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 for applying program synthesis techniques in practice, and the question selection problem aims at reducing the iterations needed by an OGIS solver for synthesis by selecting a proper input in each OGIS iteration. In theory, a question selector can generally improve 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 selectors fail in satisfying the requirement of efficiency. To ensure effectiveness, these selectors convert the question selection problem into an optimization problem that is so difficult that cannot be solved quickly.

In this paper, we propose a novel selector *LearnSy* that is both efficient and effective and thus achieves the general improvement for OGIS solvers for the first time. We notice that the difficulty of the optimization task in previous studies is from the complex behavior of operators. Therefore, in *LearnSy*, we approximate this task by estimating the behavior of operators to simple random events. We provide theoretical results on the precision of this approximation and also design an efficient algorithm for calculating it.

Our evaluation results show that (1) on interactive tasks, *LearnSy* can achieve competitive performance with existing selectors while being more efficient and more general, and (2) on non-interactive tasks, *LearnSy* can generally reduce the time cost of existing CEGIS solvers with a ratio of up to 49.9%.

1 INTRODUCTION

Oracle-guided inductive synthesis (OGIS) [Jha and Seshia 2017] is a widely-used framework for applying 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 question selector, and an oracle[1]. In each iteration, the PBE solver synthesizes a candidate program, the question selector either accepts the candidate program as the synthesis result or selects an input among all available inputs, the oracle completes the selected input into an input-output example, and this example will be provided to the PBE solver in the next iteration.

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

- In an interactive task, the human user plays the role of the oracle. In each iteration, the question selector shows an input to the user, and the user needs to respond the intended

*Corresponding author

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

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

2023. 2475-1421/2023/10-ART1 $15.00
https://doi.org/

output on this input. For interactive tasks, the performance of an OGIS solver is usually measured by the burden on the user during synthesis.

- In a non-interactive task, the oracle is an executable that can provide the output of the target program, and the performance of an OGIS solver is usually measured by the time cost for synthesis.

For both types, the number of iterations used for synthesis is an important indicator related to the performance of OGIS solvers. For interactive tasks, the number of iterations directly reflects the burden on the user during the synthesis. The fewer the iterations are, the fewer questions will be asked to the user, and thus the less burden on the user will be. For non-interactive tasks, the number of iterations is a critical factor affecting the time cost for synthesis. The fewer the iterations are, the fewer times each component in the OGIS solver will be invoked during synthesis, and thus the faster the OGIS solver will be. Therefore, to improve the performance of OGIS solvers, the problem of reducing 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 great impact on the number of questions needed. For example, when there are only three programs $0, x, y$ available, selecting input $\langle x = 1, y = 2 \rangle$ can help the synthesizer distinguish the correct program, since the output returned by the oracle must uniquely correspond to the target. In contrast, selecting input $\langle x = 0, y = 0 \rangle$ cannot make any progress for synthesis, since the output must be 0 whatever the correct program is.

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

However, solving this optimization problem is difficult, and even evaluating the objective function for a given input faces significant scalability challenges. To measure how an input distinguishes remaining programs, a direct approach needs to enumerate all remaining programs, get the output for each program, and then evaluate the diversity of all these outputs. Such a procedure can hardly be finished within an acceptable time, because the program space in a synthesis task is usually extremely large.

To address the scalability issue, a natural idea is to evaluate the objective function approximately instead of precisely. Existing studies have made attempts in this direction, but the effects of their approaches are still limited [Ji et al. 2020a; Tiwari et al. 2020]. Concretely, they make approximations by sampling. In each selection, they first sample a small set of programs from all remaining programs and then measure the diversity of outputs only on the set of samples. Though this approximation reduces the number of programs considered, it has two main shortcomings.

- **Shortcoming in efficiency**. In theory, drawing a set of samples from remaining programs is a task even harder than the PBE task, since the former requires multiple programs satisfying all examples, but the latter requires only one. Therefore, these approaches usually slow down OGIS solvers and thus can only be applied to interactive tasks.
  Nevertheless, the inefficiency still leads to significant inconvenience for the user. To reduce the response time, Ji et al. [2020a] assume the user needs a long time (2 minutes in their implementation) to answer each query and thus use this period to perform background

sampling. However, for most existing interactive tasks, the queries are usually easy to answer, such as "what are the first three digits in a phone number". For these tasks, the user must wait until enough samples have been collected, resulting in an unacceptable response time.

- **Shortcoming in generality**. Existing approaches [Ji et al. 2020a; Tiwari et al. 2020] rely on efficient witness functions to perform sampling. Therefore, their usages are limited as efficient witness functions only exist in specific synthesis domains.

In this paper, we design an efficient approximation approach for the objective function and thus propose a question selector that achieves general improvement for OGIS solvers for the first time. Compared with previous studies, our approximation approach has two noticeable properties:

- **Efficiency**. Our approach does not rely on any concrete remaining programs and thus does not involve extra synthesis tasks. Its time cost relates only to the size of the grammar specifying the program space and the number of iterations.
- **Generality**. Our approach uses only the semantics of programs and the grammar. It can be applied to all tasks under the framework of syntax-guided synthesis [Alur et al. 2013].

To get an efficient approximation, we analyze the original objective function and find that the main challenge for evaluating it comes from the complex semantics of operators. Based on this observation, we conduct our approximation in three steps. First, we propose the *unified-equivalence model*. It approximates the behavior of operators with probabilities and then estimates how likely two programs output the same on an input. Then, we approximate the objective function using the estimations provided by the model. At last, we show that the calculation of the approximated objective value can be decomposed into subtasks on sub-program spaces and thus propose an efficient dynamic-programming algorithm for it.

We complete our approximation approach into a question selector *LearnSy* in a generate-then-select manner. *LearnSy* first generates a set of candidate inputs and then selects the one with the best approximated objective value among them. 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 a competitive performance with *SampleSy* [Ji et al. 2020a], a state-of-the-art question selector for interactive synthesis, even though *SampleSy* additionally requires a long sampling period and witness functions of related operations.
- For non-interactive tasks, we apply *LearnSy* to the framework of counter-example guided inductive synthesis (CEGIS) [Solar-Lezama et al. 2006] and compare it with (1) the default selector in CEGIS, and (2) two manually designed domain-specific heuristics on selecting inputs in previous studies on PBE solvers [Jha et al. 2010; Padhi et al. 2018]. We combine them with three state-of-the-art PBE solvers, and evaluate them on 755 tasks related to 3 different domains. The results show that *LearnSy* reduces the time cost of all CEGIS solvers considered in our evaluation, with a ratio of up to 49.9%.

2 OVERVIEW

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

2.1 Motivating Example

Our motivating 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 oracle and the PBE solver, and the OGIS solver is applied to solve the sample task.

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

$$S := 1 \mid x \mid y \mid T + T \quad T := 1 \mid x \mid y$$

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

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

For simplicity, we assume the values of variables x and y are limited within $\{0, 1, 2\}$. There are only 9 possible inputs under this assumption.

