Generalizable Synthesis through Unification

RUYI JI, Peking University, China
JINGTAO XIA, Peking University, China
YINGFEI XIONG*, Peking University, China
ZHENJIANG HU, Peking University, China

The generalizability of PBE solvers is the key to the empirical synthesis performance. Despite the importance of generalizability, related studies on PBE solvers are still limited. In theory, few existing solvers provide theoretical guarantees on generalizability, and in practice, there is a lack of PBE solvers with satisfactory generalizability on important domains such as conditional linear integer arithmetic (CLIA). In this paper, we adopt a concept from the computational learning theory, Occam learning, and perform a comprehensive study on the framework of synthesis through unification (STUN), a state-of-the-art framework for synthesizing programs with nested if-then-else operators. We prove that Eusolver, a state-of-the-art STUN solver, does not satisfy the condition of Occam learning, and then we design a novel STUN solver, PolyGen, of which the generalizability is theoretically guaranteed by Occam learning. We evaluate PolyGen on the domains of CLIA and demonstrate that PolyGen significantly outperforms two state-of-the-art PBE solvers on CLIA, Eusolver and Euphony, on both generalizability and efficiency.

CCS Concepts: • Software and its engineering → Software notations and tools; General programming languages.

Additional Key Words and Phrases: Programming by Example, Synthesis through Unification, Occam Learning

ACM Reference Format:
Ruyi Ji, Jingtao Xia, Yingfei Xiong, and Zhenjiang Hu. 2021. Generalizable Synthesis through Unification. Proc. ACM Program. Lang. 5, OOPSLA, Article 167 (October 2021), 28 pages. https://doi.org/10.1145/3485544

1 INTRODUCTION

In the past decades, oracle-guided inductive program synthesis (OGIS) [Jha and Seshia 2017] receives much attention. In each iteration of OGIS, an oracle provides input-output examples to an inductive program synthesizer, or programming-by-example (PBE) synthesizer [Shaw et al. 1975], and the PBE synthesizer generates a program based on the examples. There are two typical types of OGIS problems. In the first type, the oracle can verify whether the synthesized program is correct, and provides a counter-example if the program is incorrect. Many applications under the counter-example guided inductive synthesis (CEGIS) framework [Solar-Lezama et al. 2006] fall into

*Corresponding author

Authors’ addresses: Ruyi Ji, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, EECS, Peking University, Beijing, China, jiruyi910387714@pku.edu.cn; Jingtao Xia, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, EECS, Peking University, Beijing, China, xiajt@pku.edu.cn; Yingfei Xiong, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, EECS, 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, EECS, Peking University, Beijing, China, huzj@pku.edu.cn.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2021 Copyright held by the owner/author(s).
2475-1421/2021/10-ART167
https://doi.org/10.1145/3485544
this type. In the second type, the oracle cannot verify the correctness of the synthesized program but can provide a set of input-output examples. This includes the applications where the oracle is a black-box program, such as binary programs [Zhai et al. 2016], and applications where the program is too complex to verify its correctness, e.g., the task involves system calls or complex loops, such as program repair, second-order execution, and deobfuscation [Blazytko et al. 2017; David et al. 2020; Jha et al. 2010; Mechttev et al. 2018, 2015a].

In both types of problems, the generalizability of the PBE solver is the key to synthesis performance. In the first type, generalizability significantly affects the efficiency: the fewer examples the solver needs to synthesize a correct program, the fewer CEGIS iterations the synthesis requires, and thus the faster the synthesis would be. In the second type, the generalizability of the PBE solver decides the correctness of the whole OGIS system.

Despite the importance of generalizability, the studies on the generalizability of the existing PBE solvers are still limited. On the theoretical side, as far as we are aware, no existing PBE solver provides theoretical guarantees on generalizability. On the practical side, the generalizability of the existing PBE solvers is not satisfactory. In our evaluation, on a synthesis task for solving the maximum segment sum problem, Eusolver [Alur et al. 2017], a state-of-the-art PBE solver, uses 393 examples to find the correct program, while our solver uses only 10.

In this paper, we propose a novel PBE solver, PolyGen, that provides a theoretical guarantee on generalizability by construction. We adopt a concept from the computational learning theory, Occam learning [Blumer et al. 1987], and prove that PolyGen is an Occam solver, i.e., a PBE solver that satisfies the condition of Occam learning. A PBE solver is an Occam solver if, for any possible target program consistent with the given examples, the size of the synthesized program is guaranteed to be polynomial to the target program and sub-linear to the number of provided examples with a high probability. In other words, an Occam solver would prefer smaller programs to larger programs and thus follows the principle of Occam’s Razor. In theory, Blumer et al. [1987] have proved that, given any expected accuracy, the number of examples needed by an Occam solver to guarantee the accuracy is bounded by a polynomial on the size of the target program. In practice, Occam learning has exhibited good generalizability in different domains [Aldous and Vazirani 1995; Angluin and Laird 1987; Kearns and Li 1988; Kearns and Schapire 1994; Natarajan 1993].

PolyGen follows the synthesis through unification (STUN) [Alur et al. 2015] framework. STUN is a framework for synthesizing nested if-then-else programs, and the solvers based on STUN such as Eusolver [Alur et al. 2017] and Euphony [Lee et al. 2018] achieve the state-of-the-art results on many important benchmarks, e.g., the CLIA track in the SyGuS competition. A typical STUN solver consists of a term solver and a unifier. First, the term solver synthesizes a set of if-terms, each being correct for a different subset of the input space, and then the unifier synthesizes if-conditions that combine the terms into a correct program for the whole input space.

We first analyze a state-of-the-art STUN solver, Eusolver [Alur et al. 2017], and prove that Eusolver is not an Occam solver. Then we proceed to design PolyGen. A key challenge to designing an Occam solver is to scale up while satisfying the condition of Occam learning. For example, a trivial approach to ensuring Occam learning is to enumerate programs from small to large and return the first program consistent with the examples. However, this approach only scales to small programs. To ensure scalability, we divide the synthesis task into subtasks, each synthesizing a subpart of the program, and propagate the condition of Occam learning into a sufficient set of conditions, where each condition is defined on a subtask. Roughly, these conditions require that each subtask synthesizes either a small program or a set of programs whose total size is small. Then, we find efficient synthesis algorithms that meet the respective conditions for each subtask.

We instantiate PolyGen on the domains of conditional linear integer arithmetic (CLIA), and evaluate PolyGen against Esolver [Alur et al. 2013], the best known PBE solver on CLIA that
always synthesizes the smallest valid program, *Eusolver* [Alur et al. 2017] and *Euphony* [Lee et al. 2018], two state-of-the-art PBE solvers on CLIA. Our evaluation is conducted on 100 benchmarks collected from the dataset of SyGuS-Comp [Alur et al. 2019] and an application of synthesizing divide-and-conquer algorithms [Farzan and Nicolet 2017]. Besides, our evaluation considers two major oracle models in OGIS, corresponding to the applications where (1) the oracle can provide a counter-example for a given program, and (2) the oracle can only generate the correct output for a given input. Our evaluation results show that:

- Comparing with *Esolver*, *PolyGen* achieves almost the same generalizability while solving 9.55 times more benchmarks than *Esolver*.
- Comparing with *Eusolver* and *Euphony*, on efficiency, *PolyGen* solves 43.08%-90.20% more benchmarks with ×6.14-×15.07 speed-ups. Besides, on generalizability, *Eusolver* and *Euphony* require ×1.12-×2.33 examples comparing with *PolyGen* on those jointly solved benchmarks. This ratio raises to at least ×2.27-×3.32 when the survivorship bias is considered.

To sum up, this paper makes the following contributions:

- We adopt the concept of Occam learning to the domain of PBE, prove that *Eusolver* is not an Occam solver, and provide a sufficient set of conditions for individual components in the STUN framework to form an Occam solver (Section 5).
- We design a novel Occam solver based on the STUN framework, *PolyGen*, by designing efficient algorithms for the two components that meet the above conditions. (Sections 6, 7).
- We instantiate *PolyGen* to the domain of CLIA (Section 8) and evaluate *PolyGen* against state-of-the-art PBE solvers on CLIA (Section 9). The evaluation results show that *PolyGen* significantly outperforms *Eusolver* and *Euphony* on both efficiency and generalizability.

## 2 RELATED WORK

### Generalizability of PBE Solvers

Generalizability is known to be important for PBE solvers, and there have been different approaches proposed to improve generalizability.

Guided by the principle of Occam’s Razor, a major line of previous work converts the PBE task into an optimization problem by requiring the solver to find the simplest program [Gulwani 2011; Liang et al. 2010; Mehtaev et al. 2015b; Raychev et al. 2016]. This method has been evaluated to be effective in different domains, such as user-interacting PBE systems [Gulwani 2011] and program repair [Mehtaev et al. 2015b]. However, the usage of this method is limited by efficiency, as in theory, requiring the optimality of the solution would greatly increase the difficulty of a problem. For many important domains, there is still a lack of an efficient enough PBE solver which implementing this method. For example, on the domains of CLIA, our evaluation shows that *Eusolver*, a state-of-the-art PBE solver, solves 6.22 times more benchmarks than *Esolver* [Alur et al. 2013], the known best PBE solver on CLIA that guarantees to return the simplest program.

Comparing with previous work, though our paper is also based on the principle of Occam’s Razor, we relax the constraint on the PBE solver by adopting the concept of Occam learning [Blumer et al. 1987] from computational learning theory. Occam learning allows the solver to return a program that is at most polynomially worse than the optimal and still has theoretical guarantees on generalizability. While designing an Occam solver, we have more space to improve the efficiency than designing a solver optimizing the size. In this way, we successfully implement a PBE solver on CLIA that performs well on both efficiency and generalizability.

Another line of work uses learned models to guide the synthesis procedure, and thus focuses on only probable programs [Balog et al. 2017; Chen et al. 2019; Devlin et al. 2017; Ji et al. 2020b; Kalyan et al. 2018; Lee et al. 2018; Menon et al. 2013; Singh and Gulwani 2015]. However, the efficiency of these approaches depends on domain knowledge. For example, Kalyan et al. [2018] use
input-output examples to predict the structure of the target program on the string manipulation domain: The effectiveness of their model relies on the structural information provided by strings and thus is unavailable on those unstructured domains, such as CLIA. In our evaluation, we evaluate a state-of-the-art PBE solver based on learned models, namely *Euphony* [Lee et al. 2018], and the result shows that its effectiveness is limited on CLIA.

