailable, the framework of interactive program synthesis [Le et al. 2017] can be applied to generate inputs that can distinguish at least two remaining programs. In some cases, the number of different possible inputs can be large. Therefore, an upper limit is set on the number of candidate inputs to control the time cost in our implementation.

The selection process is straightforward. *LearnSy* learns a unified-equivalence model for each related input (Lines 3, 5), calculates the approximated objective value for each candidate input (Line 6), and selects the input with the smallest approximated objective value as the result (Line 7).

A verifier is available in our motivating example, so the CEGIS framework can be used to generate candidate inputs. At this time, *LearnSy* selects the input with the smallest approximated objective

Algorithm 1: The procedure of *LearnSy*.

Input: A context-free grammar G, an input space $\mathbb{I}$, a candidate program p_c, and a set *examples* of existing examples.
Output: The selected input or symbol $\perp$ representing p_c is accepted.

```
candidateInps ← GetCandidateInputs(𝕀, p_c);
bestObj ← +∞;  bestInp ← ⊥;
exampleModels ← {Learn(G, inp) | (inp ↦ oup) ∈ examples};
foreach inp ∈ candidateInps do
    model ← Learn(G, inp);
    obj ← Approximate(G, model, exampleModels);
    if obj < bestObj then (bestObj, bestInp) ← (obj, inp);
end
return bestInp;
```

value among those inputs where the PBE result outputs other than $x + y$. For this task, *LearnSy* is as efficient as the min-pair strategy (i.e., synthesis is finished in three iterations) given a large enough number of samples for learning.

3 THE MIN-PAIR STRATEGY

Now, we introduce some necessary notations and the details of the min-pair strategy. Since this paper only focuses on the approximation of the min-pair strategy, we will keep other related concepts such as the OGIS framework [Jha and Seshia 2017] and the question selection problem [Ji et al. 2020a] informal. In this paper, these concepts are used as described in previous literature.

3.1 Preliminaries

A program synthesis task is usually discussed based on an underlying domain $\mathbb{D} = (\mathbb{P}, \mathbb{I}, \mathbb{O}, [\![\cdot]\!]_{\mathbb{D}})$, where $\mathbb{P}, \mathbb{I}$, and $\mathbb{O}$ represent a program space, an input space, and an output space, respectively. $[\![\cdot]\!]_{\mathbb{D}}$ is an oracle function that associates a program $p \in \mathbb{P}$ and an input $I \in \mathbb{I}$ with an output in $\mathbb{O}$, denoted as $[\![p]\!]_{\mathbb{D}}(I)$. The domain limits the scope of possible programs and concerned inputs and provides the corresponding program semantics.

In this paper, we make two assumptions on the underlying domain. First, we assume that there is a universal output space and a universal oracle function $[\![\cdot]\!]$ for all domains. Therefore, a domain can be abbreviated as a pair $(\mathbb{P}, \mathbb{I})$, in which only a program space and an input space are involved.

Second, following the framework of syntax-guided synthesis [Alur et al. 2013], we assume that $\mathbb{P}$ is specified by a *regular-tree grammar (RTG)*, where each program is expanded according to its syntax tree.

Definition 3.1 (RTG). An RTG is a tuple $G = \langle N, \Sigma, s_0, R \rangle$ where:

- N is a finite set of non-terminals. s_0 is an element in N, representing the start non-terminal.
- Σ is a ranked alphabet, where each symbol f is attached with an arity k, written as $f^{(k)}$.
- R is a finite set of grammar rules. Each rule in R is in the form of $s \to f^{(k)}(s_1, \ldots, s_k)$, where $s, s_1, \ldots, s_k$ are non-terminals in N and $f^{(k)}$ is a symbol in Σ.

A program is in G if it can be obtained by expanding s_0 following a finite number of grammar rules.

Similar to existing question selectors [Ji et al. 2020a; Tiwari et al. 2020], our approach requires the program space $\mathbb{P}$ to be finite, i.e., the RTG specifying $\mathbb{P}$ to be acyclic. This requirement is not critical because an infinite program space can always be truncated into a finite one in practice (by

setting a size limit large enough, for example). In our approach, we provide an iterative method to adapt the truncation on demand, which will be discussed in Section 6.

In the following sections, we will use a term *equivalence check* to denote a predicate in the form of $[\![p]\!](I) = [\![p']\!](I)$, which checks whether programs p and p' output the same on input I. This predicate will be frequently used in our approach.

3.2 The Min-Pair Strategy

Adapted from a state-of-the-art approach for the optimal decision tree problem [Adler and Heeringa 2012; Chakaravarthy et al. 2011], the min-pair strategy specifies a question selector in the OGIS framework, and its effectiveness is ensured by the bijection between the optimal decision tree problem and the question selection problem [Ji et al. 2020a].

Intuitively, to minimize the number of iterations, the selected input should exclude as many programs as possible in each turn. Based on this intuition, the min-pair strategy assumes that there is a prior distribution over the program space that represents the probability for each program to be the target. Following that, the strategy aims at minimizing the probability for a random program to be remaining after selecting the input.

Definition 3.2 (Min-Pair). Given an underlying domain $(\mathbb{P}, \mathbb{I})$, a distribution φ over $\mathbb{P}$, and a set E of existing input-output examples, the min-pair strategy selects an input I from $\mathbb{I}$ that minimizes the following objective function.

$$obj_0[\mathbb{P}, \varphi, E](I) = \Pr_{p,p^*\sim\varphi}\left[p \in remain\Big(\mathbb{P}, E \cup \{I \mapsto [\![p^*]\!](I)\}\Big) \;\middle|\; p^* \in remain(\mathbb{P}, E)\right]$$

where (1) $\Pr[C_1|C_2]$ represents the conditional probability, (2) p represents a random program, and p^* represents the target program, (3) $I \mapsto [\![p^*]\!](I)$ represents the new example when the target program is p^* and the selected input is I, and (4) $remain(\mathbb{P}, E)$ represents the set of programs in $\mathbb{P}$ that satisfy all examples in E. This objective function calculates the probability for a random program p to satisfy all examples after input I is selected.

This function can be transformed into the following equivalent form, where $\varphi(p)$ represents the probability of program p in φ, and $[\cdot]$ is a function mapping *true* to 1 and *false* to 0.

$$obj_1[\mathbb{P}, \varphi, E](I) = \sum_{p,p'\in\mathbb{P}} \varphi(p)\varphi(p')\Big[p, p' \in remain(\mathbb{P}, E)\Big]\Big[[\![p]\!](I) = [\![p']\!](I)\Big]$$

Example 3.3. The simplified form of the min-pair strategy discussed in Section 2, where the objective function is the number of pairs of remaining programs that output the same, is equivalent to the above function when φ is a uniform distribution over $\mathbb{P}$.

When taking the PBE result p_c into consideration, the transformed objective function can be expressed using only equivalence checks. It is easy to prove that the following function is equal to obj_1 when p_c satisfies all examples in E, where $eq(p, p', I)$ is an abbreviation of $[[\![p]\!](I) = [\![p']\!](I)]$, the result of an equivalence check.

$$obj[\mathbb{P}, \varphi, E, p_c](I) = \sum_{p,p'\in\mathbb{P}} \varphi(p)\varphi(p')eq(p, p', I) \prod_{(I'\mapsto O')\in E} \Big(eq(p_c, p, I')eq(p, p', I')\Big) \tag{2}$$

Since we assume that $\mathbb{P}$ is specified by a regular-tree grammar, for efficient probability assessments, we further assume that φ is also defined over the grammar, in the form of *probabilistic regular-tree grammar (PRTG)*.

Definition 3.4 (Probablistic Regular-Tree Grammar (PRTG)). A PRTG is an RTG $\langle N, \Sigma, s_0, R\rangle$ combined with a function $\gamma : R \mapsto [0, 1]$ that assigns a probability to each rule. γ needs to ensure

the sum of the probabilities of the rules starting from the same non-terminal to be exactly 1, i.e., $\sum_{r \in R(s)} \gamma(r) = 1$ for every non-terminal s in N, where $R(s)$ representing the rules starting from s.

In a PRTG, the probability assigned to a grammar rule r denotes the probability of choosing rule r to expand non-terminals. Therefore, for a program expanded by rules $r_1, \ldots, r_n$, its probability is defined as $\prod_{i=1}^{n} \gamma(r_i)$.

Example 3.5. The following example shows the PRTG specifying a uniform distribution over grammar G_e[2] in Section 2, where $r[w]$ denotes the probability assigned to rule r is w.

$$S \coloneqq 1\ [1/12] \mid x\ [1/12] \mid y\ [1/12] \mid T + T\ [3/4] \quad T \coloneqq 1\ [1/3] \mid x\ [1/3] \mid y\ [1/3]$$

In this PRTG, the probability of $x + y$ is equal to $\gamma(S \to T + T)\gamma(T \to x)\gamma(T \to y) = 1/12$.

4 UNIFIED-EQUIVALENCE MODEL

In this section, we will introduce the formal definition of the unified-equivalence model (Section 4.1), show a learning algorithm for this model (Section 4.2 and 4.3), and finally introduce the flattening operation, which can improve the precision of our model (Section 4.4).