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

PBE solver. We consider a PBE solver based on enumeration. Given a set of input-output examples, this PBE solver enumerates programs in G_e according to the order listed in Formula 1 and returns the first program that satisfies all given examples.

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

2.2 The Min-Pair Strategy for Question Selection

The question selector can significantly affect the number of iterations needed for synthesis. A naive way to select input is to find a counterexample where the PBE result outputs incorrectly and thus ensure at least one incorrect program is excluded in each iteration. However, in the worst case, the counterexample may overfit the PBE result, and thus the number of iterations can be as large as the size of the program space. Fortunately, the number of iterations can usually be reduced by selecting inputs properly. 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 is much smaller than the size of the program space.

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 for a series of greedy strategies on the question selection problem. In this paper, we consider one of these strategies, namely *min-pair* [Adler and Heeringa 2012; Chakaravarthy et al. 2011]. For simplicity, we discuss only a simplified version of the min-pair strategy in this section and leave the complete definition to Section 3.2. In this simplified version, the objective value of an input is the number of pairs of remaining programs (programs satisfying all existing examples) that output the same on the given input. In each iteration, the min-pair 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 $\langle x = a, y = b\rangle$ whose objective value is w. For example, in the second iteration, there are only four programs $y, 1 + 1, x + y$ and $y + x$ satisfying 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. So, the objective value of input $\langle 1, 0\rangle$ in the second iteration is 6, including four pairs of the same program and two pairs of $x + y$ and $y + x$ in different orders. Table 1 shows that the min-pair strategy achieves the optimal selection in the sample task. It finishes the synthesis using only three iterations.

2.3 Approximating the Objective Function of the Min-Pair Strategy

Though the min-pair strategy is effective in reducing iterations, applying it in practice is difficult. Directly evaluating its objective function requires enumerating all programs in the program space, calculating their outputs, and then counting the number of pairs that are the same. Such a procedure costs so much time that is not acceptable in practice since the size of the program space is usually extremely large. Motivated by this point, this paper targets to approximate the min-pair strategy in practice and to do this, we propose an efficient approximation approach for its objective function.

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	$x+y$			

Main idea. In this paper, 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 the whole program space is composed of these subspaces according to the grammar rules.

To ensure efficiency on such a program space, we would like our approximation to be compositional as well, that is, its result can be calculated from the results on sub-program spaces. To calculate a compositional approximation, 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 size of the grammar) 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, we take a special case as an instance, 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 sub-program space is about counting the number of pairs of sub-programs expanded from (T, T) that output the same. However, this result is not enough to calculate the objective value. For those pairs where sub-programs output differently, their outputs may still be the same as different pairs of integers may have the same sum.

To enable a compositional approximation, the main idea of our approach is to estimate the complex behavior of operators with probabilities. In the above special case, our approach approximates + as a random function and assumes that, for any two pairs of different integers, the probability for this random function to output the same is always a fixed constant c. Then, the objective value can be naturally estimated as the expected number of pairs of programs that output the same under the approximated operator. This approximation is compositional. The subtask here is to calculate the expected number of pairs of sub-programs that output the same. When this result is w, the approximation can be calculated as $c + c(81 - w)$, because (1) there are 81 pairs of programs in total, and (2) each pair where sub-programs output differently has a probability of k to output the same.

Dealing with examples. In the general cases, our approximation is not as straightforward as the above special case. When there are examples available, the objective function considers only those programs satisfying all these examples. The method we used in the special case cannot deal with such constraint since our method considers only whether the outputs of two programs are the same instead of what the concrete output of a program is.

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

- For any concrete two programs, we can estimate the probability for them to output the same, after estimating the behavior of operators with probabilities as we did in the special case.
- In an OGIS iteration, there is a PBE result available, which is a program satisfying all examples.

Therefore, for each program, our approach calculates the probability for it to satisfy all examples as the probability for it to output the same as the PBE result on all examples, where the latter can

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		
...			...

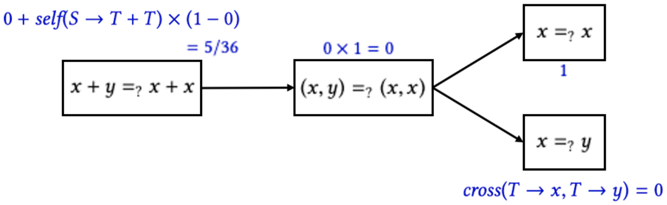

Fig. 1. The procedure for the model listed in Table 2 to estimate the check between $x + y$ and $x + x$.

be calculated in the same way as we did in the previous special case. In this way, we approximate the objective value for general cases, and the approximation is still compositional.

Unified-equivalence model. In the following, we discuss more on how our approach estimates the probability for two programs to output the same. The estimation is conducted by a *unified-equivalence model* in a compositional manner, similarly as discussed in the previous special case.

In our usage, a unified-equivalence model is bound with a grammar and a concrete input, meaning that this model is specific for estimating the probability for two programs in this grammar to output the same on this input. Table 2 shows a part of a unified-equivalence model for grammar G_e and input $\langle 0, 2 \rangle$. It includes two series of parameters.

- Parameters in *self* deal 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.
- Parameters in *cross* deal with the other case where two programs are expanded from different rules. For each pair of different rules (r, r'), parameter $cross(r, r')$ estimates the probability for two programs expanded from r and r' respectively to output the same.

Besides, the model uses the relation between the syntax and the outputs to improve the precision. That is, two programs must output the same when (1) they are expanded from the same grammar rule, and (2) the outputs of their sub-programs are the same.

The direct usage of a unified-equivalence model is to estimate whether two concrete programs output the same. The estimation proceeds recursively. Figure 1 shows how the model listed in Table 2 estimates 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, arrows show the control flow, and blue expressions show how the result is calculated. To estimate for $x + y$ and $x + x$, the model first estimate the probability for their sub-programs, (x, y) and (x, x), to output the same. The result shows that this probability is 0. Therefore, there is a probability of 0 that the outputs of $x + y$ and $x + x$ must be the same, and in the other case, the model regards the probability for $x + y$ and $x + x$ to output the same as $self(S \to T + T) = 5/36$. So, the estimation provided by the model is equal to $0 + 5/36 \times (1 - 0) = 5/36$.