Besides, Raza and Gulwani [2018] explores another synthesis model to improve the generalizability of the whole synthesis system. In this model, multiple programs are synthesized at the same time, and one among them will be selected after the query inputs are given. Such a model can be combined with existing approaches as well as ours. These approaches can be used at the first stage of this model to generate programs with better generalizability, and thus it will be more probable that a user-wanted program will be selected at the second stage. A part of the algorithm proposed by Raza and Gulwani [2018] shares some similarities with Algorithm 3 in our paper, because both algorithms are based on a classical approximation algorithm for set covering.

There also exist approaches that improve the generalizability by introducing user interactions as input to synthesizers [Ji et al. 2020a; Mayer et al. 2015; Padhi et al. 2018; Wang et al. 2017]. An interactive synthesizer selects inputs according to the ability to reduce the ambiguity and queries the user for the corresponding output. In this way, the quality of each example increases, and the number of required examples is reduced. These approaches are orthogonal to our paper and can be potentially used to further improve the generalizability of our approach, which will be future work.

**Analysis on the generalizability.** Analyzing the generalizability of learning algorithms is an important problem in machine learning and has been studied for decades. The *probably approximately correct (PAC) learnability* [Valiant 1984] is a widely used framework for analyzing generalizability. When discussing the PAC learnability of a learning task, the goal is to find a learning algorithm that (1) runs in polynomial time, (2) requires only a polynomial number of examples to achieve any requirement on the accuracy. On the synthesis side, there has been a line of previous work on the PAC learnability of logic programs [Cohen 1995a,b; Dzeroski et al. 1992]. Besides, some approaches use the framework of PAC learnability to analyze the number of examples required by some specific algorithm [Drews et al. 2019; Lau et al. 2003].

In this paper, we seek a theoretical model that can compare the generalizability of different PBE solvers. At this time, the requirement on the generalizability provided by PAC learnability is too loose: According to the general bound provided by Blumer et al. [1987], when the program space is finite, this condition is satisfied by any valid PBE solver. Therefore, we adopt another concept, Occam learning, from computational learning theory. Comparing with PAC learnability, Occam learning (1) has a higher requirement on generalizability, as shown by Blumer et al. [1987], and (2) can reflect some empirical results in program synthesis, such as a PBE solver that always returns the simplest program should have better generalizability than an arbitrary PBE solver. To our knowledge, we are the first to introduce Occam learning into program synthesis.

**Synthesizing CLIA Programs.** As mentioned in the introduction, our approach is implemented on the domains of CLIA. CLIA is important for program synthesis, as CLIA programs widely exist in real-world projects and can express complex behaviors by using nested if-then-else operators. There have been many applications of CLIA synthesizers, such as program repair [Le et al. 2017; Mechtaev et al. 2015b], automated parallelization [Farzan and Nicolet 2017; Morita et al. 2007]. On CLIA, PBE solvers are usually built on the STUN framework [Alur et al. 2015], which firstly synthesizes a set of if-terms by a term solver, and then unifies them into a program by a unifier. There are two state-of-the-art STUN solvers:

- *Eusolver* [Alur et al. 2017], which comprises an enumerative term solver and a unifier based on a decision-tree learning algorithm.
Euphony [Lee et al. 2018], which uses structural probability to guide the synthesis of Eusolver. PolyGen also follows the STUN framework. We evaluate PolyGen against these two solvers in Section 9. The result shows that PolyGen outperforms them on both efficiency and generalizability. Outside PBE, there are other techniques proposed for synthesizing CLIA programs:

- DryadSynth [Huang et al. 2020] reconciles enumerative and deductive synthesis techniques. As DryadSynth requires a logic specification, it is not suitable for PBE tasks.
- CVC4 [Reynolds et al. 2019] synthesizes programs from unsatisfiability proofs given by theory solvers. Though CVC4 is runnable on PBE tasks, it seldom generalizes from examples. We test CVC4 on a simple task where the target is to synthesize a program that returns the maximal value among three inputs. After requiring 300 random examples, the error rate of the program synthesized by CVC4 on a random input is still larger than 97%. In contrast, PolyGen requires only 12 examples on average to synthesize a completely correct program.

There are also approaches on synthesizing boolean conditions, which is an important part in CLIA [Ernst et al. 2001; Padhi and Millstein 2017]. However, none of them discuss the theoretical guarantees on the generalizability, and it is unknown whether these approaches are Occam solvers.

3 MOTIVATING EXAMPLE AND APPROACH OVERVIEW

In this section, we introduce the basic idea of our approach via a motivating example adopted from benchmark mpg ITE2.s1 in the SyGuS competition. The target program $p^*$ is shown as the following, where $x, y, z$ are three integer inputs.

$$p^*(x, y, z) := \begin{cases} 
\text{if } (x + y \geq 1) \text{ then } \{ \\
\text{if } (x + z \geq 1) \text{ then } \{x + 1\} \text{ else } \{y + 1\} \\
\text{else } \{ \\
\text{if } (y + z \geq 1) \text{ then } \{z + 1\} \text{ else } \{y + 1\} 
\} \}
\end{cases}$$

We assume that there are 9 input-output examples provided to the PBE solver. Table 1 lists these examples, where tuple $(x_0, y_0, z_0)$ in column $I$ represents an input where $x, y, z$ are set to $x_0, y_0, z_0$ respectively, and the if-term in column Term represents the executed branch on each example.

| ID | $I$  | $O$  | Term | ID | $I$  | $O$  | Term |
|----|------|------|------|----|------|------|------|
| $e_1$ | (0, 1, 2) | 1 | $x + 1$ | $e_4$ | (0, 2, 0) | 3 | $y + 1$ | $e_7$ | (0, 0, 1) | 2 |
| $e_2$ | (1, 0, 2) | 2 |          | $e_5$ | (−1, 3, 0) | 4 |          | $e_8$ | (−3, 3, −2) | −1 |
| $e_3$ | (−1, 3, 2) | 0 |          | $e_6$ | (−1, 1, −1) | 2 |          | $e_9$ | (−1, 0, 4) | 5 |

3.1 Eusolver

In this paper, we focus on designing an Occam solver for PBE tasks. As mentioned before, an Occam solver should synthesize programs whose size is polynomial to the size of the target program and sub-linear to the number of provided examples with high probability. Since the target program is unknown, an Occam solver should satisfy this requirement when the target program is the smallest valid program (programs consistent with the given input-output examples), and thus prefer smaller programs to larger programs.

We first show that Eusolver [Alur et al. 2017] may return unnecessarily large programs and thus is unlikely to be an Occam solver. We shall formally prove that Eusolver is not an Occam solver in...
Section 5.3. *Eusolver* follows the STUN framework and provides a term solver and a unifier. The term solver is responsible for synthesizing a term set that jointly covers all the given examples. In our example, one valid term set is \{x + 1, y + 1, z + 1\}. A unifier is responsible for synthesizing a set of conditions to unify the term set into a program with nested if–then–else operators. In our example, the conditions used in program \(p^*\) are \{(x + y \geq 1) \land (x + z \geq 1), \land (y + z \leq 1)\}. A unifier is responsible for synthesizing a set of conditions to unify the term set into a program with nested if–then–else operators. In our example, the conditions used in program \(p^*\) are \{(x + y \geq 1) \land (x + z \geq 1), \land (y + z \leq 1)\}. A unifier is responsible for synthesizing a set of conditions to unify the term set into a program with nested if–then–else operators. In our example, the conditions used in program \(p^*\) are \{(x + y \geq 1) \land (x + z \geq 1), \land (y + z \leq 1)\}.

Similar to Occam learning, *Eusolver* also tries to return small programs. To achieve this, *Eusolver* enumerates the terms and the conditions from small to large and then tries to combine the enumerated terms and conditions into a complete program. Though this approach controls the sizes of individual terms and conditions, it fails to control the number of the terms and the conditions, and thus may lead to unnecessarily large programs.

The term solver of *Eusolver* enumerates the terms from small to large and includes a term in the term set when the set of examples covered by this term (i.e., the term is correct on these examples) is different from all smaller terms. The term set is complete when all examples are covered. As a result, this strategy may unnecessarily include many small terms, each covering only a few examples. In our motivating example, if constants such as \(-1, \ldots, 5\) are available, \(T_E\) will return set \{-1, \ldots, 5\} instead of \{x + 1, y + 1, z + 1\}. Though such terms are small, the number of the terms grows with the number of examples.

The unifier of *Eusolver* builds a decision tree using the ID3 algorithms. Here the terms are considered as labels, the enumerated if–conditions are considered as conditions, and a term is a valid label for an example if it covers the example. However, ID3 is designed for fuzzy classification problems, uses information gain to select conditions, and may select conditions that negatively contribute to program synthesis. For example, \(x \geq 0\) will have good information gain for this example. In the original sets the three labels \(x + 1, y + 1, z + 1\) are evenly distributed. Predicate \(x \geq 0\) divides the examples into two sets where the distribution becomes uneven: in one set 50% of the examples are labeled with \(x + 1\), and in the other set only 20% of the examples are labeled \(x + 1\). However, in both sets, the three labels still exist, and we still have to find conditions to distinguish them. As a result, selecting \(x > 0\) roughly doubles the size of the synthesized program.

3.2 PolyGen

To synthesize small programs, *PolyGen* controls not only the size of individual terms and conditions but also the number of conditions and terms. The structure of *PolyGen* is shown as Figure 1(a). As
we can see in the figure, PolyGen is built of a set of sub-solvers, each responsible for synthesizing part of the program. Figure 1(b) lists a program PolyGen synthesizes, and the colored rectangles show the parts of the program synthesized by the sub-solver with the same color. In the following, we shall illustrate how PolyGen works.

3.2.1 Term Solver. To control the number of terms, the term solver of PolyGen iterates a threshold on the number of terms and tries to synthesize a term set whose size is equal to or smaller than this threshold. The threshold starts with a small number and increases by a constant \( c \) in each iteration. The process terminates if any iteration successfully synthesizes a term set. In each iteration, PolyGen uses a randomized procedure to synthesize a term set. If a term set exists under a threshold, the probability that the term solver fails to synthesize a term set is bound by a constant \( \epsilon \). Let us assume that a term set exists within the first iterated threshold. After \( n \) iterations, the probability of failing to synthesize reduces to \( \epsilon^n \), and the possible number of terms is only increased by \( nc \). In other words, the number of synthesized terms is guaranteed to be small with a high probability.