4.1 Unified-Equivalence Model

The unified-equivalence model approximates a recursive procedure *receq* for evaluating equivalence checks, in which the syntax of programs is used when making decisions. Given programs p, p' and input I, *receq* evaluates the predicate $[\![p]\!](I) = [\![p']\!](I)$ as follows.

$$receq(p, p', I) = \begin{cases} [\![p]\!](I) = [\![p']\!](I) & r \neq r' \\ \text{true} & r = r' \wedge \left(\bigwedge_{i=1}^{n} receq(p_i, p_i', I)\right) \\ [\![p]\!](I) = [\![p']\!](I) & \text{Otherwise} \end{cases}$$

where r is the first grammar rule used by p; n is the arity of the symbol in r, $p_1, \ldots, p_n$ are the sub-programs of p; and $r', n', p_1', \ldots, p_{n'}'$ are the counterparts related to program p'. In the second case, *receq* employs the fact that p and p' must output the same when their sub-programs output the same because the semantics of each operator must be a function.

Lemma 4.1 (Correctness of *receq*). *For any two programs p, p' and any input I, $receq(p, p', I)$ results in true if and only if $[\![p]\!](I) = [\![p']\!](I)$.*

Due to the space limit, we move all proofs to the appendix [Ji et al. 2023a].

In the definition of *receq*, two cases still rely on the semantics of operators. To avoid such reliance, the unified-equivalence model approximates these cases with Bernoulli trials by involving two series of parameters named *self* and *cross*. These parameter series represent the probability for the model to return *true* when approximating the first and the third cases of *receq*, respectively.

Definition 4.2 (Unified-Equivalence Model). Given an RTG G with rule set R, a unified-equivalence model $\mathcal{M}$ is specified by two functions $self : R \mapsto [0, 1]$ and $cross : R \times R \mapsto [0, 1]$, which assign probabilities to each grammar rule and each pair of different grammar rules, respectively.

A unified-equivalence model $\mathcal{M}$ induces an estimation function $\mathcal{M}(p, p')$ that estimates how likely an equivalence check between programs p and p' results in true.

$$\mathcal{M}(p, p') = \begin{cases} cross(r, r') & r \neq r' \\ w + (1 - w)self(r), \text{where } w = \prod_{i=1}^{n} \mathcal{M}(p_i, p_i') & r = r' \end{cases}$$

[2]Precisely speaking, G_e is not an RTG unless its rule $S \to T + T$ becomes $S \to +^{(2)}(T, T)$. Here, we keep the original definition of G_e for simplicity.

where r is the first grammar rule used by p; n is the arity of the symbol in r; $p_1, \ldots, p_n$ are the sub-programs of p; and $r', n', p'_1, \ldots, p'_{n'}$ are the counterparts related to program p'. In this definition, the blue and green parts correspond to the second case and the third case in *receq*, respectively, and w represents the probability for sub-programs to output the same.

Example 4.3. Figure 1 shows how a unified-equivalence model estimates the equivalence check between $x + y$ and $x + x$. The symbolic form of the estimation is given below.

$$w + (1 - w)\mathit{self}(S \to T + T), \textbf{where } w = \mathit{cross}(T \to x, T \to y)$$

4.2 Natural Unified-Equivalence Model

Given a PRTG and an input, our learning algorithm learns a unified-equivalence model specific for equivalence checks (1) between programs drawn from the PRTG and (2) on the given input. Our algorithm is designed in two steps. First, a subclass of unified-equivalence models named *natural unified-equivalence model* is highlighted, because its precision is guaranteed in theory. Then, an algorithm to efficiently learn a model similar to the natural model is developed.

In a natural unified-equivalence model, the value of each parameter precisely matches its effect in estimation. According to Definition 4.2, each parameter in the unified-equivalence model has a clear role in the estimation of equivalence checks.

- Parameter $\mathit{self}(r)$ estimates the probability for two programs to output the same under the conditions that (1) both programs are expanded from rule r, and (2) the sub-programs of the two programs output differently.
- Parameter $\mathit{cross}(r_1, r_2)$ estimates the probability for two programs to output the same when the two programs are expanded from rules r_1 and r_2, respectively.

We found that given any distribution over the program space and a concrete input, the probability estimated by each parameter is well-defined and calculable. For example, the probability estimated by parameter $\mathit{cross}(r_1, r_2)$ is the probability of event $[\![p_1]\!](I) = [\![p_2]\!](I)$, where I is the given input, and $p_i (i \in \{1, 2\})$ is a random program drawn from all programs expanded from rule r_i according to the given distribution. Therefore, a natural idea to get an effective unified-equivalence model is to set each parameter to its estimated probability. Such an assignment is named as the *natural assignment* to a parameter, and the model where each parameter is set to its natural assignment is named the *natural unified-equivalence model*.

Definition 4.4 (Natural Unified-Equivalence Model). Given a PRTG φ and an input I, the natural unified-equivalence model for I and φ, denoted as $\tilde{\mathcal{M}}^I_\varphi$, is a unified-equivalence model on the grammar of φ, and each parameter is set as follows.

$$\mathit{self}(r) = \Pr_{p, p' \sim \varphi(r)} \left[[\![p]\!](I) = [\![p']\!](I) \mid \exists i, [\![p_i]\!](I) \neq [\![p'_i]\!](I) \right] \tag{3}$$

$$\mathit{cross}(r, r') = \Pr_{p \sim \varphi(r), p' \sim \varphi(r')} \left[[\![p]\!](I) = [\![p']\!](I) \right] \tag{4}$$

where p_i and p'_i represent the i-th sub-program of p and p' respectively, $p \sim \varphi(r)$ represents that p is a random program drawn from those programs expanded from rule r in which the probability for each program to be selected is directly proportional to its probability assigned by φ.

Example 4.5. Table 2 describes a part of a unified-equivalence model, which in fact corresponds to the natural unified-equivalence model for input $\langle 2, 0 \rangle$ and the PRTG discussed in Example 4.3, i.e., a uniform distribution over grammar G_e. In this example, we explain the calculation for the natural assignment of $\mathit{self}(S \to T + T)$.

Algorithm 2: The learning algorithm `Learn` in Algorithm 1.

Input: A PRTG φ with rule set R, an input I, and two constants $n_s, w_{default}$.
Output: A unified-equivalence model learned for φ and I.

```
1  self ← {};  cross ← {};
2  foreach r ∈ R do
3      (p_1, ..., p_{n_s}) ← Sample(r, φ, n_s);  (p'_1, ..., p'_{n_s}) ← Sample(r, φ, n_s);
4      n_1 ← the number of i ∈ {1, ..., n_s} such that IsSubDiff(p_i, p'_i, I);
5      n_2 ← the number of i ∈ {1, ..., n_s} such that IsSubDiff(p_i, p'_i, I) and [[p_i]](I) = [[p_i]](I');
6      if n_1 > 0 then self(r) ← n_2/n_1 else self(r) ← w_default;
7  end
8  foreach r, r' ∈ R do
9      (p_1, ..., p_{n_s}) ← Sample(r, φ, n_s);  (p'_1, ..., p'_{n_s}) ← Sample(r', φ, n_s);
10     cross(r, r') ← (the number of i ∈ {1, ..., n_s} such that [[p_i]](I) = [[p'_i]](I))/n_s;
11 end
12 return (self, cross);
```

In a unified-equivalence model, $self(S \to T+T)$ is used when the two programs are both expanded from $S \to T + T$ but their sub-programs output differently. There are 9 programs expanded from this rule, and their probabilities are all the same on the given PRTG. More specifically, there are 1, 2, 3, 2 and 1 programs that output 0, 1, 2, 3 and 4 on input $\langle 2, 0\rangle$, respectively. These programs' sub-programs produce different outputs varying from $(0, 0)$ to $(2, 2)$. Therefore, there are 72 program pairs whose sub-programs output differently under input $\langle 2, 0\rangle$, in which 10 of such pairs output the same. As a result, the natural assignment of $self(S \to T + T)$ is $10/72 = 5/36$.

The natural unified-equivalence model **precisely** captures the overall equivalence between programs on its PRTG and its input (Lemma 4.6). When two programs in the equivalence check are random, the expected estimation given by the natural unified-equivalence model is always equal to the ground truth, the probability for the two random programs to output the same under the corresponding input.

Lemma 4.6. *For any input I and any PRTG φ, the corresponding natural unified-equivalence model $\tilde{\mathcal{M}}^I_\varphi$ always satisfies the equation below.*

$$\mathop{\mathbb{E}}_{p,p'\sim\varphi}\left[\tilde{\mathcal{M}}^I_\varphi(p,p')\right] = \Pr_{p,p'\sim\varphi}\left[[\![p]\!](I) = [\![p']\!](I)\right]$$

4.3 Learning a Unified-Equivalence Model

Learning a natural unified-equivalence model is difficult in practice. According to Formula 3 and 4, the natural assignments rely on the outputs of all related programs, thereby presenting a similar challenge to that of evaluating the objective function of the min-pair strategy.

Our learning algorithm approximates the natural-unified equivalence model by sampling. Algorithm 2 shows the pseudocode of the learning algorithm, where *self* and *cross* are implemented as Python-styled dictionaries, n_s represents the number of samples, and $w_{default}$ represents the default value of the parameters. There are two functions in Algorithm 2.

- `Sample`(r, φ, n_s) draws n_s samples from programs expanded from rule r, where the probability for a program to be selected is directly proportional to its probability assigned by φ.
- `IsSubDiff`(p, p', I) checks whether the sub-programs of p and p' output differently under input I; in other words, this function returns $\vee_i [\![p_i]\!](I) \neq [\![p'_i]\!](I)$, where $p_i(p'_i)$ represent the i-th sub-program of $p(p')$.