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.
1   candidateInps ← GetCandidateInputs($\mathbb{I}, p_c$);
2   bestObj ← $+\infty$;   bestInp ← $\perp$;
3   exampleModels ← {Learn($G$, inp) | (inp ↦ oup) ∈ examples};
4   foreach inp ∈ candidateInps do
5       model ← Learn($G$, inp);
6       obj ← Approximate($G$, model, exampleModels);
7       if obj < bestObj then (bestObj, bestInp) ← (obj, inp);
8   end
9   return bestInp;
```

Based on the unified-equivalence model, our approximation of the objective value in the min-pair strategy is defined in three steps. First, it enumerates all pairs of concrete programs in the program space. Then, it uses unified-equivalence models to estimate the probability for each pair to be a pair of remaining programs that output the same. At last, our approximation adds up the estimations for all pairs and takes the summation as the result. It would not be surprising that this approximation is compositional, given that (1) the program space specified by a grammar is compositional, and (2) the estimation of unified-equivalence models proceeds recursively according to the structure of programs.

Learning unified-equivalence models. The estimation given by the model relies on its parameters, hence a crucial problem is how to learn proper assignments to enable high-quality estimations. To solve this problem, we first define a series of ideal models, namely *natural unified-equivalence models*, whose precision is guaranteed in theory. Then, we design a learning algorithm based on sampling, which efficiently returns a model close to the ideal one.

As mentioned before, every unified-equivalence model in our approach is specific for estimating equivalence between programs in a fixed grammar on a fixed input. Given the grammar and the input, our learning algorithm learns a corresponding model by (1) sampling several pairs of programs for the program space, (2) evaluating these programs on the given input and checking whether the outputs are the same for each sampled pair, and then (3) calculating proper parameters from the check results. Our learning algorithm is efficient, and thus we learn models on demand while selecting inputs, that is, our approach learns a new model whenever a new input is involved.

It is worth noting that the unified-equivalence model is not for estimating whether two programs output the same precisely. This model is designed for simplifying 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, that is, it provides an unbiased estimation for the number of pairs of programs that output the same (Theorem 4.8). Moreover, we provide an operator, namely *flattening*, that can further improve the precision of our model on every single estimation (Theorem 4.11).

2.4 Generation and Selection in *LearnSy*

We propose a question selector *LearnSy* based on our approximation, as shown in Algorithm 1. *LearnSy* runs in a generate-then-select manner. It first generates a set of candidate inputs (Line 1) and then selects the best input among them using the approximated objective values (Lines 2-9).

The generation varies with the synthesis task. When a verifier for the target program is available, we can apply the CEGIS framework [Solar-Lezama et al. 2006] and generate those inputs where p_c outputs incorrectly. Otherwise, we can apply the framework of interactive program synthesis [Le et al. 2017] and generate those inputs that can distinguish at least two remaining programs. In some cases, the number of different possible inputs may be large. Therefore, in our implementation, we set an upper bound on the number of candidate inputs to limit the time cost.

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

In our motivating example, there is a verifier available, and thus the generation of candidate inputs can follow the CEGIS framework. At this time, *LearnSy* selects the input with the smallest approximated objective value among those inputs where the output of the PBE result is not $x + y$. *LearnSy* can achieve the same effect as the min-pair strategy on this task (i.e., finishes synthesis in three iterations) when the number of samples used for learning is large enough.

3 THE MIN-PAIR STRATEGY

Now, we introduce the min-pair strategy and those necessary notations. For simplicity, we keep the other related concepts informal, such as the OGIS framework [Jha and Seshia 2017] and the question selection problem [Ji et al. 2020a], because this paper focuses only on approximating the min-pair strategy. The usages of these concepts in this paper are just the same as the literature.

3.1 Preliminaries

A task of program synthesis is usually discussed above an underlying domain $\mathbb{D} = (\mathbb{P}, \mathbb{I}, \mathbb{O}, [\![\cdot]\!]_{\mathbb{D}})$, where $\mathbb{P}, \mathbb{I}, \mathbb{O}$ 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 ranges of possible programs and concerned inputs as well as provides the corresponding semantics for the programs.

In this paper, we make two assumptions on the underlying domain. First, for simplicity, we assume that there is a universal output space and a universal oracle function $[\![\cdot]\!]$ for all domains, and thus abbreviate a domain as a pair $(\mathbb{P}, \mathbb{I})$ of a program space and an input space.

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

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

- N is a finite set of non-terminals, $s_0 \in N$ represents 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 $s \to f^{(k)}(s_1, \dots, s_k)$, where $s, s_1, \dots, s_k$ are non-terminals in N and $f^{(k)}$ is a symbol in Σ.

A program is in G if it can be expanded from s_0 by applying a finite number of grammar rules.

Similar to previous question selectors [Ji et al. 2020a; Tiwari et al. 2020], our approach further requires the program space $\mathbb{P}$ to be finite, i.e., the RTG specifying $\mathbb{P}$ is acyclic. This is not a critical limitation because in practice we can always truncate an infinite program space to finite, e.g., by setting a large enough size limit. For our approach, we provide an iterative method for adapting the truncation on demand, which will be discussed in Section 6.

In the remainder of this paper, we use 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. We highlight this predicate here because it will be frequently used in our approach.

3.2 The Min-Pair Strategy

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

The min-pair strategy is based on the following intuition: in order to minimize the number of iterations needed, in each turn, the selected input should exclude as many programs as possible. Concretely, this strategy assumes that there is a prior distribution over the program space, representing the probability for each program to be the target in practice, and then 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 the program space, and a set E of existing input-output examples, the min-pair strategy selects the input I from $\mathbb{I}$ that minimizes the following objective function.

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

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 $\mathbb{P}$ satisfying all examples in E. The meaning of this objective function is the probability for the random program p to satisfy all examples after input I is selected.

The above function can be transformed into the following equivalent form, where $\varphi(p)$ represents the probability of program p in distribution φ, and $[\cdot]$ is a function mapping *true* to 1, *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. In Section 2, we discussed a simplified form of the min-pair strategy, where the objective function is the number of pairs of remaining programs that output the same. This case is equivalent to the above formula when φ is a uniform distribution over the program space $\mathbb{P}$.

The transformed objective function can be expressed using only equivalence checks after taking the PBE result p_c into consideration. 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. In this paper, we consider the following form by default.

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

Recall that we assume the program space $\mathbb{P}$ is specified by a regular-tree grammar. To efficiently access the probabilities, we further assume that φ is 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 $\gamma : R \mapsto [0, 1]$ assigned probabilities to each rule. Function γ needs to ensure the sum of the probabilities of 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 represents 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 shows the PRTG specifying a uniform distribution over grammar G_e[2] discussed in Section 2, where $r[w]$ represents that the probability assigned to rule r is w.

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

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 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 at last introduce the flattening operation, which can improve the precision of our model (Section 4.4).

4.1 Unified-Equivalence Model

As mentioned in Section 2, to get an approximation that can be efficiently calculated, the unified-equivalence model approximates the behavior of operators with simple random events and estimates the result of equivalence checks with probabilities.

The unified-equivalence model approximates a recursive procedure *receq* for evaluating equivalence checks, which uses the syntax of programs as much as possible to make decisions. Given programs p, p' and input I, *receq* evaluates predicate $[\![p]\!](I) = [\![p']\!](I)$ as below.

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

where r is the first grammar rule used by p, n is the arity of the symbol in r, $p_1, \dots, p_n$ are the sub-programs of p, and $r', n', p'_1, \dots, p'_{n'}$ are those counterparts related to program p'. In the second case, *receq* uses the fact that the outputs of p and p' must be the same when the outputs of their sub-programs are equal, since 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, $\mathit{receq}(p, p', I)$ 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. 2023].

In the definition of *receq*, there are still two cases relying on the semantics of operators. To avoid these usages, the unified-equivalence model approximates them using Bernoulli trials. It involves two series of parameters, *self* and *cross*. They represent the probability for the model to return *true* while 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 $\mathit{self}: R \mapsto [0, 1]$ and $\mathit{cross}: R \times R \mapsto [0, 1]$. These parameters assign probabilities to each grammar rule and each pair of different grammar rules respectively.

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

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

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

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

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 as 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 specifically for equivalence checks (1) between programs drawn from the PRTG and (2) on the given input. Our algorithm is designed in two steps. First, we highlight a subclass of unified equivalence models, namely *natural unified-equivalence model*, whose precision is guaranteed in theory. Then, we design an algorithm for efficiently learning a model that is close to the natural model by sampling.

In a natural unified-equivalence model, the value of each parameter exactly matches its effect in estimation. From the definition of unified-equivalence models (Definition 4.2), we can see that each parameter in the model has a clear role in estimating equivalence checks.

- Parameter $\mathit{self}(r)$ estimates the probability for two programs to output the same under the condition 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.

One important observation here is that given a 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.

Given the above observation, a natural idea to get an effective unified-equivalence model is to set each parameter to the probability it estimates. We call such an assignment as the natural assignment to a parameter and then denote the model where each parameter is set to its natural assignment as 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, where each parameter is set as the following.

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

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

where p_i and p'_i represent the ith 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, where the probability of selecting each program is directly proportional to its probability assigned by φ.

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

In a unified-equivalence model, $\mathit{self}(S \to T + T)$ is used when the two programs are both expanded from $S \to T + T$ and their sub-programs output differently. There are 9 programs expanded from this rule, and their probabilities are all the same on the given PRTG. Among them, $1, 2, 3, 2, 1$ programs output $0, 1, 2, 3, 4$ on input $\langle 2, 0 \rangle$ respectively, and the outputs of their sub-programs are different from each other, vary from $(0, 0)$ to $(2, 2)$. Therefore, there are 72 pairs of programs expanded from

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