Now we explain how we implement the randomized procedure to obtain a term set with a bounded failure probability. We assume there is a domain solver that synthesizes a term based on a set of examples, and the domain solver is also an Occam solver. For illustration, let us assume currently the threshold for the number of terms is 3, and in our example there exists at least one term set \( T = \{ x + 1, y + 1, z + 1 \} \) under this threshold. The term solver first samples many subsets of the examples and invokes the domain solver to synthesize a term for each subset. If any subset is covered by a term in \( T \), the domain solver will have a chance to synthesize the term. For example, if \( e_1 \) and \( e_2 \) are sampled, the domain solver has a chance to synthesize \( x + 1 \) because of the generalizability of the Occam solver. As a result, if we sample enough subsets, we can synthesize a term in \( T \) with any small bounded failure rate. Then for any successfully synthesized subsets, we repeat this procedure to synthesize terms for the remaining examples. For example, when \( x + 1 \) is synthesized, the procedure continues with examples \( e_4 \ldots e_9 \). The procedure ends when no example remains. Since in each turn the probability of failing to find a term in \( T \) is bounded, the total probability of failing to find the term set \( T \) is bounded. Please note the sizes of synthesized terms are guaranteed to be small as the domain solver is an Occam solver.

In the domain of CLIA, the if-terms are all linear integer expressions, and the domain solver can be implemented by finding the simplest valid term via linear integer programming, as we shall show in Section 8.

3.2.2 Unifier. To control the number of conditions, instead of synthesizing a decision tree, the unifier of PolyGen synthesizes a decision list [Rivest 1987]. In a decision list, each condition distinguishes one term from the rest of the terms. Figure 1(b) shows the program synthesized by PolyGen on examples \( \{ e_1, \ldots, e_9 \} \), which is semantically equivalent to the target program \( p^* \). The number of conditions is equal to the number of terms minus one and thus is bounded.

However, the conditions in a decision list may become larger and thus cannot be synthesized using an enumerative algorithm. To efficiently synthesize small conditions to distinguish the terms, we notice that the conditions are in the disjunctive normal forms (DNF) in the initial definition of decision lists, where a DNF is the disjunction of clauses, a clause is the conjunction of literals, and a literal is a predicate in the grammar or its negation. Then we design three sub-solvers for different parts of the conditions. The clause solver synthesizes clauses from the literals, where the literals are enumerated by size in the same way as Eusolver. The DNF solver synthesizes a DNF formula based on the clause solver. Finally, the unifier synthesizes all the conditions based on the DNF solver.

Given a set of terms, the unifier create a synthesis subtask for each term \( t \), where the synthesized program has to return \( \text{true} \) on example inputs that are covered \( \text{only} \) by \( t \) (positive examples) and return \( \text{false} \) on the example inputs that are not covered by \( t \) (negative examples). For example,
when synthesizing the condition for $y + 1$, $e_4, e_5, e_6$ are positive examples and $e_7, e_8, e_9$ are negative examples. Here $e_1, e_2, e_3$ are already covered by $x + 1$. Then the unifier invokes the DNF solver to solve these tasks. The conditions synthesized are guaranteed to be small if the DNF solver guarantees to return small conditions.

Before getting into the DNF solver, let us discuss the clause solver first. Given a set of literals, a set of input-output examples where the output is Boolean, the clause solver returns a clause, i.e., the conjunction of a subset of literals satisfying all examples. The clause solver reduces this problem into weighted set covering and uses a standard approximation algorithm [Chvátal 1979] to solve it. As will be formally proved later, the clause solver is an Occam solver.

Based on the clause solver, we build the DNF solver. The DNF solver synthesizes a set of clauses, where all clauses should return false for each negative example, and at least one clause should return true for each positive example. We notice this synthesis problem has the same form as the term solver: given a set of examples (in this case, a set of positive examples) and an Occam solver (in this case, the clause solver), we need to synthesize a set of programs (in this case, a set of clauses) to cover these examples. Therefore, the DNF solver uses the same algorithm as the term solver, and we uniformly refer to this algorithm as disjunctive synthesis. Since the disjunctive synthesis algorithm guarantees that the returned program set is small in terms of both the sizes of individual programs and the number of returned programs, the DNF solver guarantees to return a condition of small size.

4 OCCAM LEARNING

4.1 Preliminaries: Programming by Example

The problem of programming-by-example (PBE) is usually discussed above an underlying domain $\mathcal{D} = (\mathcal{P}, \mathcal{I}, \mathcal{O}, \mathcal{⟦} \, \mathcal{⟧}_\mathcal{D})$, where $\mathcal{P}, \mathcal{I}, \mathcal{O}$ represent a program space, an input space, and an output space respectively, $\mathcal{⟦} \, \mathcal{⟧}_\mathcal{D}$ is an oracle function that associates a program $p \in \mathcal{P}$ and an input $I \in \mathcal{I}$ with an output in $\mathcal{O}$, denoted as $\mathcal{⟦}p\mathcal{⟧}_\mathcal{D}(I)$. The domain limits the ranges of possibly used programs and concerned inputs and provides the semantics of these programs.

For simplicity, we make two assumptions on the domain: (1) There is a universal oracle function $\mathcal{⟦} \, \mathcal{⟧}$ for all domains; (2) The output space $\mathcal{O}$ is always induced by the program space $\mathcal{P}$, the input space $\mathcal{I}$, and the oracle function $\mathcal{⟦} \, \mathcal{⟧}$, i.e., $\mathcal{O} = \{\mathcal{⟦}p\mathcal{⟧}(I) \mid p \in \mathcal{P}, I \in \mathcal{I}\}$. In the remainder of this paper, we abbreviate a domain as a pair $(\mathcal{P}, \mathcal{I})$ of a program space and an input space. We shall use notation $\mathcal{T}$ to represent a family of domains, and thus discuss general properties of domains.

PBE [Shaw et al. 1975] is a subproblem of program synthesis where the solver is required to learn a program from a set of given input-output examples. As this paper focuses on the generalizability of PBE solvers, we assume that there is at least one program satisfying all given examples: The problem of determining whether there is a valid program is another domain of program synthesis, namely unrealizability [Hu et al. 2020; Kim et al. 2021], and is out of the scope of our paper.

Definition 4.1 (Programming by Example). Given a domain $\mathcal{D}$, a PBE task $T \in (\mathcal{I} \times \mathcal{O})^*$ is a sequence of input-output examples. Define $\mathcal{T}(\mathcal{D}) \subseteq (\mathcal{I} \times \mathcal{O})^*$ as the set of PBE tasks where there is at least one program satisfying all examples. PBE solver $\mathcal{S}$ is a function that takes a PBE task as the input, and returns a program satisfying all given examples, i.e., $\forall T \in \mathcal{T}(\mathcal{D}), \forall (I, O) \in T, \mathcal{⟦}S(T)\mathcal{⟧}(I) = O$.

4.2 Occam Learning and Occam Solver

In computational learning theory, Occam learning [Blumer et al. 1987] is proposed to explain the effectiveness of the principle of Occam’s Razor. In PBE, an Occam solver guarantees that the size of the synthesized program is at most polynomially larger than the size of the target program.
Definition 4.2 (Occam Solver\(^1\)). For constants \(\alpha \geq 1, 0 \leq \beta < 1\), PBE solver \(S\) is an \((\alpha, \beta)\)-Occam solver on a family of domains \(\mathcal{F}\) if there exist constants \(c, \gamma > 0\) such that for any program domain \(D \in \mathcal{F}\), for any target program \(p^* \in \mathbb{P}\), for any input set \(\{I_1, \ldots, I_n\} \subseteq \mathbb{I}\), for any error rate \(\varepsilon \in (0, \frac{1}{2})\):

\[
\Pr \left[ \text{size} \left( S(T(p^*, I_1, \ldots, I_n)) \right) > c (\text{size}(p^*))^{\alpha} n^\beta \ln^\gamma \left( \frac{1}{\varepsilon} \right) \right] \leq \varepsilon
\]

where \(\text{size}(p)\) is the length of the binary representation of program \(p\), \(T(p^*, I_1, \ldots, I_n)\) is defined as the PBE task corresponding to target program \(p^*\) and inputs \(I_1, \ldots, I_n\).

We assume the program domain is defined by a context-free grammar. At this time, a program is

\[\text{size} \left( \mathcal{L} \right) \geq 1\]

two primitive solvers \(S\) and \(T\), \(T\) is a more general solver than \(S\).

Definition 4.3. The size \(\text{size}(p)\) of program \(p\) is defined as \([\log_2 N] \times |p|\), where \(|p|\) is the number of grammar rules used to derive \(p\), and \(N\) is the number of different grammar rules.

Example 4.4. When only input variable \(x\), operator + and constants 1, 2 are available in the grammar, \(\text{size}(x + 1)\) is defined as \([\log_2 4] \times 3 = 6\). When there are \(a\) input variables, \(b\) constants and \(c\) different operators available, \(\text{size}(x + 1)\) is defined as \(3[\log_2 (a + b + c)]\).

The size provides a logarithmic upper bound on the number of programs no larger than \(p\).

Lemma 4.5. For any domain \(D\), \(\forall p \in \mathbb{P}\), \(|\{p' \in \mathbb{P} \mid \text{size}(p') \leq \text{size}(p)\}| \leq 2^{\text{size}(p)}\).

Blumer et al. [1987] analyze Occam solvers under the probably approximately correct (PAC) learnability framework and proves that the generalizability of Occam solvers is always guaranteed.

Theorem 4.6. Let \(S\) be an \((\alpha, \beta)\)-Occam solver on domain \(D\). Then there exist constants \(c, \gamma > 0\) such that for any \(0 < \varepsilon, \delta < 1\), for any distribution \(D\) over \(\mathbb{I}\) and any target program \(p^* \in \mathbb{P}\):

\[
\forall n > c \left( \frac{1}{\varepsilon} \ln \left( \frac{2}{\delta} \right) + \frac{(\text{size}(p^*))^{\alpha} \ln^\gamma (2/\delta)}{\varepsilon} \right)^{1/(1-\beta)} \right), \Pr_{I_1 \sim D, \ldots, I_n \sim D} \left[ \text{err}_{D, p^*} \left( S(T(p^*, I_1, \ldots, I_n)) \right) \geq \varepsilon \right] \leq \delta
\]

where \(\text{err}_{D, p^*}(p)\) represents the error rate of program \(p\) when the input distribution is \(D\) and the target program is \(p^*\), i.e., \(\text{err}_{D, p^*}(p) := \Pr_{I \sim D} [\|p\|_I \neq \|p^*\|_I]\).

Due to the space limit, we move all proofs to our appendix [Ji et al. 2021].

When \(\varepsilon\) and \(\delta\) are fixed, Theorem 4.6 implies that an \((\alpha, \beta)\)-Occam solver can find a program similar to the target \(p^*\) with only \(O \left( \text{size}(p^*)^{\alpha/(1-\beta)} \right)\) examples. Such a bound matches the principle of Occam’s Razor, as it increases monotonically when the size of the target program increases.

The class of Occam solvers can reflect the practical generalizability of PBE solvers. Let us take two primitive solvers \(S_{\text{min}}\) and \(S_{\text{rand}}\) as an example. For any PBE task \(T\), let \(\mathbb{P}(T) \subseteq \mathbb{P}\) be the set of programs that are consistent with examples in \(T\).