In brief, for each parameter, Algorithm 2 approximates its natural assignment by generating n_s pairs of random programs and calculating the corresponding probability on the samples only. This algorithm is efficient because, under a fixed number of samples, its time cost is polynomial to the grammar size (a small number generally). Therefore, this algorithm is capable to learn models on demand while selecting inputs without any obvious efficiency loss.

Our learning algorithm can provide an unbiased estimation for the natural unified-equivalence model (Lemma 4.7). In other words, the expected estimation given by the learned model is always equal to the estimation from the natural unified-equivalence model.

Lemma 4.7. *For any input I and any PRTG φ, the equation below holds for every two programs p, p'.*

$$\mathbb{E}[\mathcal{M}(p,p')] = \tilde{\mathcal{M}}^I_\varphi(p,p'), \textbf{\textit{where}}\ \mathcal{M} = \texttt{Learn}(\varphi, I)^3$$

where the randomness comes from the random samples used in `Learn`.

Based on the lemmas provided above, we prove that for any natural unified-equivalence model, the learned model provides an unbiased estimation of the real probability for two random programs to output the same (Theorem 4.8).

Theorem 4.8. *For any input I and any PRTG φ, the following equation is always satisfied.*

$$\mathop{\mathbb{E}}_{p,p'\sim\varphi}[\mathcal{M}(p,p')] = \Pr_{p,p'\sim\varphi}\left[[\![p]\!](I) = [\![p']\!](I)\right], \textbf{\textit{where}}\ \mathcal{M} = \texttt{Learn}(\varphi, I)$$

where the randomness on the left-hand side also includes the random samples used in `Learn`.

4.4 Improving the Precision by Flattening

Despite that the precision of the learned model is guaranteed to some extent (Theorem 4.8), the estimations provided by the model are not completely accurate in many cases. In this section, we introduce an operator named *flattening* on the program space and the PRTG to improve the precision of unified-equivalence models.

The flattening operator transforms the program space by flattening a selected grammar rule into a series of 0-arity rules while keeping the probability of each program unchanged. Each of these 0-arity rules corresponds to a program expanded from the selected rule.

Definition 4.9 (Flattening Operation). Given a PRTG φ defined on grammar G, a flattening operation is specified by a grammar rule r^* in G and transforms φ into a new PRTG with the following procedure.

(1) Let s be the start non-terminal of rule r^* and P be the set of programs expanded from rule r^*.
(2) The set of non-terminals and the start non-terminal are kept unchanged.
(3) For each program $p \in P$, a 0-arity symbol $\square_p$ is added with the same semantics as p.
(4) Rule r^* is removed, and for each program $p \in P$, rule $s \to \square_p$ is added.
(5) The probabilities assigned to existing rules remain unchanged, but those to the new rules are set to the probability of the corresponding program in φ.

Example 4.10. Consider a program space specified by the following grammar G.

$$S \to S_1 + S_1 \mid S_1 - S_1 \qquad S_1 \to x \mid y$$

By flattening rule $S \to S_1 - S_1$ in G, another grammar G' can be obtained, as shown below.

$$S \to S_1 + S_1 \mid \square_{x-x} \mid \square_{x-y} \mid \square_{y-x} \mid \square_{y-y} \qquad S_1 \to x \mid y$$

[3]Here we assume the default value $w_{default}$ is never used as its usage corresponds to an unexpected case whose probability is negligible with a large enough number of samples.

Consider the equivalence check between $x+y$ and $x-y$ on input $\langle x=0, y=1\rangle$. The ground truth of this check is false, and the two estimations given from the two grammars are shown below. Since the calculations for these estimations are straightforward, the details are omitted for simplicity.

- Let $\mathcal{M}_1$ be the natural-unified equivalence model for grammar G, input $\langle x=0, y=1\rangle$, and the uniform distribution over the program space. In this model, $cross(S \to S_1 + S_1, S \to S_1 - S_1)$ is $1/4$, and thus estimation $\mathcal{M}_1(x+y, x-y)$ is also $1/4$, with an absolute error of $1/4$.
- Let $\mathcal{M}_2$ be the natural-unified equivalence model for grammar G', input $\langle x=0, y=1\rangle$, and the uniform distribution over the program space. In this model, $cross(S \to S_1 + S_1, S \to \square_{x-y})$ is 0, and thus estimation $\mathcal{M}_2(x+y, x-y)$ is also 0, which is accurate.

Therefore, in this case, the estimation given by the natural-unified equivalence model becomes more accurate following the flattening operation.

Theorem 4.11 generalizes the case study in Example 4.10, showing that the precision of the natural unified-equivalence model is never damaged following the flattening operation. Intuitively, the flattening operation reduces the average number of operations used by programs, and thus provides benefits to the precision of unified-equivalence models since errors in a unified-equivalence model come from approximating the behavior of operators.

Theorem 4.11. *Given two programs p and p', an input I, and a unified-equivalence model $\mathcal{M}$, define $same(p, p', I, \mathcal{M})$ as the probability for the estimation of $\mathcal{M}$ to be exactly the same as the recursive procedure $receq(p, p', I)$, that is, the result of each estimation happens in $\mathcal{M}(p, p')$ is equal to the ground truth in $receq(p, p', I)$[4]. Given an input I and a PRTG φ, define $acc(\varphi, I)$ as a function measuring the accuracy of the natural model $\tilde{\mathcal{M}}^I_\varphi$, as shown below.*

$$acc(\varphi, I) \coloneqq \mathop{\mathbb{E}}_{p,p'\sim\varphi}\left[same\left(p, p', I, \tilde{\mathcal{M}}^I_\varphi\right)\right]$$

For any input I, any PRTG φ, and any grammar rule r, $acc(\varphi_r, I)$ is always no smaller than $acc(\varphi, I)$, where φ_r represents the result of flattening grammar rule r in φ.

Besides, if the flattening operation is applied repeatedly, the grammar will ultimately become completely flattened[5], in which every program is flattened into a separate grammar rule. At this time, the estimation given by the natural unified-equivalence model will become absolutely accurate because the ground truth of every equivalence check is recorded into paramteter *cross*.

Example 4.12. Following Example 4.10, by further flattening rule $S \to S_1 + S_1$ in grammar G', a completely flattened grammar G'' can be obtained, as shown below.

$$S \to \square_{x+x} \mid \square_{x+y} \mid \square_{y+x} \mid \square_{y+y} \mid \square_{x-x} \mid \square_{x-y} \mid \square_{y-x} \mid \square_{y-y}$$

Let $\mathcal{M}_3$ be the natural unified-equivalence model for grammar G'', input $\langle x=0, y=1\rangle$, and the uniform distribution over the program space. For any two different programs p_1 and p_2, estimation $\mathcal{M}_3(p_1, p_2)$ is equal to $cross(S \to \square_{p_1}, S \to \square_{p_2})$. This parameter is equal to the ground truth of the equivalence check between p_1 and p_2 on $\langle x=0, y=1\rangle$, by the definition of natural unified-equivalence models.

It is worth noting that there is a trade-off between precision and efficiency in the usage of flattening. Flattening always increases the grammar size while improving the precision of the unified-equivalence models, which would lead to an undesired rise in the time cost.

[4]Due to the space limit, we omit the formal definition of *same* here, which can be found in the appendix [Ji et al. 2023a].
[5]Recall that the grammar is assumed to be acyclic in this section.

5 APPROXIMATION OF THE MIN-PAIR STRATEGY

In this section, we will introduce our approximation (Section 5.1) and propose an efficient algorithm for calculating the approximation (Section 5.2 and 5.3).

5.1 Approximated Objective Function

In our approximation, we apply unified-equivalence models to the min-pair strategy. The approximated objective function is obtained by replacing each equivalence check in Formula 2 with the estimation given by a learned unified-equivalence model (Definition 4.2). For example, equivalence check $eq(p, p', I)$ in Formula 2 is replaced with estimation $\mathcal{M}_I(p, p')$, where $\mathcal{M}_I$ denotes the unified-equivalence model learned for input I.

Definition 5.1 (Approximated Objective Function). Given domain $(\mathbb{P}, \mathbb{I})$, a PRTG φ over the program space, a set E of input-output examples, and a candidate program p_c satisfying all examples in E, the approximated objective function $appr[\mathbb{P}, \varphi, E, p_c](I)$ is defined as the following.

$$appr[\mathbb{P}, \varphi, E, p_c](I) = \sum_{p,p' \in \mathbb{P}} \varphi(p)\varphi(p')\mathcal{M}_I(p, p') \prod_{(I' \mapsto O') \in E} \Big(\mathcal{M}_{I'}(p_c, p)\mathcal{M}_{I'}(p, p')\Big)$$

where $\varphi(p)$ represents the probability of program p assigned by φ, and $\mathcal{M}_I$ denotes the unified-equivalence model learned for input I, i.e., $\texttt{Learn}(\varphi, I)$.