```
Input: A PRTG $\varphi$ with rule set $R$, an input $I$, and two constants $n_s, w_{default}$.
Output: A unified-equivalence model learned for $\varphi$ and $I$.
$self \leftarrow \{\}$;  $cross \leftarrow \{\}$;
foreach $r \in R$ do
    $(p_1, \dots, p_{n_s}) \leftarrow$ Sample$(r, \varphi, n_s)$;  $(p'_1, \dots, p'_{n_s}) \leftarrow$ Sample$(r, \varphi, n_s)$;
    $n_1 \leftarrow$ the number of $i \in \{1, \dots, n_s\}$ such that IsSubDiff$(p_i, p'_i, I)$;
    $n_2 \leftarrow$ the number of $i \in \{1, \dots, n_s\}$ such that IsSubDiff$(p_i, p'_i, I)$ and $[\![p_i]\!](I) = [\![p_i]\!](I')$;
    if $n_1 > 0$ then $self(r) \leftarrow n_2/n_1$ else $self(r) \leftarrow w_{default}$;
end
foreach $r, r' \in R$ do
    $(p_1, \dots, p_{n_s}) \leftarrow$ Sample$(r, \varphi, n_s)$;  $(p'_1, \dots, p'_{n_s}) \leftarrow$ Sample$(r', \varphi, n_s)$;
    $cross(r, r') \leftarrow$ (the number of $i \in \{1, \dots, n_s\}$ such that $[\![p_i]\!](I) = [\![p'_i]\!](I))/n_s$;
end
return $(self, cross)$;
```

$S \to T + T$ whose sub-programs output differently on input $\langle 2, 0 \rangle$, and 10 pairs of programs among them output the same. So the natural assignment of parameter $self(S \to T + T)$ is $10/72 = 5/36$.

As shown in the following lemma, the natural unified-equivalence model **precisely** captures the overall equivalence between programs on its PRTG and its input. Concretely, when the 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 on 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. As shown in Formula 3 and 4, the natural assignments to the parameters rely on the outputs of all related programs, and thus calculating these natural assignments faces 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 of selecting each program is directly proportional to its probability assigned by φ.
- `IsSubDiff`(p, p', I) checks whether the sub-programs of p, p' output differently on input I, i.e., it returns $\vee_i [\![p_i]\!](I) \neq [\![p'_i]\!](I)$, where $p_i(p'_i)$ represent the ith sub-program of $p(p')$.

In brief, for each parameter, Algorithm 2 approximates its natural assignment by (1) generating n_s pairs of random programs, and (2) calculating the corresponding probability only on the samples.

Algorithm 2 is efficient. When the number of samples is fixed, its time cost is polynomial to the size of the grammar, a number usually small. It allows us to learn models on demand while selecting inputs, without any obvious loss of efficiency.

As shown in the following lemma, our learning algorithm provides an unbiased estimation for the natural unified-equivalence model. That is, the expected estimation given by the learned model is always equal to the estimation given by the natural unified-equivalence model.

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

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

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

By combining the above lemma and the lemma we proved for natural unified-equivalence models, we know that the learned model also provides an unbiased estimation of the real probability for two random programs to output the same.

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

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

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

4.4 Improving the Precision by Flattening

Though we have provided some precision guarantees for the learned model (Theorem 4.8), in most cases, the estimations provided by the model are not completely accurate. For example, the natural unified-equivalence model discussed in Example 4.5, which is for input $\langle 0, 2\rangle$, estimates the equivalence check between $x + y$ and $x + 1$ as $5/36$, but the ground truth is 0. In this section, we introduce an operator on the program space and the PRTG, namely *flattening*, which can further improve the precision of unified-equivalence models.

As shown below, the flattening operator transforms the program space by flattening a selected grammar rule into a series of 0-arity rules, each corresponding to a program expanded from the selected rule, and meanwhile keeps the probability of each program unchanged.

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 as below.

(1) Let s be the start non-terminal of rule r^*, P be the set of programs expanded from rule r^*.
(2) The set of non-terminals and the start non-terminal keep unchanged.
(3) For each program $p \in P$, a 0-arity symbol $\square_p$ is added, whose semantics is the same as p.
(4) Rule r^* is removed and rule $s \to \square_p$ is added for each program $p \in P$.
(5) The probabilities assigned to existing rules are unchanged, while that for the new rules are equal 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, we can get another grammar G', as shown below.

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

To see how the flattening operation improves the accuracy of estimations, let us 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 on the two grammars are shown below. Their calculations are straightforward, and thus we omit the details for simplicity.

- Let $\mathcal{M}_1$ be the natural-unified equivalence model for (1) grammar G, (2) input $\langle x = 0, y = 1\rangle$, and (3) the uniform distribution over the program space. In this model, parameter $\mathit{cross}(S \rightarrow S_1 + S_1, S \rightarrow S_1 - S_1)$ is equal to $1/4$, 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 (1) grammar G', (2) input $\langle x = 0, t = 1\rangle$, and (3) the uniform distribution. In this model, parameter $\mathit{cross}(S \rightarrow S_1 + S_1, S \rightarrow \Box_{x-y})$ is 0, and thus estimation $\mathcal{M}_2(x + y, \Box_{x-y})$ is also 0, which is accurate.

As we can see, in this case, the accuracy of the estimations given by the natural-unified equivalence model is improved after applying the flattening operation.

The following theorem generalizes the case study we made in the above example. It tells that the precision of the natural unified-equivalence model can never be harmed after applying the flattening operation. Intuitively, the flattening operation reduces the average number of operations used by programs. Such a reduction benefits the precision of unified-equivalence models, as the loss of a unified-equivalence model comes from approximating the behavior of operators.

Theorem 4.11. *Given two programs p, p', an input I, and a unified-equivalence model $\mathcal{M}$, define $\mathit{same}(p, p', I, \mathcal{M})$ be the probability for the estimation of $\mathcal{M}$ to be exactly the same as the recursive procedure $\mathit{receq}(p, p', I)$ it approximated, that is, the result of each estimation happens in $\mathcal{M}(p, p')$ is equal to the ground truth in $\mathit{receq}(p, p', I)$*[3]*. Given an input I and a PRTG φ, define $\mathit{acc}(\varphi, I)$ as a function measuring the accuracy of the natural model $\tilde{\mathcal{M}}^I_\varphi$, as shown below.*

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

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

Moreover, if the flattening operation is repeatedly applied, the grammar will ultimately become completely flattened[4], where every program has been flattened into a separate grammar rule. At this time, the estimation given by the natural unified-equivalence model will become completely accurate, because the ground truth of every equivalence check will be recorded into *cross* parameters, as shown in the following example.

Example 4.12. Continuing with the previous example, after further flattening rule $S \rightarrow S_1 + S_1$ in grammar G', we can get the following completely flattened grammar G''.

$$S \rightarrow \Box_{x+x} \mid \Box_{x+y} \mid \Box_{y+x} \mid \Box_{y+y} \mid \Box_{x-x} \mid \Box_{x-y} \mid \Box_{y-x} \mid \Box_{y-y}$$