- \(S_{\text{rand}}\) is the most trivial synthesizer that has no guarantee on the quality of the result. It just uniformly returns a program from \(\mathbb{P}(T)\): \(\forall p \in \mathbb{P}(T), \Pr [S_{\text{rand}}(T) = p] = |\mathbb{P}(T)|^{-1}\).
- \(S_{\text{min}}\) regards a PBE task as an optimization problem, and always returns the syntactically smallest program in \(\mathbb{P}(T)\): \(S_{\text{min}}(T) := \arg \min_{p \in \mathbb{P}(T)} \text{size}(p)\).

In practice, it is usually believed that \(S_{\text{min}}\) has better generalizability than \(S_{\text{rand}}\). We prove that the class of Occam solvers can discover this advantage, as \(S_{\text{min}}\) is an Occam solver but \(S_{\text{rand}}\) is not.

Theorem 4.7. Let \(\mathcal{F}_A\) be the family of all possible domains. Then \(S_{\text{min}}\) is an \((1, 0)\)-Occam solver on \(\mathcal{F}_A\), and \(S_{\text{rand}}\) is not an Occam solver on \(\mathcal{F}_A\).

\(^1\)The original definition of Occam solvers is only for deterministic algorithms. Here we extend its definition to random algorithms. We compare these two definitions in the appendix [Ji et al. 2021].
5 SYNTHESIS THROUGH UNIFICATION

5.1 Preliminaries: Synthesis through Unification

The framework of synthesis through unification (STUN) focuses on synthesizing programs with nested if-then-else operators. Formally, STUN assumes that the program space can be decomposed into two subspaces, for if-conditions and if-terms respectively.

Definition 5.1. A program space $P$ is a conditional program space if and only if there exist two program spaces $P_t$ and $P_c$ such that $P$ is the smallest set of programs such that:

$$P = P_t \cup \{ \text{if } c \text{ then } p_1 \text{ else } p_2 \mid p_1, p_2 \in P, c \in P_c \}$$

We use pair $(P_t, P_c)$ to denote a conditional program space derived from term space $P_t$ and condition space $P_c$. Besides, we call a domain $D$ conditional if the program space in $D$ is conditional.

A STUN solver synthesizes programs in two steps:

1. A term solver is invoked to synthesize a set of programs $P \subseteq P_t$ such that for any input, there is always a consistent program in $P$.
2. A unifier is invoked to synthesize a valid program from conditional program space $(P, P_c)$.

In this paper, we only consider the specialized version of the STUN framework on PBE tasks.

Definition 5.2 (Term Solver). Given conditional domain $D$, term solver $T : T(D) \rightarrow P(P_t)$ returns a set of terms covering all examples in a given PBE task, where $P(P_t)$ denotes the power set of $P_t$:

$$\forall T \in T(D), \forall (I, O) \in T, \exists p \in T(T), \| p \| (I) = O$$

Definition 5.3 (Unifier). Given a conditional domain $D$, a unifier $U$ is a function such that for any set of terms $P \subseteq P_t$, $U(P)$ is a valid PBE solver for $(P, P_c)$.

A STUN solver consists of a term solver $T$ and a unifier $U$. Given a PBE task $T$, the solver returns $U(T(T))(T)$ as the synthesis result. Alur et al. [2015] prove that such a combination is complete when the conditional domain is if-closed. For other domains, STUN can be extended to be complete by backtracking to the term solver when the unifier fails [Alur et al. 2015, 2017].

Definition 5.4 (If-Closed). A conditional domain $D$ is if-closed if:

$$\forall p_1, p_2 \in P_t, \exists c \in P_c, \forall I \in I, (\| c \| (I) \iff \| p_1 \| (I) = \| p_2 \| (I))$$

Please note that any conditional domain with equality is if-closed, as $c$ can be constructed by testing the equality between the outputs of $p_1$ and $p_2$. In the rest of the paper, we assume the conditional program space is if-closed, and use $F_C$ to denote a family of if-closed domains.

Eusolver [Alur et al. 2017] is a state-of-the-art solver following the STUN framework. It takes efficiency and generalizability as its design purposes and makes a trade-off between them.

The term solver $F_E$ in Eusolver is motivated by $S_{\text{min}}$. $F_E$ enumerates terms in $P_t$ in the increasing order of the size. For each term $t$, if no smaller term covers the same set of examples as $t$ does, $t$ will be included in the result. $F_E$ returns when the result is enough to cover all examples.

The unifier $U_E$ in Eusolver regards nested if-then-else operators as a decision tree, and uses ID3 [Quinlan 1986], a standard decision-tree learning algorithm, to unify the terms. $U_E$ learns a decision tree recursively: In each recursion, it first tries to use a term to cover all remaining examples. If there is no such term, $U_E$ will heuristically pick up a condition $c$ from $P_c$ as the if-condition. According to the semantics of $c$, the examples will be divided into two parts, which will be used to synthesize the then-branch and the else-branch respectively.
5.2 Generalizability of STUN

In this section, we study the generalizability of the STUN framework. To start, we extend the concept of Occam solvers to term solvers and unifiers: For an Occam term solver, there should be a polynomial bound on the total size of synthesized terms, and for an Occam unifier, the induced PBE solver should always be an Occam solver for any possible term set. These definitions will be used to guide our design of PolyGen later.

Definition 5.5. For constants $\alpha \geq 1, 0 \leq \beta < 1$, term solver $T$ is an $(\alpha, \beta)$-Occam term solver on $\mathcal{T}_C$ if there exist constants $c, \gamma > 0$ such that for any domain $\mathcal{D} \in \mathcal{T}_C$, for any target program $p^* \in \mathbb{P}$, for any input set $\{I_1, \ldots, I_n\} \subseteq \mathbb{I}$, for any error rate $\epsilon \in (0, \frac{1}{2})$:

$$\text{Pr} \left[ \text{tsize} \left( T(p^*, I_1, \ldots, I_n) \right) > c \left( \text{size}(p^*) \right)^\alpha n^\beta \ln^\gamma \left( \frac{1}{\epsilon} \right) \right] \leq \epsilon$$

where $\text{tsize}(P)$ is the total size of terms in term set $P$, i.e., $\sum_{t \in P} \text{size}(t)$.

Definition 5.6. For constants $\alpha \geq 1, 0 \leq \beta < 1$, unifier $U$ is an $(\alpha, \beta)$-Occam unifier on $\mathcal{T}_C$ if there exist constants $c, \gamma > 0$ such that for any domain $\mathcal{D} \in \mathcal{T}_C$, for any term set $P \subseteq \mathcal{P}_t$, for any target program $p^* \in (P, \mathcal{P}_t)$, for any input set $I = \{I_1, \ldots, I_n\} \subseteq \mathbb{I}$, for any error rate $\epsilon \in (0, \frac{1}{2})$:

$$\text{Pr} \left[ \text{size} \left( U(P) \left( T(p^*, I_1, \ldots, I_n) \right) \right) > c \left( \text{max}(\text{size}(p^*), \text{tsize}(P)) \right)^\alpha n^\beta \ln^\gamma \left( \frac{1}{\epsilon} \right) \right] \leq \epsilon$$

In Definition 5.6, besides the size of the target program, the bound also refers to the total size of $P$. Such relaxation allows the unifier to use more examples when a large term set is provided.

Based on the above definitions, we prove that under some conditions, a STUN solver comprised of an Occam term solver and an Occam unifier is also an Occam solver.

Theorem 5.7. Let $T$ be an $(\alpha_1, \beta_1)$-Occam term solver on $\mathcal{T}_C$, $U$ be an $(\alpha_2, \beta_2)$-Occam unifier on $\mathcal{T}_C$ where $\beta_1 \alpha_2 + \beta_2 < 1$. Then the STUN solver comprised of $T$ and $U$ is an $((\alpha_1 + 1) \alpha_2, \beta_1 \alpha_2 + \beta_2)$-Occam solver on $\mathcal{T}_C$.

5.3 Generalizability of Eusolver

In this section, we analyze the generalizability of Eusolver and prove that Eusolver is not an Occam solver. We start from the term solver $\mathcal{T}_E$. As $\mathcal{T}_E$ enumerates terms in the increasing order of the size, $\mathcal{T}_E$ guarantees that all synthesized terms are small. However, the main problem of $\mathcal{T}_E$ is that it does not control the total number of synthesized terms. Therefore, the total size of the term set returned by $\mathcal{T}_E$ can be extremely large, as shown in Example 5.8.

Example 5.8. Consider the following term space $\mathcal{P}^n_t$, input space $\mathcal{I}^n_t$ and target program $p$:

$$\mathcal{P}^n_t = \{2, 3, \ldots, n+1, x+1\} \quad \mathcal{I}^n_t = [1, n] \cap \mathbb{Z} \quad p = x + 1$$

As $p$ is the largest term in $\mathcal{P}^n_t$, on any input $x_0$ in $\mathcal{I}^n_t$, there is always a smaller term $c$ that performs the same as $p$, where $c$ is a constant equal to $x_0 + 1$. Therefore, whatever the PBE task is, $\mathcal{T}_E$ always returns a subset of $P_A = \{2, 3, \ldots, n+1\}$ and never enumerates to the target program $p$.

When all inputs in $\mathcal{I}^n_t$ are included in the PBE task, the term set synthesized by $\mathcal{T}_E$ is always $P_A$. At this time, the total size of $P_A$ is $n \lceil \log_2(n + 3) \rceil$, the number of examples is $n$ and the size of the target program is $\lceil \log_2(n + 3) \rceil$. Clearly, there are no $\alpha \geq 1, 0 < \beta < 1$ and $c > 0$ such that $\forall n, t\text{size}(P_A) \leq c \left( \text{size}(p) \right)^\alpha n^\beta$. Therefore, $\mathcal{T}_E$ is not an Occam term solver.

Moreover, at this time, the program synthesized by Eusolver must utilize all terms in $P_A$, and thus its size is no smaller than $t\text{size}(P_A)$. So Eusolver is not an Occam solver as well.

The following fact comes from Example 5.8 immediately.
Theorem 5.9. \( \mathcal{T} \) is not an Occam term solver on \( \mathcal{F}^A \), and Eusolver is not an Occam solver on \( \mathcal{F}^A \), where \( \mathcal{F}^A \) is the family of all if-closed conditional domains.

The generalizability of \( \mathcal{U} \) is related to the underlying decision-tree learning algorithm ID3. Hancock et al. [1995] prove that there is no polynomial-time algorithm for learning decision trees that generalizes within a polynomial number of examples unless \( \text{NP} = \text{RP} \), where \( \text{RP} \) represents the class of polynomial-time random algorithms with one-side error. Combining with Theorem 4.6, we obtain the following lemma, which implies that \( \mathcal{U} \) is unlikely to be an Occam unifier.