Example 5.2. Consider the second iteration of the synthesis procedure in Table 1, where the PBE result is y, and the only existing example is $\langle 0, 2\rangle \mapsto 2$. The approximated objective value of input $\langle 1, 0\rangle$ in this iteration when the given PRTG is φ can be expressed as follows:

$$\sum_{p,p' \in G_e} \varphi(p)\varphi(p')\mathcal{M}_{\langle 1,0\rangle}(p, p')\mathcal{M}_{\langle 0,2\rangle}(y, p)\mathcal{M}_{\langle 0,2\rangle}(p, p')$$

Our approximation is designed not for precision but for a compositional approximation to enable efficient question selection. Even so, the following two corollaries show that our approximation is precise in some aspects. Both of them are direct corollaries to the properties of unified-equivalence models discussed in Section 4.

In the first OGIS iteration where no input-output example is available, our approximation is unbiased. In this case, the original objective function measures the overall equivalence of two random programs on the given input, i.e., the probability for two random programs to output the same. Therefore, the property of the unbiased estimation is direct from Theorems 4.6 and 4.8.

Corollary 5.3. *For any program space $\mathbb{P}$, any PRTG φ over the program space, and any program $p_c \in \mathbb{P}$, the following equation always holds.*

$$\mathbb{E}\Big[appr[\mathbb{P}, \varphi, \emptyset, p_c](I)\Big] = obj[\mathbb{P}, \varphi, \emptyset, p_c](I)$$

where the randomness on the left-hand side comes from the random samples used in `Learn`.

In addition, our approximation is also precise when the program space is completely flattened. This is because errors in our approximation come from the loss of the estimations given by unified-equivalence models, but these estimations are accurate when the program space is completely flattened (Section 4.4).

Corollary 5.4. *For any completely flattened program space $\mathbb{P}_f$, any PRTG φ over $\mathbb{P}_f$, any set E of input-output examples, and any program p_c satisfying all examples in E, the following equation always holds.*

$$appr[\mathbb{P}_f, \varphi, E, p_c](I) = obj[\mathbb{P}_f, \varphi, E, p_c](I)$$

5.2 Two Examples of Calculating the Approximation

The compositionality of the approximations is used to calculate them efficiently. Our algorithm works in a divide-and-conquer manner by decomposing the task into subtasks on sub-program spaces, conquering each subtask recursively, and combining the sub-results. We find that the number of different subtasks in this procedure is bounded, and thus we speed up the calculation by dynamic programming, i.e., reusing results among identical subtasks.

For clarification, let us first consider two concrete cases on our sample program space G_e, in which two assumptions are made for simplicity.

- PRTG φ specifies a uniform distribution over G_e, i.e., it is the one discussed in Example 4.3.
- The learning algorithm always returns a dummy model $\mathcal{M}$, where all parameters are $1/2$.

Case 1: zero example. In the first OGIS iteration, there is no example, and thus the approximated objective value is $\sum_{p,p' \in G_e} \varphi(p)\varphi(p')\mathcal{M}(p,p')$. To calculate this expression, we divide the summation into cases according to the first grammar rules used by p and p', as shown below.

$$\varphi(1)\varphi(1)\mathcal{M}(1,1) + \cdots + \sum_{p_1+p_2, p_1'+p_2' \in G_e} \varphi(p_1+p_2)\varphi(p_1'+p_2')\mathcal{M}(p_1+p_2, p_1'+p_2')$$

where the first and the last terms correspond to the case where p and p' are both expanded from rule $S \to 1$ and $S \to T+T$ respectively. These terms are calculated separately, and then the approximated objective value is obtained by summing up the calculation results.

Each term is calculated by a three-staged procedure: (1) unfolding, (2) extraction of subtasks on sub-program spaces and recursive solving, and (3) combination of the sub-results. The unfolding stage proceeds in four steps (using the most complex last term as an example):

(1) The summation of programs expanded from $S \to T + T$ can be unfolded into summations of sub-programs, each ranging on programs expanded from non-terminal T.
(2) The probability of each program can be unfolded into the probability of sub-programs by the definition of PRTG. For example, $\varphi(p_1 + p_2)$ is equal to $\gamma(S \to T + T)\varphi(p_1)\varphi(p_2)$.
(3) The estimation can be unfolded into estimations on sub-programs by its definition.

The result following these three steps is shown below, where each parameter is replaced by its concrete assignment, and $p \in T$ represents that p ranges on those programs expanded from T.

$$\sum_{p_1,p_2,p_1',p_2' \in T} \frac{9}{16}\varphi(p_1)\varphi(p_2)\varphi(p_1')\varphi(p_2')\left(\frac{1}{2} + \frac{1}{2}\mathcal{M}(p_1,p_1')\mathcal{M}(p_2,p_2')\right)$$

(4) In the last step, the inner plus (the blue part) is unfolded into two separate summations and reorganized as below, where a global coefficient of $9/32$ is neglected for simplicity.

$$\sum_{p_1,p_2,p_1',p_2' \in T} \varphi(p_1)\varphi(p_2)\varphi(p_1')\varphi(p_2') + \left(\sum_{p_1,p_1' \in T} \varphi(p_1)\varphi(p_1')\mathcal{M}(p_1,p_1')\right)\left(\sum_{p_2,p_2' \in T} \varphi(p_2)\varphi(p_2')\mathcal{M}(p_2,p_2')\right) \quad (5)$$

In this unfolded formula, the first summation must be 1 by the definition of PRTG and the second part consists of two separate sub-summations with the same form as the original task. Each of these sub-summations corresponds to a non-terminal, and the task is to calculate the sum of (the product of the program probabilities and an estimation) over all program pairs expanded from the non-terminal. Therefore, the two sub-summations are subtasks of the original task and can be calculated recursively. Moreover, these two subtasks are identical because they correspond to the same non-terminal, and we only need to solve one of them to get the results for two.

Case 2: single example. The procedure in Case 1 is almost identical to the general algorithm, except the form of subtasks is still partial due to the lack of examples and the PBE result. Now, we discuss a more general case (Example 5.2), where both the example and the PBE result are involved.

The procedure is generally identical to that in Case 1 till the third stage of unfolding. After that, the term where both programs are expanded from $S \to T + T$ becomes as follows, where a global coefficient of $9/128$ is ignored for simplicity.

$$\sum_{p_1,p_2,p_1',p_2' \in T} \varphi(p_1)\varphi(p_2)\varphi(p_1')\varphi(p_2')\Big(1 + \mathcal{M}_{\langle 1,0\rangle}(p_1,p_1')\mathcal{M}_{\langle 1,0\rangle}(p_2,p_2')\Big)\Big(1 + \mathcal{M}_{\langle 0,2\rangle}(p_1,p_1')\mathcal{M}_{\langle 0,2\rangle}(p_2,p_2')\Big)$$

The next step is to unfold the inner plus (the blue part) into separate summations. Four sub-summations are obtained, where no model, only model $\mathcal{M}_{\langle 1,0\rangle}$, only model $\mathcal{M}_{\langle 0,2\rangle}$, and both models are considered, respectively. These sub-summations are in the following form, where $\overline{\mathcal{M}}$ represents the list of considered models, and $\overline{\mathcal{M}}$ is equal to $[]$, $[\mathcal{M}_{\langle 1,0\rangle}]$, $[\mathcal{M}_{\langle 0,2\rangle}]$, and $[\mathcal{M}_{\langle 1,0\rangle}, \mathcal{M}_{\langle 0,2\rangle}]$ for the four sub-summations, respectively.

$$\left(\sum_{p_1,p_1' \in T} \varphi(p_1)\varphi(p_1') \prod_{\mathcal{M} \in \overline{\mathcal{M}}} \mathcal{M}(p_1,p_1')\right)\left(\sum_{p_2,p_2' \in T} \varphi(p_2)\varphi(p_2') \prod_{\mathcal{M} \in \overline{\mathcal{M}}} \mathcal{M}(p_2,p_2')\right) \tag{6}$$

With a more general definition of subtasks, we can unify the two parts in the formula above and the original task. Each subtask here is specified by a non-terminal, an optional concrete program (only in the original task here), and a series of estimations among the concrete program and two program variables p, p'. Then the goal is to calculate the sum of (the product of $\varphi(p)$, $\varphi(p')$, and all estimations) over all program pairs (p, p') expanded from the non-terminal. Through this unification, we can solve both parts in Formula 6 recursively.

5.3 A Dynamic-Programming Algorithm for the Approximation

In this section, we give a brief introduction to our dynamic-programming algorithm for the approximation, which is naturally generalized from Section 5.2.

Subtasks. Given a PRTG φ, each subtask generated in our algorithm is specified by a non-terminal s, a concrete program p_c, and two lists of models $\overline{\mathcal{M}}_p$ and $\overline{\mathcal{M}}_c$[6]. The task is to calculate a summation over all program pairs (p, p') expanded from non-terminal s, where the contribution of each pair is equal to the product of (1) the probabilities of p and p', (2) the estimations for (p, p') given by models in $\overline{\mathcal{M}}_p$, and (3) the estimations for (p_c, p) given by models in $\overline{\mathcal{M}}_c$. Formally, the task is to calculate the formula below.

$$\sum_{p,p' \in s} \varphi(p)\varphi(p') \prod_{\mathcal{M} \in \overline{\mathcal{M}}_p} \mathcal{M}(p,p') \prod_{\mathcal{M} \in \overline{\mathcal{M}}_c} \mathcal{M}(p_c,p)$$

Specially, we regard p_c as $\perp$ when it is never used, i.e., when $\overline{\mathcal{M}}_c = []$.