Let $\mathcal{M}_3$ be the natural unified-equivalence model for (1) grammar G'', (2) input $\langle x = 0, y = 1\rangle$, and (3) 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 the value of parameter $\mathit{cross}(S \rightarrow \Box_{p_1}, S \rightarrow \Box_{p_2})$. The latter is further equal to the ground truth of the equivalence check between p_1 and p_2 on input $\langle x = 0, y = 1\rangle$, by the definition of natural unified-equivalence models.

It is worth noting that there is a trade-off between precision and efficiency while using flattening. Though applying flattening can improve the precision of the unified-equivalence model, it always increases the size of the grammar, and thus increases the time cost of our approach.

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

5 APPROXIMATING THE MIN-PAIR STRATEGY

In this section, we introduce our approximation (Section 5.1) and then 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. In Section 3.2, we show an equivalent form of the original objective function (Formula 2), made up of equivalence checks. Then, in Section 4, we design the unified-equivalence model and show how it estimates equivalence checks in a compositional manner. Our approximated objective function is obtained by replacing each equivalence check in Formula 2 with the estimation given by a learned unified-equivalence model, for instance, equivalence check $eq(p, p', I)$ is replaced with estimation $\mathcal{M}_I(p, p', I)$, where $\mathcal{M}_I$ represents 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 $\mathcal{M}_I$ represents the unified-equivalence model learned for input I, i.e., $\texttt{Learn}(\varphi, I)$.

Example 5.2. Let us 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 following formula shows the approximated objective value of input $\langle 1, 0\rangle$ in this iteration, when the given PRTG is φ.

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

Recall that our approximation is not designed for precision. Our main purpose is to design a compositional approximation to enable efficient question selection. Nevertheless, the following two results show that the precision of our approximation is ensured in some cases. Both of them are direct corollaries to the properties of unified-equivalence models discussed in Section 4.

First, 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 on the given input, i.e., the probability for two random programs to output the same. Therefore, the property of unbiased estimation is direct from Theorem 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`.

Second, our approximation is precise when the program space is completely flattened. This is because (1) the loss of our approximation comes from the loss of the estimations given by unified-equivalence models, and (2) these estimations are accurate when the program space is completely flattened, as discussed in Section 4.4.

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

We use the compositionality of the approximation to calculate it efficiently. Our algorithm works in a divide-and-conquer manner: it decomposes the task into tasks on sub-program spaces, conquers each of them recursively, and then combines the sub-results together. We observe that the number of different subtasks in this procedure is bounded, and thus we further sped up the calculation by dynamic programming, i.e., by reusing results among identical subtasks.

To illustrate our algorithm clearly, let us first consider two concrete cases on our sample program space G_e, where the first case is simple and is for illustrating the full procedure, and the second case is a little more complex and is for supplying some missing key points. Besides, we make the following two assumptions 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 term and the last term correspond to the case where p, p' are both expanded from rule $S \to 1$ and $S \to T + T$ respectively. We calculate each of these terms separately and then get the approximated objective value by summing these results up.

We calculate each term by (1) performing a thorough unfolding, (2) extracting tasks on sub-program spaces from the unfolded formula and solving them recursively, and (3) combining these sub-results together. Let us take the last term in the above formula as an example, as it corresponds to the most complex case. The unfolding proceeds in four steps, as shown below.

(1) The summation over programs expanded from $S \to T + T$ can be unfolded into summations for sub-programs, each ranges 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 following formula shows the results so far, where each parameter is replaced by its concrete assignment, and $p \in T$ represents that p is 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, we unfold the inner plus (the blue part) into two separate summations and then reorganize them as below, where we ignore a global coefficient of 9/32 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 part must be 1 by the definition of PRTG and the second part is formed by two separate sub-summations. Both sub-summations are in the same form as the original task. Each of them 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 pairs of programs expanded from the non-terminal. Therefore, the two sub-summations are subtasks of the original task and thus can be calculated recursively. Moreover, these two subtasks are identical as they correspond to the same non-terminal. Therefore, we need only to solve one and then can directly reuse this result for the other.

Case 2: single example. The procedure in the previous example is almost the same as the general algorithm, except the form of subtasks is still partial due to the lack of examples and the PBE result. To see this point, let us move to a more general case, the approximated objective value in Example 5.2, where both the example and the PBE result are involved.

To calculate this value, the first several steps are the same as the first case, including (1) dividing the summation according to the first rules, and (2) unfolding the summation, probabilities of programs, and the estimations for each term. We still focus on the term where both programs are expanded from $S \to T + T$, and now this term is transformed into the formula below[5], where a global coefficient of 9/128 is ignored for simplicity.

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

The next step is to unfold the inner plus (the blue part) into separate summations. Here, the formula is more complex than that in the first case, since there are two inner pluses. At this time, the unfolding results in four sub-summations, 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. Concretely, these sub-summations are in the following form, where $\overline{\mathcal{M}}$ represents the list of considered models. It 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.

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

We can unify the two parts in the above formula and the original task via a more general definition of subtasks. Each subtask here is specified by a non-terminal, an optional concrete program (which is used 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 pairs of programs (p, p') expanded from the non-terminal. As will show later, this form is already the same as that in the general algorithm, and in this case, we can solve both parts in Formula 6 recursively through this unification.

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. This algorithm is naturally generalized from the procedure discussed in Section 5.2.

Subtasks. Given a PRTG φ in the context, 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 pairs of programs (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 $\overline{\mathcal{M}}_c$. Formula, the task is to calculate the formula below.

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

[5] Here, we still distinguish the two models learned for input ⟨0, 2⟩ and ⟨1, 0⟩ for clarity of description. But according to the assumption we made in this section, parameters in these two models are all 1/2.

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