Theorem 5.10. There is no polynomial-time Occam unifier on \( \mathcal{F}^A \) unless \( \text{NP} = \text{RP} \).

These results indicate that (1) Eusolver itself is not an Occam solver, and (2) if we would like to design an Occam solver following Theorem 5.7, neither \( \mathcal{T} \) nor \( \mathcal{U} \) can be reused.

6 TERM SOLVER

6.1 Overview

For an Occam term solver, the total size of returned terms should be bounded. Therefore, a term solver must be an Occam solver if (1) the number of returned terms is bounded, and (2) the maximal size of returned terms is bounded.

Lemma 6.1. For constants \( \alpha_1, \alpha_2 \geq 0, 0 \leq \beta_1, \beta_2 < 1 \) where \( \beta_1 + \beta_2 < 1 \), term solver \( \mathcal{T} \) is an \((\alpha_1 + \alpha_2, \beta_1 + \beta_2)\)-Occam solver on \( \mathcal{F}_C \) if there exist constants \( c, \gamma > 0 \) such that for any conditional domain \( \mathcal{D} \in \mathcal{F}_C \), any target program \( p^* \in \mathcal{P} \), and any input set \( \{I_1, \ldots, I_n\} \subseteq \mathcal{I} \):

1. With a high probability, the size of terms returned by \( \mathcal{T} \) is bounded by \( \text{size}(p^*)^{\alpha_1 n^{\beta_1}} \).

\[
\Pr \left[ \max \left\{ \text{size}(p) \mid p \in \mathcal{T}(I_1, \|p\| (I_1)), \ldots, (I_n, \|p\| (I_n)) \right\} > c (\text{size}(p^*))^{\alpha_1} n^{\beta_1} \ln^{\gamma} \left( \frac{1}{\epsilon} \right) \right] \leq \epsilon
\]

2. With a high probability, the number of terms returned by \( \mathcal{T} \) is bounded by \( \text{size}(p^*)^{\alpha_2 n^{\beta_2}} \).

\[
\Pr \left[ \left| \mathcal{T}(I_1, \|p\| (I_1)), \ldots, (I_n, \|p\| (I_n)) \right| > c (\text{size}(p^*))^{\alpha_2} n^{\beta_2} \ln^{\gamma} \left( \frac{1}{\epsilon} \right) \right] \leq \epsilon
\]

In Lemma 6.1, the first condition has a form similar to the guarantee provided by an Occam solver. Motivated by this point, we design \( \mathcal{T}_{\text{poly}} \) as a meta-solver that takes an Occam solver \( \mathcal{S}_T \) on the term space as an input. To solve a term finding task, \( \mathcal{T}_{\text{poly}} \) firstly decomposes it into several standard PBE tasks and then invokes \( \mathcal{S}_T \) to synthesize terms with bounded sizes.

One challenge is that in a term finding task, different examples correspond to different target terms, i.e., if-terms used in the target program. To find a target term using \( \mathcal{S}_T \), \( \mathcal{T} \) should pick up enough examples that correspond to the same target term. To do so, we utilize the fact that there must be a target term that covers a considerable portion of all examples, as shown in Lemma 6.2.

Lemma 6.2. Let \( T \) be a PBE task, and let \( P \) be a set of terms that covers all examples in \( T \), i.e., \( \forall (I, O) \in T, \exists p \in P, (\|p\| (I) = O) \). There is always a term \( p \in P \) such that:

\[
\left| \{(I, O) \in T \mid \|p\| (I) = O\} \right| \geq |T| - |P|
\]

Given a term finding task \( T \) where \( P^* \) is the set of target terms, let \( t^* \in P^* \) be the term that covers the most examples. According to Lemma 6.2, if we randomly select \( n_T \) examples from \( T \), term \( t^* \) will be consistent with all selected examples with a probability of at least \( |P^*|^{-n_T} \). Therefore, \( \mathcal{T}_{\text{poly}} \) repeatedly invokes \( \mathcal{S}_T \) on a small set of random examples drawn from \( T \): When the generalizability of \( \mathcal{S}_T \) is guaranteed on term space \( \mathcal{P}_T \), the number of examples and the number of turns are large enough, \( \mathcal{T}_{\text{poly}} \) will find a term semantically similar with \( t^* \) with a high probability.
Algorithm 1: The term solver $T_{\text{poly}}$ in PolyGen.

Input: A PBE task $T$, an $(\alpha, \beta)$-Occam sovler $S_T$ on the term space $P_T$ and a constant $c$.

Output: A set of programs $P$ that covers all examples.

Function GetCandidatePrograms$(\text{examples}, k, n_t, s)$:
1: $\text{result} \leftarrow \emptyset$;
2: for each $\text{turn} \in [1, n_T \times k^n]$ do
3: 
4: Uniformly and independently samples $n_T$ examples $e_1, \ldots, e_{n_T}$ from $\text{examples}$;
5: $p \leftarrow S_T(e_1, \ldots, e_{n_T})$;
6: if $p \neq \bot \land |\text{Covered}(p, \text{examples})| \geq |\text{examples}|/k \land \text{size}(p) \leq cs^{\alpha}n^\beta$ then
7: $\text{result} \leftarrow \text{result} \cup \{p\}$;
8: end
9: end
10: return $\text{result}$;

Function Search$(\text{examples}, k, n_t, s)$:
11: if $|\text{examples}| = 0$ then return $\emptyset$;
12: if $(\text{examples}, k)$ is visited before or $k = 0$ then return $\bot$;
13: for each $p \in \text{GetCandidatePrograms}(\text{examples}, k, n_t, s)$ do
14: $\text{searchResult} \leftarrow \text{Search}(\text{examples} - \text{Covered}(p, \text{examples}), k - 1, n_t, s)$;
15: if $\text{searchResult} \neq \bot$ then return $\{p\} \cup \text{searchResult}$;
16: end
17: return $\bot$;
18: $s \leftarrow 1$;
19: while True do
20: $n_l \leftarrow cs^{\alpha}/\beta$; $k_l \leftarrow cs \ln |T|$;
21: for each $(k, n_t) \in [1, k_l] \times [1, n_l]$ do
22: if $(k, n_t)$ has not been visited before then
23: $P \leftarrow \text{Search}(T, k, n_t, s)$;
24: if $P \neq \bot$ then return $P$;
25: end
26: end
27: $s \leftarrow s + 1$;
28: end

For the second condition, $T_{\text{poly}}$ assumes that there is an upper bound $k$, and searches among term sets with at most $k$ terms. If the search process guarantees to find a valid term set with a high probability when $k$ is larger than a small bound, which is polynomial to the size of the target program and sub-linear to the number of examples, the second condition of Lemma 6.1 can be satisfied by iteratively trying all possible $k$ from small to large.

6.2 Algorithm

The pseudo-code of $T_{\text{poly}}$ is shown as Algorithm 1. $T_{\text{poly}}$ is configured by a domain solver $S_T$, which is used to synthesize terms, and a constant $c$, which is used to configure bounds used in $T_{\text{poly}}$. We assume that $S_T$ can discover the case where there is no valid solution, and returns $\bot$ at this time.

The algorithm of $T_{\text{poly}}$ is comprised of three parts. The first part implements the random sampling discussed previously, as function GetCandidatePrograms$(\cdot)$ (abbreviated as Get$(\cdot)$). Get$(\cdot)$ takes four inputs: examples is a set of input-output examples, $k$ is an upper bound on the number of terms, $n_T$ is the number of examples provided to $S_T$ and $s$ is an upper bound on the size of terms.
Guided by Lemma 6.2, \texttt{Get()} returns a set of programs that covers at least \(k^{-1}\) portion of examples. The body of \texttt{Get()} is a repeated sampling process (Line 3). In each turn, \(n_t\) examples are sampled (Line 4), and solver \(S_t\) is invoked to synthesize a program from these sampled examples (Line 5). \texttt{Get()} collects all valid results (Line 7) and returns them to \texttt{Search()} (Line 10).

Note that \texttt{Get()} only considers those programs of which the sizes are at most \(cs^\alpha n_t^\beta\) (Line 6). This bound comes from the definition of Occam solvers (Definition 4.2): When all examples provided to \(S_t\) correspond to the same target term and the size of this term is at most \(s\), with a high probability, the term found by \(S_t\) will be no larger than \(cs^\alpha n_t^\beta\). However, in other cases, the selected examples may happen to correspond to some other unwanted terms: At this time, the term found by \(S_t\) may be much larger than the target term. Therefore, \(T_{\text{poly}}\) sets this limitation on the size and rejects those programs that are too large. This bound can be safely replaced by any function that is polynomial to \(s\) and sub-linear to \(n_t\) without affecting \(T_{\text{poly}}\) to be an Occam term solver.

The second part implements the backtracking as function \texttt{Search()} (Lines 11-18). Given a set of examples \textit{examples} and size limit \(k\), \texttt{Search()} searches for a set of at most \(k\) terms that covers all examples. \texttt{Search()} invokes function \texttt{Get()} to obtain a set of possible terms (Line 14), and then recursively tries each of them until a valid term set is found (Lines 15-16).

The third part of \(T_{\text{poly}}\) selects proper values for \(k\), \(n_t\) and \(s\) iteratively (Lines 19-29). In each turn, \(T_{\text{poly}}\) considers the case where the number of target terms and their sizes are all \(O(s)\) and select proper values for \(n_t\) and \(k\) in the following ways:

- By Theorem 4.6, when the size of the target term is \(O(s)\), \(S_t\) requires \(O(s^{\alpha/(1-\beta)})\) examples to guarantee the accuracy. Therefore, \(T_{\text{poly}}\) sets the upper bound of \(n_t\) to \(cs^\alpha/(1-\beta)\).
- Also by Theorem 4.6, when \(n_t\) is set to \(cs^\alpha/(1-\beta)\), the term synthesized by \(S_t\) may still differ with the target term on a constant portion of inputs. As a result, \(T_{\text{poly}}\) may use \(O(\ln n)\) times more terms to cover all examples in \(T\). Therefore, \(T_{\text{poly}}\) sets the upper bound of \(k\) to \(cs\ln n\).

The time cost of \texttt{Get()} and \texttt{Search()} grows rapidly when \(k\) and \(n_t\) increases, \(T_{\text{poly}}\) tries all values of \(k\) and \(n_t\) from small to large (Lines 22-27), instead of directly using the largest possible \(k\) and \(n_t\). The iteration ends immediately when a valid term set is found (Line 25).

### 6.3 Properties of \(T_{\text{poly}}\)

In this section, we discuss the properties of \(T_{\text{poly}}\). As a meta solver, \(T_{\text{poly}}\) guarantees to be an Occam term solver when \(S_t\) is an Occam term solver on the term space.