Example 5.5. Now we apply this definition to the tasks discussed in previous sections. For simplicity, we use a temporary notation $\mathcal{S}(s, p_c, \overline{\mathcal{M}}_p, \overline{\mathcal{M}}_c)$ to denote a subtask in this example.

- The first case in Section 5.2 and its subtask in Formula 5 correspond to $\mathcal{S}(S, \perp, [\mathcal{M}], [])$ and $\mathcal{S}(T, \perp, [\mathcal{M}], [])$, respectively. The second case and its subtask in Formula 6 correspond to $\mathcal{S}(S, y, [\mathcal{M}_{\langle 1,0\rangle}, \mathcal{M}_{\langle 0,2\rangle}], [\mathcal{M}_{\langle 0,2\rangle}])$ and $\mathcal{S}(T, \perp, \overline{\mathcal{M}}, [])$ respectively.

[6]Here we use lists instead of sets because a model may be involved multiple times in theory.

- Given PBE result p_c and an example set E, the approximated objective value of input I corresponds to $\mathcal{S}(s_0, p_c, \mathcal{M}_I : \overline{\mathcal{M}}, \overline{\mathcal{M}})$, where s_0 is the start non-terminal, $\mathcal{M}_I$ represents the model trained for input I, operator : represents the constructor of lists, and $\overline{\mathcal{M}}$ represents the learned models for inputs involved in E, i.e., $[\mathcal{M}_{I'} \mid (I' \mapsto O') \in E]$.

Procedure. The procedure of our algorithm is almost as discussed in Section 5.2, so we only make a summary here. For any subtask, our algorithm runs in the following steps, with a memoization table restoring the results of visited subtasks.

(1) Directly return the result in the memoization table when the subtask has been visited.
(2) Divide the summation into cases according to the first rules used by p and p'.
(3) For each case, unfold the summation to sub-programs, unfold probabilities of programs and estimations according to their definition, and unfold the inner plus via the distributive law.
(4) Express each case using subtasks and then solve these subtasks recursively.
(5) Calculate the result from sub-results, restore it into the memoization table, and then return.

Time complexity. The time cost of dynamic programming depends on (1) the number of different subtasks and (2) the time cost to process each subtask without considering the recursions.

- For the first factor, our algorithm ensures that, in each subtask, the concrete program is a sub-program of that in the original task, and both model lists are sub-lists of the counterparts in the original task. Therefore, the number of different subtasks is bounded by the grammar size and the scale of the original task.
- For the second factor, the unfolding procedure and the calculation from sub-results can be efficiently implemented by the hypercube prefix-sum algorithm [Kumar et al. 1994].

Combining both factors, the time complexity of our algorithm is $O(mn_r^2 s_p 2^m)$, where m represents the total size of the model lists in the original task, n_r represents the number of rules in the grammar, and s_p represents the size of the concrete program in the original task. Therefore, the time cost of our algorithm to calculate an approximated objective value is polynomial to the grammar size and exponential to the number of existing examples.

In some extreme cases, the number of examples could be large. To ensure efficiency in these cases, we set up a limit lim_e on the number of examples and consider only the latest lim_e examples when calculating the approximation. Intuitively, such a truncation may not affect the result much because the neglected examples have been considered many times in previous iterations.

6 IMPLEMENTATION

We implement the algorithms discussed in Section 5 into a question selector named *LearnSy*. Our implementation is in C++ and is available online [Ji et al. 2023b].

Dealing with infinite program spaces. Our approximation requires the program space to be finite (Section 3.2). Even so, we can generalize it to the infinite case by an iterative truncation of the program space. When running on an infinite program space, *LearnSy* iterates with a depth limit lim_d and works on a finite subspace including only programs with a depth no larger than lim_d. When the depth of the PBE result exceeds lim_d, *LearnSy* enlarges lim_d to the depth of the PBE result to include this program in the subspace considered.

In our implementation, the initial value of the depth limit is simply set to 1.

Generating candidate inputs. *LearnSy* uses a generator to generate candidate inputs (Section 2.4). We implement different generators for interactive and non-interactive tasks, respectively.

The generator for interactive tasks follows the framework of interactive program synthesis [Ji et al. 2020a]. It generates those inputs that distinguish the PBE result from some other remaining

program. Specifically, this generator encodes the conditions above into an SMT formula by an existing method [Ji et al. 2020a], and thus generates candidate inputs via an SMT solver.

The generator for non-interactive tasks follows the CEGIS framework [Solar-Lezama et al. 2006]. It generates those inputs where the PBE result outputs incorrectly. Specifically, this generator assumes that the complete specification of the target program is provided as either an SMT formula or a list of examples, and thus generates candidate inputs either via an SMT solver or simply by enumeration.

We use *Z3* [de Moura and Bjørner 2008] as the underlying SMT solver and set the upper bound on the number of candidate inputs as 5.

Learning the PRTG. Our approximation requires a PRTG φ over the program space. We learn this PRTG in the same way as previous studies [Ji et al. 2020b; Lee et al. 2018]. Given a RTG, the learning method relies on a set of sample programs in the grammar. For each vertex on the AST of each sample program, the method records the non-terminal and the first grammar rule corresponding to each vertex. Then, for each grammar rule r expanding from non-terminal s, the method learns its probability as the ratio of the number of vertices with rule r to the number of vertices with non-terminal s.

In our evaluation, sample programs are obtained by 2-fold cross-validation. We divide each dataset into two halves, learn a PRTG for each half by taking the synthesis targets of tasks in this half as the samples, and use the PRTG learned for one half to solve tasks in the other half.

Flattening. Our implementation repeatedly applies the flattening operation to improve the precision of the learned model. *LearnSy* greedily flattens the rule that expands the fewest programs in each turn and stops once the number of introduced grammar rules exceeds a limit lim_f. As will be discussed in Section 7, by making a trade-off between precision and efficiency, we set lim_f to 3000 and 100 for interactive and non-interactive tasks, respectively.

While forming OGIS solvers, our implementation invokes the PBE solver with the original grammar instead of the flattened one to reduce the time rise led by flattening. Since there is a natural bijection between programs in the original grammar and those in the flattened grammar, in each OGIS iteration, our implementation invokes the PBE solver with the original grammar, converts the synthesis result to the corresponding program in the flattened grammar, and then invoke *LearnSy* with the flattened grammar. In this way, the time rise led by flattening is limited to only the question selector.

Other Configuration. We set n_s (the number of samples used while learning models) to 10 and set lim_e (the limit on the number of examples considered in the approximation) to 3.

7 EVALUATION

To evaluate *LearnSy*, we report several experiments to answer the following research questions.

- **RQ1**: Can *LearnSy* improve the performance of OGIS solvers on interactive tasks?
- **RQ2**: Can *LearnSy* improve the performance of OGIS solvers on non-interactive tasks?
- **RQ3**: How does the prior distribution φ affects the performance of *LearnSy*?
- **RQ4**: How does the flattening operator affect the performance of *LearnSy*?

7.1 Exp 1: Effect of Improving OGIS Solvers on Interactive Tasks

In this experiment, our evaluation is conducted on the interactive synthesizer proposed by Ji et al. [2020a]. We implement several variants of this synthesizer for *LearnSy* and two other baseline selectors and compare these variants on the dataset collected by Ji et al. [2020a].

Baseline. We consider selectors proposed by previous studies on interactive synthesis.

- *RandomSy* [Mayer et al. 2015] has no preference for the selected input. It repeatedly selects a random input until an input distinguishing at least two remaining programs is found.
- *SampleSy* [Ji et al. 2020a] is a state-of-the-art selector to reduce iterations. It approximates a strategy for building effective decision trees, but unlike our approach, its approximation is performed by sampling remaining programs. To ensure quick responses, *SampleSy* (1) requires efficient witness functions [Polozov and Gulwani 2015] for all operators in the domain, and (2) performs sampling in the background when the user is answering the question.

Dataset. The dataset collected by Ji et al. [2020a] is used in this experiment. This dataset involves two synthesis domains and is adapted from the datasets in SyGuS-Comp [Alur et al. 2017a] by bounding each program space with a proper depth limit to ensure the termination of the interaction.

- 150 tasks on the domain of string manipulation are included. In these tasks, the specification is provided as a set of examples whose average size is 45.8, and the average size of the program space is 4.0×10^{25}.
- 16 tasks on the domain of program repair are included, which are extracted from the program repair process for real-world Java bugs. In these tasks, the specification is provided as an SMT formula, and the average size of the program space is 2.4×10^{8}.

Configurations. This experiment is conducted on Intel Core i7-8700 3.2GHz 6-Core Processor. For each execution, we set the time limit to 60 minutes and the memory limit to 8GB; in other words, if the time or memory cost of a synthesis procedure exceeds the corresponding limit, the procedure will be considered to fail. Besides, since many approaches in this experiment are random, we repeat each execution 3 times and consider only the average performance.

To simulate the interactive scenario, we limit the *response time* of each OGIS iteration. In practice, for the sake of the user experience, interactive synthesizers are usually expected to respond within 2 seconds [Ji et al. 2020a], that is, after the user provides the answer, the question selector is expected to determine the next question within two seconds. In this experiment, we limit the response time to 2 seconds for *LearnSy* and all baseline selectors.

- For *LearnSy*, we set the number limit of rules introduced by flattening (i.e., parameter lim_f in Section 6) to 3000. In this case, the response time of *LearnSy* never exceeds 2 seconds in our dataset, with an average time of 0.91 second per iteration.
- For *RandomSy*, this requirement is already satisfied on all tasks in our dataset. Thus, no extra configuration is required.