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

Example 5.5. Now we correspond the above definition to the tasks we have seen before. 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.
- Given PBE result p_c and a set E of examples, 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 models learned for inputs involved in E, i.e., $[\mathcal{M}_{I'} \mid (I' \mapsto O') \in E]$.

Procedure. The procedure of our algorithm is almost the same as we discussed in Section 5.2, and thus we only make a summary here. Given a subtask, our algorithm runs in the following steps, with a memoization table restoring the results of visited examples.

(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 all inner plus via the distributive law.
(4) Express each case using subtasks and then recurse into each of them.
(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 for processing 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 size of the grammar 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 this algorithm to calculate an approximated objective value is polynomial to the size of the grammar and exponential to the number of existing examples.

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

6 IMPLEMENTATION

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

Dealing with infinite program spaces. As assumed in Section 3.2, our approximation requires the program space to be finite. We generalize it to the infinite case via an iterative procedure for truncating the program space. Given an infinite program space, *LearnSy* iterates with a depth limit lim_d and works on a finite subspace including only programs whose depth is at most lim_d. During synthesis, each time when the depth of the PBE result exceeds the depth limit, *LearnSy* enlarges the limit lim_d to the depth of the PBE result to include this program into the considered subspace.

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

Generating candidate inputs. As discussed in Section 2.4, *LearnSy* uses a generator to generate a set of candidate inputs. We implement two 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 can distinguish the PBE result with some other remaining program. Our generator uses an encoding provided by Ji et al. [2020a], which encodes the above condition into an SMT formula, 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. Our generator assumes that the complete specification of the target function 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 (1) the non-terminal corresponds to this vertex, and (2) the first grammar rule applied to this vertex. Then, for each grammar rule r expanding from non-terminal r, 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, we get the sample programs 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 while solving tasks in the other half.

Other Configurations. We repeatedly apply 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.

Besides, to ensure efficiency, 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

For interactive tasks, our evaluation is conducted on the interactive synthesizer proposed by Ji et al. [2020a]. We implement three different variants of this synthesizer for *LearnSy* and three other baseline selectors, and then compare them on the two datasets collected by Ji et al. [2020a].

Baseline. We consider two 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 for reducing iterations. Similar to our approach, it approximates a strategy for building effective decision trees, but its approximation is based on sampling remaining programs. To ensure the response time, *SampleSy* (1) requires efficient witness functions [Polozov and Gulwani 2015] for all operators in the domain, and (2) performs sampling at background while the user is answering the question.

Dataset. We use the two dataset $\mathcal{I}_S$ and $\mathcal{I}_R$ collected by Ji et al. [2020a]. These datasets are 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.

- String dataset $\mathcal{I}_S$ includes 150 benchmarks. In this dataset, the spec 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}.
- Repair dataset $\mathcal{I}_R$ includes 16 benchmarks extracted from the program repair process for real-world Java bugs. In this dataset, the spec is provided as an SMT formula, and the average size of the program space is 2.4×10^{8}.

Configurations. Our experiments are conducted on Intel Core i7-8700 3.2GHz 6-Core Processor. For each execution, we set the time limit to 30 minutes and the memory limit to 8GB, that is, we will regard a synthesis procedure on it as failed if its total time cost exceeds 30 minutes, or it uses more than 8GB memory at some moment. Besides, since many approaches considered in our experiment are random, we repeat each execution 3 times and consider only the average performance to reduce the affect of randomness.

To simulate the interactive scenario, we also 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. Therefore, 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. At this time, the response time of *LearnSy* never exceeds 2 seconds in our dataset, with an average time of 0.91 second per iteration.
- For *RandomSy*, it satisfies this requirement on all tasks in our dataset and thus does not require extra configurations.

The configuration for the other baseline selector, *SampleSy*, is more complex. As mentioned before, *SampleSy* selects the input based on a set of samples drawn from all remaining programs. In each iteration, *SampleSy* runs in two stages.

1. In the first state, *SampleSy* performs sampling. This stage proceeds in the background while the user is answering the previous question, and thus its time limit is related to how fast the user can respond to the questions. The default implementation of *SampleSy* assumes that the user requires at least 2 minutes to answer each question and then sets the time limit for sampling to 2 minutes.
2. In the second stage, *SampleSy* selects an input according to the samples. This stage proceeds online after the user responds to the previous question, and thus its cost matters the response time. We set the time limit for this stage to 2 seconds, the same as the other two selectors.

Besides the above configuration, we consider several other variants of *SampleSy* to simulate the cases where questions have different 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 $\mathcal{I}_S$ is about

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

Dataset	Selector	#Solved	#Iteration				
			Ave	RED. for All		RED. for Hard	
$\mathcal{I}_S$	*LearnSy*	131	2.81				
	RandomSy		3.16	0.35	11.0%	0.81	13.2%
	SampleSy$_3$		2.81	0.01	0.18%	0.59	10.0%
	SampleSy$_5$		2.79	−0.02	−0.72%	0.38	6.79%
	SampleSy$_{10}$		2.80	0.00	−0.18%	0.44	7.62%
	SampleSy$_{15}$		2.78	−0.02	−0.86%	0.33	5.94%
	SampleSy		2.78	−0.04	−1.51%	0.21	3.74%
	The average time cost of *LearnSy* is 1.11 seconds per iteration						
$\mathcal{I}_R$	*LearnSy*	16	7.94				
	RandomSy		10.4	2.50	24.0%	24.3	55.7%
	SampleSy$_3$		8.40	0.46	5.46%	3.33	15.6%
	SampleSy$_5$		8.35	0.42	4.99%	3.67	16.9%
	SampleSy$_{10}$		8.10	0.17	2.06%	2.33	11.5%
	SampleSy$_{15}$		8.15	0.21	2.56%	2.67	12.9%
	SampleSy		7.83	−0.10	−1.32%	−0.66	−3.57%
	The average time cost of *LearnSy* is 0.32 seconds per iteration						

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 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. Motivated by this observation, we introduce a series of variants of *SampleSy*, denoted as *SampleSy*$_t$ for some integer 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.

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

The results show that *LearnSy* significantly outperforms *RandomSy*, and thus we focus our discussion on the comparison between *LearnSy* and *SampleSy*. From the results, we can see that the performance of *LearnSy* is competitive compared with *SampleSy*.

- First, *LearnSy* outperforms *SampleSy* when the time limit for sampling is 3 seconds.