**Theorem 6.3.** \(S_t\) is an \((\alpha, \beta)\)-Occam solver on \(T(F_C) \implies T_{\text{poly}}\) is an \((\alpha'+1, \beta')\)-Occam term solver on \(F_C\) for any \(\alpha' > \alpha, \beta < \beta' < 1\), where \(T(F_C)\) is defined as \(\{(P, \{I\}) | ((P, P_c), I) \in F_C, I \subseteq \|\}\}.

Then, we discuss the time cost of \(T_{\text{poly}}\). With a high probability, \(T_{\text{poly}}\) invokes \texttt{Search()} only polynomial times, but the time cost of \texttt{Search()} may not be polynomial. By Algorithm 1, an invocation of \texttt{Search()} of depth \(i\) on the recursion tree samples \(n_t(k-i)^{n_t}\) times. In the worst case, \(S_t\) successfully synthesizes programs for all these samples, and the results are all different. At this time, \texttt{Search()} will recurse into \(n_t(k-i)^{n_t}\) different branches. Therefore, for each \(n_t, k\), domain solver \(S_t\) will be invoked \(n_t^k(k!)^{n_t}\) times in the worst case.

However, in practice, \(T_{\text{poly}}\) is usually much faster than the worst-case because:

- The domain solver \(S_t\) usually fails when the random examples correspond to different target terms, as the expressive ability of term domain \(P_t\) is usually limited.
- For those incorrect terms that happen to be synthesized, they seldom satisfy the requirement on the size and the number of covered examples (Line 6 in Algorithm 1).

In the best case where \texttt{Get()} never returns a term that is not used by the target program, \(S_t\) will be invoked at most \(n_t^2k^{n_t}\) times: Such a bound is much smaller than the worst case.
Algorithm 2: The framework of $\mathcal{U}_{\text{poly}}$.

**Input:** A term set $P = \{p_1, \ldots, p_m\}$, a PBE task $T$ and a condition solver $\mathcal{C}$.

**Output:** A program in $(P, \mathcal{P}_c)$ satisfying all examples.

1. $\text{conditionList} \leftarrow \{\}$.
2. for $i \leftarrow 1; i < m; i \leftarrow i + 1$ do
   3. examples $\leftarrow \{(i, \text{false}) | (i, O) \in T - \text{Covered}(p_i, T)\}$;
   4. examples $\leftarrow \{(i, \text{true}) | (i, O) \in \text{Covered}(p_i, T) - \bigcup_{j=i+1}^{m} \text{Covered}(p_j, T)\}$;
   5. $c_i \leftarrow \mathcal{C}(\text{examples})$;
   6. $\text{conditionList}.\text{Append}(c_i)$; $T \leftarrow \{(i, O) \in T | \lnot c_i(I)\}$;
7. end
8. result $\leftarrow p_m$;
9. for $i \leftarrow m - 1; i > 0; i \leftarrow i - 1$ do
   10. result $\leftarrow$ (if $\text{conditionList}_i$ then $p_i$ else result);
11. end
12. return result;

7 UNIFIER

7.1 Overview

$\mathcal{U}_{\text{poly}}$ unifies terms into a decision list, a structure proposed by Rivest [1987] for compactly representing decision procedures. $\mathcal{U}_{\text{poly}}$ unifies a term set $P = \{p_1, \ldots, p_m\}$ into the following form:

$$\text{if } (c_1) \text{ then } p_1 \text{ else if } (c_2) \text{ then } p_2 \text{ else } \ldots \text{ if } (c_{m-1}) \text{ then } p_{m-1} \text{ else } p_m$$

where $c_1, \ldots, c_{m-1}$ belong to DNF($\mathcal{P}_c$), the disjunctive normal form comprised of conditions in $\mathcal{P}_c$. DNF($\mathcal{P}_c$) is defined together with another two sets $L(\mathcal{P}_c)$ and $CL(\mathcal{P}_c)$, where the set of literals $L(\mathcal{P}_c)$ includes conditions in $\mathcal{P}_c$ and their negations, the set of clauses $CL(\mathcal{P}_c)$ includes the conjunctions of subsets of $L(\mathcal{P}_c)$, and the set of DNF formulas DNF($\mathcal{P}_c$) includes disjunctions of subsets of $CL(\mathcal{P}_c)$.

We use notation $(\mathcal{P}_t, \mathcal{P}_c)_{DL}$ to denote the set of decision lists where if-terms and if-condition belong to $\mathcal{P}_t$ and DNF($\mathcal{P}_c$) respectively. In the following lemma, we show that $(\mathcal{P}_t, \mathcal{P}_c)_{DL}$ is a suitable normal form for designing an Occam unifier, because for any program in $(\mathcal{P}_t, \mathcal{P}_c)$, there is always a semantically equivalent program in $(\mathcal{P}_t, \mathcal{P}_c)_{DL}$ with a close size.

**Lemma 7.1.** For any conditional domain $\mathbb{D}$ and any program $p \in (\mathcal{P}_t, \mathcal{P}_c)$, there exists a program $p' \in (\mathcal{P}_t, \mathcal{P}_c)_{DL}$ such that (1) $p'$ is semantically equivalent to $p$ on $\mathbb{I}$, and (2) size($p'$) $\leq 2 \text{size}(p)^2$.

$\mathcal{U}_{\text{poly}}$ decomposes the unification task into $m-1$ PBE tasks for $c_1, \ldots, c_{m-1}$ respectively. Algorithm 2 shows the framework of $\mathcal{U}_{\text{poly}}$, which synthesizes conditions in order (Lines 2-7). For each term $p_i$ and each remaining example $e = (I, O)$, there are three possible cases:

- If $p_i$ is not consistent with $e$, the value of if-condition $c_i$ must be false on input $I$ (Line 3).
- If $p_i$ is the only program in $p_i, \ldots, p_n$ that is consistent with $e$, the value of $c_i$ must be true on input $I$ (Line 4).
- Otherwise, the value of $c_i$ does not matter, as $p_i$ is not the last choice. Therefore, $\mathcal{U}_{\text{poly}}$ ignores this example: If the synthesized $c_i$ is false on input $I$, $e$ will be left to be subsequent terms.

In this way, $\mathcal{U}_{\text{poly}}$ obtains a PBE task for $c_i$, and it invokes a DNF solver $\mathcal{C}$ to solve it (Line 5). Then, $\mathcal{U}_{\text{poly}}$ excludes all examples covered by $c_i$ (Line 6) and moves to the next term. At last, $\mathcal{U}_{\text{poly}}$ unifies all terms and conditions into a complete program (Lines 8-11).

Just like $T_{\text{poly}}$, unifier $\mathcal{U}_{\text{poly}}$ is an Occam unifier when $\mathcal{C}$ is an Occam solver on DNF($\mathcal{P}_c$).
Algorithm 3: The pseudo code of clause solver $C_{\text{CL}}$.

**Input:** A condition space $P_c$ and a PBE task $T$.
**Output:** A program in $\text{CL}(P_c)$ satisfying all examples or $\bot$.

1. **Function** $\text{SimplifyClause}(c_u, T)$:
   2. $\text{remNeg} \leftarrow I_N(T)$; $c^* \leftarrow \emptyset$;
   3. **while** $\text{remNeg} \neq \emptyset$ **do**
      4. $l^* \leftarrow \arg\max_{l \in c_u} \left( |N(l(T), I) \cap \text{remNeg}|/\text{size}(l) \right)$;
      5. $c^* \leftarrow c^* \cup \{l^*\}$; $\text{remNeg} \leftarrow \text{remNeg} - N(l(T), I)$;
   6. **end**
   7. **return** $c^*$.

8. $c_u \leftarrow \{l \in L(P_c) \mid I_p(T) \subseteq P(l(T), l)\}$;
9. **if** $I_N(T) \not\subseteq N(l(T), c_u)$ **then** **return** $\bot$;
10. **return** $p_c(\text{SimplifyClause}(c_u, T))$;

**Lemma 7.2.** $C$ is an $(\alpha, \beta)$-Occam solver on $\text{DNF}(\mathcal{F}_C) \Rightarrow U_{\text{poly}}$ is an $(4\alpha', \beta)$-Occam unifier on $\mathcal{F}_C$ for any $\alpha' > \alpha$, where $\text{DNF}(\mathcal{F}_C)$ is defined as $\{(\text{DNF}(P_c), \mathcal{I}') \mid ((P, P_c), \mathcal{I}) \in \mathcal{F}_C, \mathcal{I}' \subseteq \mathcal{I}\}$.

$C$ is a deterministic $(\alpha, \beta)$-Occam solver on $\text{DNF}(\mathcal{F}_C) \Rightarrow U_{\text{poly}}$ is a $(4\alpha, \beta)$-Occam unifier on $\mathcal{F}_C$.

By Lemma 7.2, the only problem remaining is to design an Occam solver for $\text{DNF}(\mathcal{F}_C)$. We shall gradually build such a condition solver in the following two subsections.

### 7.2 Condition Synthesis for Clauses

For the sake of simplicity, we start by introducing some useful notations:

- We regard a DNF formula $d$ as a set of clauses and regard a clause $c$ as a set of literals. We use notation $p_c(c)$ and $p_d(d)$ to represent their corresponding program respectively.

- For an input space $l$ and a condition $p$, we use $P(l, p)$ and $N(l, p)$ to denote the set of inputs where $p$ is evaluated to true and false respectively.

- For a PBE task $T$, we use $I(l, T)$, $I_p(T)$ and $I_N(T)$ to denote the inputs of examples, positive examples and negative examples in $T$ respectively, i.e.:

$$I(l, T) := \{i \mid (i, O) \in T\} \quad I_p(T) := \{i \mid (i, \text{true}) \in T\} \quad I_N(T) := \{i \mid (i, \text{false}) \in T\}$$

For a PBE task $T$ on domain $(\text{DNF}(P_c), l)$, a valid condition $p$ should satisfy the following two conditions: (1) $p$ takes true on all positive examples in $T$, i.e., $I_p(T) \subseteq P(l(T), p)$; (2) $p$ takes false on all negative examples in $T$, i.e., $I_N(T) \subseteq N(l(T), p)$. In this section, we build solver $C_{\text{CL}}$ for the subproblem where the target condition is assumed to be a single clause.

The pseudo-code of $C_{\text{CL}}$ is shown as Algorithm 3. $C_{\text{CL}}$ starts with the first condition of valid clauses: $I_p(T) \subseteq P(l(T), c)$. By the semantics of operator and, for any clause $c$ and any literal $l \in c$, $P(l(T), c)$ must be a subset of $P(l(T), l)$. Therefore, only those literals that cover all positive examples can be used in the result. $C_{\text{CL}}$ collects all these literals as clause $c_u$ (Line 8). Then the subsets of $c_u$ are exactly those clauses satisfying the first condition.