The other baseline selector, *SampleSy*, needs more complex configurations. In each iteration, *SampleSy* runs in two stages.

(1) In the first stage, *SampleSy* performs sampling in the background while the user is answering the previous question. Therefore, the time limit of this stage correlated to the time the user needs to respond to the queries. The default implementation of *SampleSy* assumes this time to be at least 2 minutes for each question and thus sets the time limit to 2 minutes.
(2) In the second stage, *SampleSy* selects an input online according to the samples after the user has responded to the previous question. The time limit for this stage was set to 2 seconds in our experiment since the time cost in this stage is indeed the response time.

Aside from these configurations, we consider several other variants of *SampleSy* to simulate the cases with different question difficulties. We observe that the assumption made by the default implementation of *SampleSy* (i.e., the user requires at least two minutes to answer each question) does not hold for many existing tasks. For example, task `phone.sl` in dataset $\mathbb{I}_S$ is about synthesizing a program that extracts the first three digits from a phone number. In this task, the question is like "what is the intended output on input `938-242-504`", and the user needs to respond `938`. Such

Table 3. The results of comparing *LearnSy* with baselines on interactive tasks.

Selector	#Solved	#Iteration				
		Ave	RED. for All		RED. for Hard	
LearnSy	145	**3.245**				
RandomSy		3.756	0.510	13.58%	3.410	39.00%
$SampleSy_3$		3.262	0.016	0.493%	0.615	10.43%
$SampleSy_5$		3.309	0.063	1.910%	0.897	14.52%
$SampleSy_{10}$		3.287	0.041	1.258%	0.692	11.48%
$SampleSy_{15}$		3.275	0.029	0.912%	0.487	8.370%
SampleSy		3.251	0.005	0.176%	0.358	6.306%
The average time cost of *LearnSy* is 0.91 seconds per iteration						

a query can be easily answered in seconds. At this time, the user will have to wait for nearly 2 minutes until *SampleSy* finishes sampling if the default time limit for sampling is used. Therefore, we introduce a series of variants of *SampleSy*, denoted as $SampleSy_t$, where the time limit for sampling is set to t seconds and the time limit for the second stage is still 2 seconds. In this experiment, we evaluate four choices of $t \in \{3, 5, 10, 15\}$ and compare their performances with *LearnSy*.

Results. The results of this experiment are summarized in Table 3, organized as follows.

- For each selector, columns "#Solved" and "#Iteration-Ave" report the number of solved tasks and the average number of iterations used, respectively.
- For each baseline selector, column "#Iteration-RED. for All" reports the absolute/relative reductions in the average number of iterations achieved by *LearnSy*.
- For each baseline selector, column "#Iteration-RED. for Hard" reports the absolute/relative reduction achieved by *LearnSy* on the hardest 10% of tasks, where the difficulty of a task is measured as the total number of iterations used by *LearnSy* and the baseline selector.

In this experiment, all considered OGIS solvers solve the same subset of tasks. This is because the time limit and memory limit in this experiment are large enough for all selectors so that the OGIS solver fails only when the underlying synthesizer [Ji et al. 2020a] (which is fixed in this experiment) violates the requirement on the response time.

The results show that *LearnSy* outperforms *RandomSy* and offers competitive performance to *SampleSy*. Although *SampleSy* tends to perform better as the time limit of sampling increases, its number of iterations is still close to that of *LearnSy* even when the time limit is as large as 2 minutes. Note that this time cost of sampling is already more than 100 times larger than the average time cost of *LearnSy*.

Besides, the advantage of *LearnSy* compared with *SampleSy* becomes more remarkable on the hard tasks. This tendency is related to the difficulty to sample remaining programs. *SampleSy* can get enough samples quickly on simple tasks so that the sampling time becomes insignificant. However, when processing difficult tasks, *SampleSy* can hardly collect enough samples within a short time, leading to its low performance. In our opinion, the performance of a selector on hard tasks is more important because the user burden when solving these tasks is usually heavier.

In the end, it is noteworthy that the comparably superior performance of *LearnSy* is achieved **without losing any generality**. *LearnSy* only uses the semantics of programs and the grammar structure of the program space. As a result, it can be applied to all tasks under the SyGuS framework. In contrast, *RandomSy* has a much worse performance despite being general, and *SampleSy* has a scope limited to domains with efficient witness functions despite a similar performance to *LearnSy*.

Table 4. The datasets of non-interactive tasks considered in our evaluation

Name	Size[8]	Domain	#Operators	Spec	PBE Solver	Selector
$\mathcal{C}_S$	205	String	16	Examples	*Eusolver*, *MaxFlash*	*Default*, *SigInp*
$\mathcal{C}_I$	100	Integer	10	SMT formula	*Eusolver*, *PolyGen*	*Default*
$\mathcal{C}_B$	450	Bit-Vector	10	Examples	*Eusolver*	*Default*, *Baised*

7.2 Exp 2: Effect of Improving OGIS Solvers on Non-Interactive Tasks

In this experiment, our evaluation is conducted under the CEGIS framework [Solar-Lezama et al. 2006]. We consider three state-of-the-art PBE solvers to form CEGIS solvers. Subsequently, we implement multiple CEGIS solvers by combining these PBE solvers with *LearnSy* or baseline selectors and evaluate these CEGIS solvers on three datasets in SyGuS-Comp.

PBE solvers. *LearnSy* regards a PBE solver as a black box and thus is capable to combine with any existing PBE solvers. To test the generality of *LearnSy*, we consider three PBE solvers based on completely different techniques, including one general-purpose and two domain-specific solvers.

- *Eusolver* [Alur et al. 2017b] is a general-purpose solver based on enumeration.
- *MaxFlash* is a solver specialized for domains with efficient witness functions [Polozov and Gulwani 2015], such as the domain of string manipulation. It also uses a probabilistic model to guide the search procedure.
- *PolyGen* [Ji et al. 2021] is a solver specialized for branch expressions. It uses the theory of Occam learning to guarantee generalizability [Blumer et al. 1987] and also uses constraint solvers to efficiently synthesize sub-programs.

Baseline. As no previous question selector can improve the efficiency of CEGIS solvers, we compare *LearnSy* with the default selector used in the original CEGIS framework [Solar-Lezama et al. 2006], denoted as *Default*. This selector has no preference for the selected input and directly returns the first found input where the PBE result outputs incorrectly.

Besides, some domain-specific heuristics for input selection have been proposed as optimizations in previous studies on domain-specific PBE solvers [Jha et al. 2010; Padhi et al. 2018]. They are also involved as baselines in this experiment:

- *Significant input* [Padhi et al. 2018] (denoted as *SigInp*) is specialized for the domain of string manipulation. It synthesizes the syntactic profile for each input through another synthesis task and prefers inputs with a different profile compared to those in existing examples[7].
- *Biased Bit-Vector* [Jha et al. 2010] (denoted as *Biased*) is specialized for the domain of bit-vectors. It is based on the property that for many bit-vector functions, the rightmost bits in the input influence the output more than the leftmost ones [Warren 2002]. Therefore, it prefers those inputs with different rightmost bits compared to those in existing examples.

Dataset. We consider three datasets $\mathcal{C}_S$, $\mathcal{C}_I$, and $\mathcal{C}_B$ in SyGuS-Comp. These datasets are on three different domains, and their profiles are shown in Table 4.

Configuration. The configurations are similar to those for interactive tasks except for two changes. First, the time limit of each execution is reduced to 5 minutes to save time since the number of non-interactive tasks far exceeds interactive ones. Second, the number limit of rules introduced by flattening in *LearnSy* is set to 100 to ensure that the time cost of the question selection has a

[7] *SigInp* is proposed as an application of *FlashProfile* [Padhi et al. 2018], a synthesizer for syntactic profiles, using a cost function with several parameters. Since its original implementation has not been released, we re-implement it in our evaluation using *FlashProfile* and set all missing parameters as a default constant.

Table 5. The results of comparing *LearnSy* with baselines on non-interactive tasks.

	Solver	Selector	#Solved	#Iteration			TimeCost (s)				
				Ave	RED. for All		Ave	RED. for All		RED. for Hard	
$\mathcal{C}_S$	*Eusolver*	*LearnSy*	**146**	2.69			**14.4**				
		Default	145	2.82	0.13	4.56%	15.2	0.83	5.46%	8.68	6.61%
		SigInp	**146**	**2.61**	−0.08	−3.24%	14.9	0.47	3.15%	3.09	2.18%
	MaxFlash	*LearnSy*	**159**	**1.42**			**0.96**				
		Default	154	1.51	0.09	6.29%	1.41	0.45	31.9%	4.56	24.8%
		SigInp	155	1.47	0.06	4.04%	1.57	0.61	38.8%	6.58	32.7%
$\mathcal{C}_I$	*Eusolver*	*LearnSy*	**43**	**15.1**			**9.26**				
		Default	42	21.0	5.92	28.2%	12.3	3.06	24.9%	6.68	11.0%
	PolyGen	*LearnSy*	**82**	**34.4**			**7.27**				
		Default	78	47.8	13.4	28.0%	12.7	5.48	43.0%	59.9	50.4%
$\mathcal{C}_B$	*Eusolver*	*LearnSy*	**427**	**10.2**			**24.2**				
		Default	424	11.2	0.91	8.16%	27.5	3.32	12.1%	19.0	11.8%
		Biased	424	10.9	0.69	6.34%	25.0	0.75	2.99%	1.08	0.76%

limited effect on the efficiency of CEGIS solvers. Under this limit, the time cost of *LearnSy* is about 0.03 second per iteration, negligible in most cases.