- Second, the advantage of *LearnSy* compared with *SampleSy* is still clear even when the time limit for sampling is enlarged to 15 seconds. *LearnSy* outperforms *SampleSy*$_{15}$ on dataset $\mathcal{I}_R$ and achieves almost the same performance on dataset $\mathcal{I}_S$.
- At last, *LearnSy* uses only slightly more iterations than *SampleSy* when the time limit for sampling is 2 minutes. Note that such a time cost of sampling is already more than 100 times larger than the average time cost of *LearnSy*.

Besides, the results show that the advantage of *LearnSy* compared with *SampleSy* increases on the hard tasks when the sampling time of *SampleSy* is limited within 15 seconds. Such a tendency is related to the difficulty of sampling remaining programs. On those simple tasks, *SampleSy* can get enough samples quickly so that the limited sampling time does not matter. In contrast, on those difficult tasks, *SampleSy* can hardly collect enough samples within 15 seconds, and thus its performance becomes much worse. We believe the performance of a selector on hard tasks is more important because the burden on the user while solving these tasks is usually heavier.

At last, we would like to emphasize again that the performance of *LearnSy* is achieved **without losing any generality**. It uses only the semantics of programs and the grammar structure of the program space and thus can be applied to all tasks under the SyGuS framework. In comparison, *RandomSy* is also general, but its performance is much worse; *SampleSy* has a close performance, but it can only be applied to those domains with efficient witness functions.

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

For non-interactive tasks, our evaluation is conducted under the CEGIS framework [Solar-Lezama et al. 2006]. To form CEGIS solvers, we consider three state-of-the-art PBE solvers. We implement multiple CEGIS solvers by combining each PBE solver with *LearnSy* and several baseline selectors respectively and evaluate these CEGIS solvers on three datasets in SyGuS-Comp.

PBE solvers. *LearnSy* regards the PBE solver as a black box and thus can be combined 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 where efficient witness functions [Polozov and Gulwani 2015] are available, 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. Because no previous question selector can improve the efficiency of CEGIS solvers, we compare *LearnSy* with the default selector used by 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, we notice that in some previous studies on domain-specific PBE solvers, some domain-specific heuristics for selecting inputs have been proposed as optimizations [Jha et al. 2010; Padhi et al. 2018]. We also consider these heuristics as baselines in this experiment.

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

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

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

Dataset	Solver	Selector	#Solved	#Iteration			TimeCost (s)				
				Ave	RED. for All		Ave	RED. for All		RED. for Hard	
$\mathcal{C}_S$	*Eusolver*	*LearnSy*	**145**	**2.51**			**8.98**				
		Default	144	2.61	0.10	3.69%	9.40	0.42	4.50%	4.58	5.93%
		SigInp	**145**	2.69	0.18	6.83%	9.32	0.34	3.67%	2.76	3.23%
	MaxFlash	*LearnSy*	155	**1.41**			**0.97**				
		Default	155	1.50	0.08	5.60%	1.27	0.29	23.1%	3.27	29.3%
		SigInp	**158**	1.47	0.06	3.78%	1.94	0.97	49.9%	8.68	50.0%
$\mathcal{C}_I$	*Eusolver*	*LearnSy*	**43**	**18.5**			**13.6**				
		Default	**43**	23.7	5.19	22.0%	17.3	3.63	21.0%	25.6	20.1%
	PolyGen	*LearnSy*	**82**	**31.8**			**4.87**				
		Default	79	39.9	8.14	20.4%	5.66	0.79	13.9%	13.1	26.9%
$\mathcal{C}_B$	*Eusolver*	*LearnSy*	**424**	**10.2**			**23.1**				
		Default	**424**	11.0	0.62	5.60%	24.6	1.53	6.24%	12.1	8.42%
		Biased	**424**	11.1	0.65	5.83%	24.3	1.12	4.93%	8.12	5.79%

- *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 those inputs whose profile is different from the inputs in existing examples[7].
- *Biased Bit-Vector* [Jha et al. 2010] (denoted as *Biased*) is specialized for the domain of bit-vectors. It uses the property that for many bit-vector functions, the rightmost bits in the input influence the output more than the leftmost bits [Warren 2002]. Therefore, it prefers those inputs whose rightmost bits are different from the inputs 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, including the size[8], the underlying domain, the form of the provided spec, and applicable synthesizers and baseline selectors.

Configuration. The configurations are similar to those for interactive tasks except of two changes. First, we decrease the time limit of each execution to 5 minutes to save time, since non-interactive tasks are much more than interactive ones. Second, in *LearnSy*, we set the number limit of rules introduced by flattening to 100 to ensure that the time cost of the question selection does not affect the efficiency of CEGIS solvers much. Under this setting, the time cost of *LearnSy* is about 0.03 second per iteration, which is negligible for most cases.

[7] *Significant input* is proposed as an application of *FlashProfile* [Padhi et al. 2018], a synthesizer for syntactic profiles, and uses a cost function that relies on several parameters. Because its original implementation is not released, we re-implement it in our evaluation using *FlashProfile* and simply set all missing parameters as a default constant.

[8] The original dataset of $\mathcal{C}_B$ includes 750 tasks. Many of these tasks are the same except for the number of provided examples. Therefore, to save time, we consider only the one with the most number of examples for each group of duplicated tasks.

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 achieved by *LearnSy* in the number of used iterations (Column "#Itertion-RED. for All"), the average time cost (Column "TimeCost-Ave"), the absolute/relative reduction achieved by *LearnSy* in the time cost (Column "TimeCost-RED. for All"), and the reduction in the time cost on the hardest 10% of tasks (Column "TimeCost-RED. for Hard"). Here, we change the measurement of the difficulty of tasks as the total time cost of *LearnSy* and the baseline selector, because efficiency is the main pursuit while designing non-interactive synthesizers.

First, compared with the default selector, the CEGIS solver with *LearnSy* always uses fewer iterations and is always more efficient. The results show that, by replacing the default selector in the CEGIS framework with *LearnSy*, we can immediately speed up existing CEGIS solvers with a ratio of up to 23.1%. The value of such an improvement comes from two aspects.

- **Generality**. As a general selector, *LearnSy* can be applied to all tasks under the SyGuS framework no matter what the synthesis domain and the PBE solver are. Though the speedup achieved by *LearnSy* seems not so significant for an individual PBE solver on a concrete domain, such a speedup potentially exists for all PBE solvers and all domains. Therefore, the overall speedup should be considerable, given the generality of *LearnSy*.
- **Originality**. Though there exist multiple approaches that reduce the number of iterations, *LearnSy* explores a new method for question selection (i.e., estimating the behavior of operators to enable a compositional approximation) and **first** achieves speedups for OGIS solvers by question selection. The performance of *LearnSy* can benefit from a better method for estimating the operators, which is future work.

Second, for those heuristics, *LearnSy* outperforms *Biased* and achieves competitive performance compared with *SigInp*: *LearnSy* performs better on those jointly solved tasks while *SigInp* solves slightly more tasks when combined with *MaxFlash*. Note that such a result is already valuable because both *SigInp* and *Biased* are based on domain-specific heuristics designed by domain experts, while *LearnSy* is general.

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