The remaining task is to find a subset $c^*$ of $c_u$ that satisfies the second condition, i.e., $I_N(T) \subseteq N(l(T), c^*) = \cup_{l \in c^*} N(l(T), l)$. Meanwhile, to make $C_{\text{CL}}$ an Occam solver, the size of $p_c(c^*)$ should be as small as possible. This problem is an instance of weighted set covering: $C_{\text{CL}}$ needs to select some sets from $\{N(l(T), l) \mid l \in c^*\}$ to cover $I_N(T)$. **Weighted set covering** is known to be difficult: Moskiovitz [2011] proves that for any $\epsilon > 0$, there is no polynomial-time algorithm that always finds a solution at most $(1 - \epsilon) \ln n$ times worse than the optimal, unless $\text{NP} = \text{P}$, where $n$ is $|I_N(T)|$ in our case.

Proc. ACM Program. Lang., Vol. 5, No. OOPSLA, Article 167. Publication date: October 2021.
In our implementation, we use a standard greedy algorithm for weighted set covering, which runs in polynomial time and always finds a solution at most \((\ln n + 1)\) times worse than the optimal (Lines 1-7). \(\mathcal{C}_{CL}\) maintains set \(\text{remNeg}\), representing the set of negative examples that have not been covered yet (Line 3). In each turn, \(\mathcal{C}_{CL}\) selects the literal \(l'\) which covers the most uncovered negative examples in each unit of size (Line 5) and includes \(l'\) into the result (Line 6).

The size of the clause found by \(\mathcal{C}_{CL}\) is bounded, as shown in Lemma 7.3. Therefore, \(\mathcal{C}_{CL}\) is an Occam solver, as shown in Corollary 7.4. The time complexity of \(\mathcal{C}_{CL}\) is polynomial to \(|\mathcal{P}_c|\) and \(|T|\), and thus \(\mathcal{C}_{CL}\) is efficient when \(\mathcal{P}_c\) is not large.

**Lemma 7.3.** Given condition space \(\mathcal{P}_c\) and PBE task \(T\), let \(c^*\) be the smallest valid clause and \(c\) be the clause found by \(\mathcal{C}_{CL}\). Then size\((p_c(c))\) < 2size\((p_c(c^*))\)(ln|\(T|\) + 1).

**Corollary 7.4.** For any \(0 < \beta < 1\), \(\mathcal{C}_{CL}\) is an \((1, \beta)\)-Occam solver on all possible clause domains.

### 7.3 Condition Synthesis for Disjunctive Normal Forms

In this section, we implement an Occam solver \(\mathcal{C}\) for disjunctive normal forms. By the semantics of operator \(\lor\), we could obtain a lemma that is similar with Lemma 6.2 in form.

**Lemma 7.5.** Let \(T\) be a PBE task and \(d\) be a DNF formula satisfying all examples in \(T\), then:

- All clauses in \(d\) must be false on all negative examples in \(T\), i.e., \(\forall c \in d, \mathbb{I}_N(T) \subseteq N(\mathbb{I}(T), c)\).
- There exists a clause in \(d\) that is true on at least \(|d|^{-1}\) portion of positive examples in \(T\), i.e., \(\exists c \in d, |\mathbb{P}(\mathbb{I}(T), c)| \geq |d|^{-1}|\mathbb{P}(\mathbb{T})|\).

By this lemma, \(\mathcal{C}\) can be implemented similarly as \(\mathbb{T}\_poly\) by regarding \(\mathcal{C}_{CL}\) as the domain solver, as shown in Algorithm 4. Comparing with the counterpart in \(\mathbb{T}\_poly\), there are two main differences:

- GetPossibleClause() finds a set of clauses that are evaluated to \(false\) on all negative examples, and are evaluated to \(true\) on at least \(k^{-1}\) portion of positive examples (Line 4).
- For the efficiency of \(\mathcal{C}_{CL}\), \(\mathcal{C}\) iteratively selects a parameter \(s'\) (Line 13). In each iteration, only those literals with size at most \(s'\) are available (Line 15).

Our implementation of GetPossibleClause() (abbreviated as Get()) optimizes the sampling algorithm in \(\mathbb{T}\_poly\) by opening the box of clause solver \(\mathcal{C}_{CL}\). By Algorithm 3, \(\mathcal{C}_{CL}\) synthesizes clauses in two steps. It firstly finds a set \(c_u\) of all usable literals and then simplifies it greedily. To synthesize a usable clause, set \(c_u\) should satisfy all negative examples and at least \(k^{-1}\) portion of positive examples. We find the number of different \(c_u\) satisfying this condition is usually small in practice. Therefore, Get() tries to find all possible \(c_u\), and simplifies them using function SimplifyClause().

Given an input space \(\mathbb{I}\) and a set of literals \(L\), define relation \(\sim_\mathbb{I}\) on clause space CL(\(L\)) as \(c_1 \sim_\mathbb{I} c_2 \iff \forall I \in \mathbb{I}, [p_c(c_1)](I) = [p_c(c_2)](I)\), i.e., \(\sim_\mathbb{I}\) represents the relation of semantically equivalence on input space \(\mathbb{I}\). Under relation \(\sim_\mathbb{I}\), CL(\(L\)) is divided into equivalent classes. We denote the class corresponding to clause \(c\) as \([c]\). It is easy to show that each class \([c]\) contains a globally largest clause that is the union of all its elements, i.e., \(\exists c' \in [c], c' = (\cup x for x \in [c])\). Then, we introduce the concept of representative clauses to denote the set of all possible \(c_u\) generated by \(\mathcal{C}_{CL}\).

**Definition 7.6 (Representative Clauses).** Given an input space \(\mathbb{I}\), a size limit \(k\), and a set of literals \(L\), representative set \(R(\mathbb{I}, k, L) \subseteq CL(\mathbb{L})\) includes all clause \(c\) satisfying: (1) \(c\) is the largest clause in \([c]\); (2) \(c\) takes \(true\) on at least \(k^{-1}\) portion of the inputs, i.e., \(|\mathbb{P}(\mathbb{I}, c)| \geq k^{-1}|\mathbb{I}|\).

According to this definition, \(R(\mathbb{I}\_\mathbb{P}(T), k, L)\) is the set of possible \(c_u\) when the PBE task is \(T\), the size limit is \(k\) and the set of available literals is \(L\). Our implementation of Get() is shown as Algorithm 5. Get() maintains set result which is equal to \(R(\mathbb{I}\_\mathbb{P}(T), k, L')\) for some hidden literal set \(L'\). Initially,
Algorithm 4: DNF solver $\mathcal{C}$ for disjunctive normal forms

Input: A condition space $P_c$, a PBE task $T$ and a constant $c_0$.
Output: A DNF formula in $\text{DNF}(P_c)$ satisfying all examples.

1. $\text{Function Search(literals, } T, k, s) :$
   2. if $|I_P(T)| = 0$ then return $\emptyset$;
   3. if $(\text{literals, } T, k)$ is visited before or $k = 0$ then return $\top$;
   4. for each $c \in \text{GetPossibleClause}(\text{literals, } T, k)$ do
      5. if size($p_c(c)$) > $c_0 s \ln |T|$ then continue;
      6. $\text{searchResult} \leftarrow \text{Search}(\text{literals}, T - \{l, \text{true}\} \in T \mid \| p_c(c) \| (l) = \text{true}, k - 1, s)$;
      7. if $\text{searchResult} \neq \bot$ then return $\{c\} \cup \text{searchResult}$;
   8. end
   9. return $\top$;
10. $s \leftarrow s + 1$;
11. while $\text{True}$ do
12.   $k_f \leftarrow c_0 s$;
13.   for each $(k, s') \in [1, k_f] \times [1, s]$ do
14.     if $(k, s')$ has not been visited before then
15.        $P_c \leftarrow \text{GetConditionsWithSizeBound}(P_c, s')$;
16.        $d \leftarrow \text{Search}(L(P_c), T, k, s)$;
17.        if $d \neq \bot$ then return $p_d(d)$;
18.     end
19.   end
20. end

Algorithm 5: Implementation of $\text{GetPossibleClause()}$ in $\mathcal{C}$.

Input: A set of literals $\text{literals}$, a PBE task $T$, and an upper bound $k$.
Output: A set of possible clauses according to Lemma 7.5.

1. $\text{result} \leftarrow \emptyset$;
2. for each $l \in \text{literals}$ do
3.   for each $c \in \text{result}$ do
4.     if $|P(\| l \| P(T), c \cup \{l\})| \geq k^{-1} |\| P(T)\| |$ then $\text{result}.\text{Insert}(c \cup \{l\})$;
5.   end
6. for each $c \in \text{result}$ do
7.     if $\exists (P(\| l \| P(T), c) = P(\| l \| P(T), c') \cap c \subseteq c'$ then $\text{result}.\text{Delete}(c)$;
8. end
9. return $\{\text{SimplifyClause}(c) \mid c \in \text{result} \land \exists N(T) \subseteq N(\| l \| T, c)\}$

$L'$ is empty and thus $\text{result}$ includes only an empty clause, i.e., true (Line 1). $\text{Get()}$ considers all literals in order (Lines 2-9). In each turn, a new literal is inserted to $L'$ and thus $\text{result}$ is updated correspondingly (Lines 3-8). At last, $\text{result}$ is equal to $R(\| l \| P(T), k, \text{literals})$, and thus $\text{Get()}$ simplifies and returns all valid clauses in $\text{result}$ (Line 10).

We prove that $\mathcal{C}$ is an Occam solver, as shown in Lemma 7.7. Combining with Lemma 7.2, $\mathcal{U}_{\text{poly}}$ is also an Occam unifier, as shown in Theorem 7.8.

**Lemma 7.7.** For any $0 < \beta < 1$, $\mathcal{C}$ is a $(2, \beta)$-Occam solver on $\text{DNF}(P_c)$. 

Proc. ACM Program. Lang., Vol. 5, No. OOPSLA, Article 167. Publication date: October 2021.
THEOREM 7.8. For any $0 < \beta < 1$, $U_{\text{poly}}$ is a $(8, \beta)$-Occam unifier on $T_C$.

Finally, we prove that PolyGen is an Occam solver by combining Theorem 7.8, Theorem 6.3 and Theorem 5.7, as shown in Theorem 7.9.

THEOREM 7.9. $S_I$ is an $(\alpha, \beta)$-Occam solver on $T(T_C)$ with $\beta < \frac{1}{8}$ $\implies$ PolyGen is an $(8(\alpha' + 1), 8\beta')$-Occam solver on $T_C$ for any $\alpha' > \alpha, \beta < \beta' < \frac{1}{8}$.