Result. The results of this experiment are summarized in Table 5. For each dataset, each PBE solver, and each selector, we report the number of solved tasks (column "#Solved"), the average number of used iterations (column "#Iteration-Ave")[9], the absolute/relative reduction in the number of iterations achieved by *LearnSy* (column "#Itertion-RED. for All"), the average time cost (column "TimeCost-Ave"), the absolute/relative reduction in the time cost achieved by *LearnSy* (column "TimeCost-RED. for All"), and the reduction in the time cost on the hardest 10% of tasks (column "TimeCost-RED. for Hard"). In this experiment, the difficulty of tasks is measured as the total time cost of *LearnSy* and the baseline selector since efficiency is the primary pursuit for non-interactive synthesizers. We summarize these results as follows.

First, compared with the default selector, the CEGIS solver with *LearnSy* is always more efficient with fewer iterations. By replacing the default selector in the CEGIS framework with *LearnSy*, the existing CEGIS solvers can be immediately accelerated with a ratio of up to 43.0%. This improvement is valuable because of two aspects:

- **Generality**. As a general selector, *LearnSy* can be applied to all tasks under the SyGuS framework regardless of the synthesis domain and the PBE solver. Though the speedup achieved by *LearnSy* may not be significant for an individual PBE solver on a concrete domain, it has the potential to hold for all PBE solvers and all domains. Therefore, the overall improvement in efficiency should be considerable.
- **Originality**. Although multiple approaches to reduce the number of iterations are available, *LearnSy* provides new insights for question selection (i.e., estimating the behavior of operators to enable a compositional approximation) and achieves speedups for OGIS solvers by question selection **for the first time**. The performance of *LearnSy* may be enhanced by more desirable methods to estimate the operators.

Second, *LearnSy* offers competitive performance to those heuristics (*Biased* and *SigInp*). Such a result is already valuable because both *SigInp* and *Biased* are based on domain-specific heuristics

[9]In Exp2, the tasks solved by different selectors are not always the same. Therefore, to ensure fair comparisons, only those tasks solved by all applicable selectors are considered when calculating the average values.

designed by domain experts while *LearnSy* is general. Besides, we observe that *SigInp* performs slightly better than *LearnSy* in reducing iterations when combined with *EuSolver* but only achieves a smaller speed-up. This is because *SigInp* selects inputs by solving another synthesis task, whose time cost is large enough to affect the efficiency of the whole OGIS solver.

Third, we would like to explain why selector *SampleSy* cannot be applied to improve the efficiency of the solvers considered in this experiment.

(1) There is no efficient witness function for the domain of integers and bit-vectors, so *SampleSy* cannot be used on dataset $\mathcal{C}_I$ and $\mathcal{C}_B$. In brief, a witness function acts as the inverse semantics of an operator, returning those possible inputs to produce a given input. However, the number of possible inputs may be too large to enable an efficient witness function for many operators in these two domains. For example, there are 2^{64} possible inputs to output 0 for *bvadd*, an operator in the domain of bit-vectors that calculates the sum of two 64-bit vectors after regarding them as 64-bit unsigned integers.
(2) A full version-space algebra for all remaining programs is required by *SampleSy* to perform sampling. As shown in a previous evaluation [Ji et al. 2020b], the time cost to construct the full version-space algebra is 400 times larger compared to that needed for *MaxFlash* to find the PBE result. Therefore, *SampleSy* can hardly accelerate *MaxFlash* on $\mathcal{C}_S$.
(3) The only possible case to apply *SampleSy* is to accelerate *Eusolver* on $\mathcal{C}_S$, but this case is still difficult. In this experiment, *Eusolver* takes only 5.8 seconds per iteration on average when the default selector is used. Therefore, the time cost of *Eusolver* will rise significantly even with $SampleSy_3$.

Finally, the speedups achieved by *LearnSy* vary among different datasets, different PBE solvers, and tasks with different difficulties. Theoretically, there are some cases where *LearnSy* is likely to achieve higher speedups:

- *LearnSy* tends to perform better when more examples are required to solve the synthesis task (as seen in the rows of dataset $\mathcal{C}_I$ in Table 5). In this case, a larger space to reduce the number of iterations tends to be present, and an effective question selector can lead to a more significant performance enhancement.
- *LearnSy* tends to perform better when the time cost of the PBE solver increases rapidly with the number of examples (as seen in the rows of *MaxFlash* in Table 5). In this case, a given reduction in the number of iterations can result in a more significant drop in the time cost.
- *LearnSy* tends to perform better on harder tasks. Solving a harder task usually requires more examples and thus matches the first case discussed. In Table 5, the absolute improvement achieved by *LearnSy* is always more significant when only hard tasks are considered.

7.3 Exp 3: Comparison between Different Prior Distributions

LearnSy requires a prior distribution φ, and our implementation of *LearnSy* uses 2-fold cross-validation to learn φ by default (Section 6). In this experiment, we test how φ affects the performance of *LearnSy*. For simplicity, we consider only one single setting in this experiment, where only dataset $\mathcal{C}_S$ and PBE solver *MaxFlash* are considered.

Baseline. We consider a trivial PRTG where the probability assigned to each rule starting from the same non-terminal is the same. We denote the variant of *LearnSy* with this PRTG as $LearnSy_U$ and compare it with the default version.

Configuration. The configurations are exactly the same as those in Exp 2.

Results. The results are summarized in Table 6 in the same way as Table 5. *LearnSy* outperforms its invariant $LearnSy_U$, implying a tendency that the precision of φ contributes positively to the

Table 6. The results of comparing *LearnSy* with its variants.

PBE Solver	Exp	Selector	#Solved	#Iteration			TimeCost (s)				
				Ave	RED. for All		Ave	RED. for All		RED. for Hard	
MaxFlash	3	*LearnSy*	**159**	**1.44**			**1.97**				
		$LearnSy_U$	155	1.46	0.02	1.25%	2.36	0.39	16.7%	3.98	18.1%
	4	*LearnSy*	**159**	**1.45**			**2.37**				
		$LearnSy_0$	158	1.48	0.03	2.15%	2.74	0.37	13.5%	3.99	15.2%

performance of *LearnSy*. Moreover, even though $LearnSy_U$ does not perform as well as *LearnSy*, it still outperforms the default CEGIS selector and *SigInp* (see Section 7.2). Therefore, for those scenarios where learning a PRTG is difficult, it is still possible to improve OGIS solvers by applying *LearnSy* with some pre-defined PRTGs.

7.4 Exp 4: Comparison between Different Usages of Flattening

LearnSy uses flattening to improve the precision of the learned model, and when solving non-interactive tasks, its default applies flattening repeatedly until the number of introduced rules exceeds 100. In this experiment, we test how flattening affects the performance of *LearnSy*.

Similar to Exp 3, only dataset $\mathcal{C}_S$ and PBE solver *MaxFlash* are considered in this experiment.

Baseline. We consider a variant of *LearnSy* where no flattening is applied, denoted as $LearnSy_0$, and compare this variant with the default *LearnSy*.

Configuration. The configurations are exactly the same as those in Exp 2.

Results. The results are summarized in Table 6 in the same way as Table 5. After applying flattening, the number of iterations and the time cost of *LearnSy* are reduced. This finding provides indirect evidence that the flattening operation can improve the precision of our approximation, which justifies our theoretical analysis.

8 RELATED WORK

Question selection for interactive program synthesis. The question selection problem for interactive program synthesis targets to reduce the user burden by reducing the number of iterations and has been studied by Ji et al. [2020a] and Tiwari et al. [2020]. Both studies adopt a greedy strategy for constructing effective decision trees and approximate the strategy by sampling from remaining programs. Compared to these approaches, *LearnSy* performs approximation via unified-equivalence models instead of sampling, leading to a significant improvement in the efficiency of question selection and wider applications to domains without an efficient sampling algorithm. However, Ji et al. [2020a] also explore a variant of the question selection problem where a bounded error rate is allowed. We do not consider this variant in this paper. Applying *LearnSy* to it is future work.

Many interactive synthesis systems [Ferreira et al. 2021; Mayer et al. 2015; Wang et al. 2017] include a question selector for input-output examples, whose goal is to find an input where at least two remaining programs output differently. These question selectors have no preference between multiple such inputs. Concretely, Mayer et al. [2015] randomly select an input until a valid one is found, Wang et al. [2017] randomly mutate an existing input until a valid one is found, and Ferreira et al. [2021] get valid inputs by invoking an SMT solver. We approximate all these approaches as *RandomSy* in our evaluation.

Question selection for CEGIS. Several heuristics have been proposed for selecting inputs to improve the performance of CEGIS solvers. *Biased BV* [Jha et al. 2010] is proposed for bit-vector functions, and *Significant Input* [Padhi et al. 2018] is proposed for string manipulation programs.

Both of them rely on domain knowledge and cannot be generalized to other domains. To our knowledge, *LearnSy* is the first general question selector that can improve the efficiency of state-of-the-art PBE solvers under the CEGIS framework.