(1) There is no efficient witness function available for the domain of integers and bit-vectors, and thus *SampleSy* cannot be used on dataset $\mathcal{C}_I$ and $\mathcal{C}_B$.
(2) To perform sampling, *SampleSy* requires to construct the full version-space algebra [Polozov and Gulwani 2015] for all remaining programs. As shown in the evaluation conducted by Ji et al. [2020b], the time cost for constructing the full version-space algebra is 400 times larger than that for *MaxFlash* to find the PBE result. Therefore, *SampleSy* can hardly accelerate *MaxFlash* on $\mathcal{C}_S$.
(3) The only possible case is to accelerate *Eusolver* on $\mathcal{C}_S$. Nevertheless, it is still difficult. In this experiment, *Eusolver* takes only 3.6 seconds per iteration on average when the default selector is used. Therefore, even using $\mathit{SampleSy}_{3s}$ will significantly increase the time cost of each iteration, and it is almost impossible to cover this cost by reducing iterations.

At last, the results in Table 5 show that the speedups achieved by *LearnSy* vary among different datasets, different PBE solvers, and tasks with different difficulties. In theory, there are some cases where *LearnSy* is likely to achieve higher speedups. We list these cases below.

[9] In this experiment, the tasks solved by different selectors are not always the same when the dataset and the PBE solver are the same. Therefore, to establish a fair comparison among the averages, we consider only those tasks jointly solved by all applicable selectors while calculating the average.

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*	**155**	**1.40**			**0.98**				
		LearnSy$_U$	154	1.42	0.02	1.24%	1.07	0.10	8.87%	1.04	11.7%
	4	*LearnSy*	**155**	**1.42**			**0.77**				
		LearnSy$_0$	**155**	1.46	0.04	2.93%	1.13	0.36	31.8%	3.77	39.8%

- *LearnSy* tends to perform better when more examples are required to solve the synthesis task. In this case, there is usually a larger space for reducing the number of iterations, and thus an effective question selector can lead to a more significant improvement. In Table 5, the domain of integers (dataset $\mathcal{C}_I$) matches this case.
- *LearnSy* tends to perform better when the time cost of the PBE solver increases rapidly with the number of examples. In this case, the same reduction in the number of iterations can result in a more significant reduction in the time cost. In Table 5, *MaxFlash* matches this case while the time cost of the other PBE solvers is affected less by the number of examples.
- *LearnSy* tends to perform better on harder tasks. Solving a harder task usually required more examples and thus matches the first case we discussed right above. In Table 5, we can see that the absolute improvement achieved by *LearnSy* always becomes more significant when only the hardest 10% of tasks are considered.

7.3 Exp 3: Comparison between Different Prior Distributions

LearnSy requires a prior distribution φ. As discussed in Section 6, our implementation of *LearnSy* uses 2-fold cross-validation to learn φ by default. In this experiment, we test how this distribution 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. We choose this case because *LearnSy* performs the best under this setting (as shown in Section 7.2) so that the results here can well reflect the relation between φ and the improvement achieved by *LearnSy*.

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 Exp 2.

Results. The results are summarized in Table 6 in the same way as Exp 2. The results show that *LearnSy* outperforms its invariant *LearnSy*$_U$ and thus imply a tendency that the precision of φ contributes positively to the performance of *LearnSy*. The more precise φ is, the better the performance of *LearnSy* will be. Besides, it is worth noting that though *LearnSy*$_U$ does not perform as well as *LearnSy*, it still outperforms the default CEGIS selector and the heuristics-based selector *SigInp* discussed in Exp 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 the default *LearnSy* repeatedly applies flattening until the number of introduced rules exceeds 100 while solving non-interactive tasks. In this experiment, we test how flattening affects the performance of *LearnSy*.

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

Baseline. We consider a variant of *LearnSy*, denoted as $\mathit{LearnSy}_0$, which does not use the flattening operation at all, and then compare it with the default version.

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

Results. The results are summarized in Table 6 in the same way as Exp 2. These results show that both the used iterations and the time cost of *LearnSy* are reduced after applying the flattening operations. The reduction of iterations provides indirect evidence that the flattening operation can improve the precision of our approximation, which matches our theoretical analysis.

8 RELATED WORK

Question selection for interactive program synthesis. The question selection problem for interactive program synthesis targets to reduce the burden on the user by reducing the number of iterations and has been studied by Ji et al. [2020a]; Tiwari et al. [2020]. Both previous studies adopt a greedy strategy for constructing effective decision trees and approximate the strategy by sampling from remaining programs. Compared with these approaches, *LearnSy* approximates via unified-equivalence models instead of sampling and thus (1) significantly improves the efficiency of question selection, and (2) can be applied to those domains where no efficient sampling algorithm is available. Besides, 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 this task is future work.

Multiple existing interactive synthesis systems [Ferreira et al. 2021; Mayer et al. 2015; Wang et al. 2017] also include a question selector for input-output examples. The target of these selectors is an input where at least two remaining programs output differently, and they 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 a valid input by invoking an SMT solver. We approximate all these approaches as *RandomSy* in our evaluation.

Question selection for CEGIS. As mentioned before, there have been several heuristics 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 as *LearnSy* does. We compare *LearnSy* with these heuristics in our evaluation, and 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 the role of the question selector is taken by either the oracle or the learner depending on the type of queries. For example, in CEGIS, the query asks for a counterexample for a candidate program, and thus the input is selected by the oracle; in interactive program synthesis, the query asks for the corresponding output for a given input, and thus 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, in this paper, we consider only queries in the form of input-output examples. Though it is indeed the most common form of queries, there have been multiple OGIS solvers 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 different kinds of 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* for constructing effective decision trees, which searches for the input best distinguishing between remaining programs. To efficiently apply this strategy in synthesis, this paper designs a compositional approximation for the objective function in this strategy.

Our 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. On the one hand, we prove that this model can provide an unbiased estimation for the overall equivalence in the program space. On the other hand, we prove that the precision of each estimation can be further improved by an operator namely flattening.

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

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. 2023. *Artifact for OOPSLA'23: Improving Oracle-Guided Inductive Synthesis by Efficient Question Selection.* https://github.com/jiry17/LearnSy

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