Oracle guided inductive synthesis. Jha and Seshia [2017] establish a formal theory for OGIS. In this theory, an OGIS solver is formed by an oracle and a learner only, and either of these two components may fulfill the role of the question selector, depending on the type of queries. For example, in CEGIS, the query asks for a counterexample for a candidate program, so the input is selected by the oracle; in interactive program synthesis, the query asks for the corresponding output for a given input, so the input is selected by the learner. In this paper, to focus on the question selection problem, we explicitly regard the question selector as a component in OGIS.

Besides, only queries in the form of input-output examples are considered in this paper. This query form is indeed the most common, but multiple OGIS solvers have been proposed based on other forms of queries, including (1) selecting a program from a set of given programs [Mayer et al. 2015; Yessenov et al. 2013], (2) selecting the correct output for a given input among several candidates [Ramos et al. 2020], (3) providing input-output examples to refine the specification [Galenson et al. 2014; Kandel et al. 2011; Narita et al. 2021; Yessenov et al. 2013], and (4) changing the configuration of the synthesizer [Barman et al. 2015; Gvero et al. 2011; Kandel et al. 2011; Leung et al. 2015]. Generalizing *LearnSy* to these queries is future work.

9 CONCLUSION

In this paper, we propose a question selector *LearnSy* to improve the performance of OGIS solvers by reducing the number of iterations used for synthesis. *LearnSy* is based on a greedy strategy *min-pair* that finds the input best distinguishing between remaining programs.

To apply the min-pair strategy efficiently in synthesis, we design a compositional approximation of the objective function in this strategy. The approximation is based on the unified-equivalence model, which approximates the behavior of operators as random and estimates whether two programs output the same with probabilities. We prove that this model can provide an unbiased estimation of the overall equivalence in the program space. In addition, we also prove that the precision of each estimation can be further improved by an operator named flattening.

Finally, we propose an efficient dynamic-programming algorithm for our approximation and implement it into *LearnSy*. Our evaluation shows that *LearnSy* can generally improve the performance OGIS solvers on both interactive and non-interactive tasks. Such an improvement is completely free as *LearnSy* neither requires domain knowledge nor limits the behavior of PBE solvers.

ACKNOWLEDGEMENT

We sincerely thank the anonymous OOPSLA reviewers for their valuable feedback on this work and the anonymous OOPSLA Artifacts reviewers for their suggestions for our experiments. This work is supported in part by the National Key Research and Development Program of China under Grant No. 2021ZD0110202, the National Natural Science Foundation of China under Grant No. 62161146003, and a grant from ZTE-PKU Joint Laboratory for Foundation Software.

REFERENCES

Micah Adler and Brent Heeringa. 2012. Approximating Optimal Binary Decision Trees. *Algorithmica* 62, 3-4 (2012), 1112–1121. https://doi.org/10.1007/s00453-011-9510-9

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*. 1–8. http://ieeexplore.ieee.org/document/6679385/

Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017a. SyGuS-Comp 2017: Results and Analysis. In *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017*. 97–115. https://doi.org/10.4204/EPTCS.260.9

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017b. 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*. 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

Shaon Barman, Rastislav Bodík, Satish Chandra, Emina Torlak, Arka Aloke Bhattacharya, and David E. Culler. 2015. Toward tool support for interactive synthesis. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Gail C. Murphy and Guy L. Steele Jr. (Eds.). ACM, 121–136. https://doi.org/10.1145/2814228.2814235

Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. 1987. Occam's Razor. *Inf. Process. Lett.* 24, 6 (1987), 377–380. https://doi.org/10.1016/0020-0190(87)90114-1

Venkatesan T. Chakaravarthy, Vinayaka Pandit, Sambuddha Roy, Pranjal Awasthi, and Mukesh K. Mohania. 2011. Decision trees for entity identification: Approximation algorithms and hardness results. *ACM Trans. Algorithms* 7, 2 (2011), 15:1–15:22. https://doi.org/10.1145/1921659.1921661

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Margarida Ferreira, Miguel Terra-Neves, Miguel Ventura, Inês Lynce, and Ruben Martins. 2021. FOREST: An Interactive Multi-tree Synthesizer for Regular Expressions. In *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12651)*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer, 152–169. https://doi.org/10.1007/978-3-030-72016-2_9

Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. 2014. CodeHint: dynamic and interactive synthesis of code snippets. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 653–663. https://doi.org/10.1145/2568225.2568250

Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. 2011. Interactive Synthesis of Code Snippets. 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, 418–423. https://doi.org/10.1007/978-3-642-22110-1_33

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, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 215–224. https://doi.org/10.1145/1806799.1806833

Susmit Jha and Sanjit A. Seshia. 2017. A theory of formal synthesis via inductive learning. *Acta Informatica* 54, 7 (2017), 693–726. https://doi.org/10.1007/s00236-017-0294-5

Ruyi Ji, Chaozhe Kong, Yingfei Xiong, and Zhenjiang Hu. 2023a. *Artifact for OOPSLA'23: Improving Oracle-Guided Inductive Synthesis by Efficient Question Selection.* https://github.com/jiry17/LearnSy

Ruyi Ji, Chaozhe Kong, Yingfei Xiong, and Zhenjiang Hu. 2023b. *Artifact for OOPSLA'23: Improving Oracle-Guided Inductive Synthesis by Efficient Question Selection.* https://doi.org/10.5281/zenodo.7722241

Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. 2020a. Question selection for interactive program synthesis. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1143–1158. https://doi.org/10.1145/3385412.3386025

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

Ruyi Ji, Jingtao Xia, Yingfei Xiong, and Zhenjiang Hu. 2021. Generalizable synthesis through unification. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28. https://doi.org/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. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. https://openreview.net/forum?id=rywDjg-RW

Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2011. Wrangler: interactive visual specification of data transformation scripts. In *Proceedings of the International Conference on Human Factors in Computing Systems, CHI 2011, Vancouver, BC, Canada, May 7-12, 2011*, Desney S. Tan, Saleema Amershi, Bo Begole, Wendy A. Kellogg, and Manas Tungare (Eds.). ACM, 3363–3372. https://doi.org/10.1145/1978942.1979444

Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. 1994. *Introduction to parallel computing*. Vol. 110. Benjamin/Cummings Redwood City, CA.

Vu Le, Daniel Perelman, Oleksandr Polozov, Mohammad Raza, Abhishek Udupa, and Sumit Gulwani. 2017. Interactive Program Synthesis. *CoRR* abs/1703.03539 (2017). arXiv:1703.03539 http://arxiv.org/abs/1703.03539

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*. 436–449. https://doi.org/10.1145/3192366.3192410

Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive parser synthesis by example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 565–574. https://doi.org/10.1145/2737924.2738002

Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin G. Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST 2015, Charlotte, NC, USA, November 8-11, 2015*, Celine Latulipe, Bjoern Hartmann, and Tovi Grossman (Eds.). ACM, 291–301. https://doi.org/10.1145/2807442.2807459

Minori Narita, Nolwenn Maudet, Yi Lu, and Takeo Igarashi. 2021. Data-centric disambiguation for data transformation with programming-by-example. In *IUI '21: 26th International Conference on Intelligent User Interfaces, College Station, TX, USA, April 13-17, 2021*, Tracy Hammond, Katrien Verbert, Dennis Parra, Bart P. Knijnenburg, John O'Donovan, and Paul Teale (Eds.). ACM, 454–463. https://doi.org/10.1145/3397481.3450680

Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd D. Millstein. 2018. FlashProfile: a framework for synthesizing data profiles. *PACMPL* 2, OOPSLA (2018), 150:1–150:28. https://doi.org/10.1145/3276520

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*. 107–126. https://doi.org/10.1145/2814270.2814310

Daniel Ramos, Jorge Pereira, Inês Lynce, Vasco M. Manquinho, and Ruben Martins. 2020. UNCHARTIT: An Interactive Framework for Program Recovery from Charts. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 175–186. https://doi.org/10.1145/3324884.3416613

David E. Shaw, William R. Swartout, and C. Cordell Green. 1975. Inferring LISP Programs From Examples. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 3-8, 1975*. 260–267. http://ijcai.org/Proceedings/75/Papers/037.pdf

Rishabh Singh and Sumit Gulwani. 2015. Predicting a Correct Program in Programming by Example. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 398–414. https://doi.org/10.1007/978-3-319-21690-4_23

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. 404–415. https://doi.org/10.1145/1168857.1168907

Ashish Tiwari, Arjun Radhakrishna, Sumit Gulwani, and Daniel Perelman. 2020. Information-theoretic User Interaction: Significant Inputs for Program Synthesis. *CoRR* abs/2006.12638 (2020). arXiv:2006.12638 https://arxiv.org/abs/2006.12638

Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Interactive Query Synthesis from Input-Output Examples. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1631–1634. https://doi.org/10.1145/3035918.3058738

Henry S Warren. 2002. Hacker's Delight.

Kuat Yessenov, Shubham Tulsiani, Aditya Krishna Menon, Robert C. Miller, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. 2013. A colorful approach to text processing by example. In *The 26th Annual ACM Symposium on User Interface Software and Technology, UIST'13, St. Andrews, United Kingdom, October 8-11, 2013*, Shahram Izadi, Aaron J. Quigley, Ivan Poupyrev, and Takeo Igarashi (Eds.). ACM, 495–504. https://doi.org/10.1145/2501988.2502040

Received 2022-10-28; accepted 2023-02-25