Note that the bound in Theorem 7.9 is loose in practice, as it considers all corner cases for the sake of preciseness. For example, while analyzing solver $C$, we consider the case where both the number of clauses and the size of the clauses are linear to the size of the target condition $d'$. At this time, there must be $O(1)$ clauses of which the size is $\Omega(|d'|^2)$, and $\Omega(|d'|^2)$ clauses of which the size is $O(1)$. Such an if-condition is seldom used in practice.

8 IMPLEMENTATION

We instantiate our approach PolyGen on the domains of conditional linear integer arithmetic (CLIA). Our implementation is in C++, and is available online [Ji et al. 2021].

Our implementation supports the family of CLIA domains $T_I$ defined in the evaluation of the study on Eusolver [Alur et al. 2017], where different domains only differ in the number of inputs. Concretely, given the number of inputs $n$, a domain $D_I = ((P_T, P_c), I) \in T_I$ is defined as follows.

- Term space $P_T$ contains all linear integer expressions of input variables $x_1, \ldots, x_n$.
- Condition space $P_c$ contains all arithmetic comparisons between linear expressions and their boolean expressions, i.e., $P_c$ is the smallest set satisfying the following equation.

$$P_c = \{ e_1 \circ e_2 \mid e_1, e_2 \in P_T, \circ \in \{<, \leq, =\} \} \cup \{ c_1 \circ c_2 \mid c_1, c_2 \in P_c, \circ \in \{\text{and}, \text{or}\} \} \cup \{ \neg c \mid c \in P_c \}$$

By Definition 5.4, $D_I$ is an if-closed conditional domain.

- Input space $I$ contains integer assignments to input variables. For simplicity, we assume $I$ contains all assignments in a bounded range: $I := \{ (w_1, \ldots, w_n) \mid w_i \in [\text{int}\_\text{min}, \text{int}\_\text{max}] \}$.

To implement $T_{\text{poly}}$ and $U_{\text{poly}}$, we set parameters $c$ and $c_0$ as 2 by default.

At last, by Theorem 7.9, an Occam solver $S_I$ on $T(T_I)$ is required to instantiate PolyGen on $T_I$. We implement $S_I$ as an optimized solver that always synthesizes the smallest valid program. Concretely, given PBE task $T$, $S_I$ synthesizes $c_0 + c_1 x_1 + \cdots + c_n x_n$ by solving the following optimization problem with respect to $c_0, \ldots, c_n$:

Minimize $\text{size}(c_0 + c_1 x_1 + \cdots + c_n x_n)$ Subject to $\forall ((w_1, \ldots, w_n), O) \in T, \sum_{i=1}^n w_i c_i + c_0 = O$

This problem is an instance of integer linear programming (ILP), and $S_I$ solves it by invoking Gurobi [Gurobi Optimization 2021], a state-of-the-art solver for ILP. Clearly, $S_I$ is an $(1, 0)$-Occam solver, and thus by Theorem 7.9, PolyGen is an Occam solver on $T_I$.

9 EVALUATION

To evaluate PolyGen, we report several experiments to answer the following research questions:

- **RQ1:** How does PolyGen compare against existing PBE solvers?
- **RQ2:** How do term solver $T_{\text{poly}}$ and unifier $U_{\text{poly}}$ affect the performance of PolyGen?
- **RQ3:** How do the values of parameter $c$ and $c_0$ affect the performance of PolyGen?

9.1 Experimental Setup

Baseline Solvers. We compare PolyGen with three existing PBE solvers, Esolver [Alur et al. 2013], Eusolver [Alur et al. 2017], and Euphony [Lee et al. 2018], which represent the state-of-the-art of three different methods on improving generalizability:
(1) The first method is guided by the principle of Occam’s Razor, which guarantees to synthesize the smallest valid program. On CLIA, E Solver is the best existing solver following this method, which enumerates programs in the increasing order of size, and prunes off useless programs via a strategy namely observational equivalence.

(2) The second method combines the first method with efficient synthesis techniques heuristically, and thus makes a trade-off between generalizability and efficiency. Among them, Eusolver combines the principle of Occam’s Razor with the STUN framework by requiring the term solver to enumerate terms in the increasing order of size.

(3) The third method uses a learned model to guide the synthesis. In this category, Euphony is the state-of-the-art among solvers available on CLIA. Euphony is based on Eusolver and uses a model based on structural probability to guide the search of Eusolver.

Besides, as the efficiency of E Solver is limited on CLIA, we also compare PolyGen with an improved solver E Solver+2, which adds a special treatment for if-then-else operators to E Solver and still guarantees to synthesize the smallest valid program. When using the if-then-else operator to construct larger programs, E Solver+ first selects an existing Boolean expression as the if-condition. Then, for each existing program p, E Solver+ tries p as the if-term only when p is the smallest program satisfying a subset of examples consistent with the corresponding if-branch. Concretely, let E_i (E_f) be the set of examples where the selected if-condition is evaluated to true (false), p is considered only when there exists a subset E’ ⊆ E_i (E_f) such that p is the smallest program satisfying all examples in E’.

Oracle Models. Our evaluation follows the framework of OGIS [Jha and Seshia 2017]. We consider two different models of oracles, which cover major usages of PBE solvers in practice.

(1) In model O_V, the oracle can verify whether a program is correct, and can provide a counter-example if the program is incorrect. To synthesize from these oracles, the framework of counter-example guided inductive synthesis (CEGIS) [Solar-Lezama et al. 2006] is usually used. Given an oracle O in O_V and a PBE solver S, we run CEGIS with solver S to synthesize a program from O. We measure the generalizability of S on O as the number of examples finally used by S to synthesize a correct program, which is equal to the number of CEGIS turns, and we measure the efficiency of S as the total time cost of the CEGIS framework.

(2) In model O_R, the oracle cannot verify the correctness of a program but can provide a set of input-output examples. At this time, a program is usually synthesized by (1) invoking the oracle to generate as many examples as possible under some limits on resource, and then (2) invoking a PBE solver to synthesize a program from these examples. To evaluate the performance of a PBE solver S on an oracle O in O_R, we assume that there is a corresponding oracle O’ in O_R that could verify whether the synthesized program is completely correct for O. We run S in a similar way as CEGIS: starting from an empty set of examples, in each turn, we run S on all existing examples. If the synthesis result is verified to be incorrect by O’, we request a new example from O and then start a new turn. We measure the generalizability of S on O as the total number of used examples. Because the PBE solver is usually invoked only once in practice, we measure the efficiency of S as the time cost of the last invocation.

Benchmark. Our evaluation is conducted on a dataset D of 100 benchmarks. For each benchmark, two different oracles O_V and O_R are provided, which correspond to models O_V and O_R respectively. The programs are synthesized in a domain of CLIA as stated in Section 8. D consists of two parts, D_S and D_D, each obtained from an existing dataset.

*We thank an anonymous reviewer for proposing this baseline solver.*
Dataset $\mathcal{D}_S$. The first dataset $\mathcal{D}_S$ consists of 82 benchmarks collected from the general track$^3$ in SyGuS-Comp [Alur et al. 2019], where each benchmark is provided with a logic specification $\Phi$.

To implement two oracles, we apply the algorithm $\mathcal{A}$ used in Eusolver [Alur et al. 2017], which could (1) get the correct output for a given input, (2) get a counter-example for an incorrect result:

- **Oracle $\mathcal{O}_V$.** Given a candidate program $p$, $\mathcal{O}_V$ firstly verifies the correctness of $p$ via an SMT solver and invokes $\mathcal{A}$ to generate a counter-example if $p$ is incorrect.
- **Oracle $\mathcal{O}_R$.** $\mathcal{O}_R$ randomly selects an input and invokes $\mathcal{A}$ to complete it into an example.

$\mathcal{A}$ is applicable only for special specifications, namely *point-wise*: specification $\Phi$ is point-wise if it only relates an input point to its output. Therefore, we filter out those benchmarks where the specification is not point-wise, and those benchmarks that cannot be solved by a CLIA program.

Dataset $\mathcal{D}_D$. The second dataset $\mathcal{D}_D$ consists of 18 tasks for synthesizing a combinator in a divide-and-conquer algorithm, which are collected by Farzan and Nicolet [2017]. The synthesized program can be converted to a divide-and-conquer algorithm using ParSynt [Farzan and Nicolet 2017].

For example, the following specifies a task for synthesizing a combinator $c$ in the divide-and-conquer algorithm for the maximum segment sum ($mss$) problem.

$$\forall l_1, l_2 : \text{List} \mid c (\text{mss}(l_1), \text{mss}(l_2), \text{mps}(l_1), \text{mps}(l_2), \text{mts}(l_1), \text{mts}(l_2)) = \text{mss}(l_1 + l_2) \tag{1}$$

where $l_1 + l_2$ represents the list concatenation of lists $l_1, l_2$, $\text{mps}$ represents the maximum prefix sum of a list and $\text{mts}$ represents the maximum suffix sum of a list. In this case, a valid combinator can be obtained from equation $\text{mss}(l_1 + l_2) = \max(\text{mss}(l_1), \text{mss}(l_2), \text{mts}(l_1) + \text{mps}(l_2))$.

We choose this dataset because of the following reasons.

1. It is a typical application scenario for the oracle model $\mathcal{O}_R$. On the one hand, it is difficult to verify the correctness of a program, as the specification involves complex list operations that are difficult to model in SMT-Lib. On the other hand, it is easy to collect input-output examples for the combinator, as all inputs and the output are generated by some executable function, as shown in Equation 1.

2. if-then-else operators are frequently used in the combinator, as there are usually many possible cases when merging two halves. For example, the combinator for mss deals with 3 cases, corresponds to if-terms $\text{mss}(l_1)$, $\text{mss}(l_2)$ and $\text{mts}(l_1) + \text{mps}(l_2)$ respectively.

3. Synthesizing the combinator directly is difficult, as it can be rather complex in practice. ParSynt can successfully synthesize the combinator only when a program sketch is provided.

Though it is difficult to verify the correctness of the synthesized program against the specification involving complex list operations, it is not difficult to verify the equivalence of two CLIA programs. The original dataset provides the ground truth program $c^*$ for each task, and thus we can still implement the two oracles.

- Given a candidate program $p$, $\mathcal{O}_V$ uses an SMT solver to verify whether $p$ and $c^*$ are semantically equivalent on the input space.
- $\mathcal{O}_R$ randomly selects an input and runs $c^*$ to get the corresponding output.

Configurations. All of the following experiments are conducted on Intel Core i7-8700 3.2GHz 6-Core Processor with 48GB of RAM. We use Z3 [de Moura and Bjørner 2008] as the underlying SMT solver for oracles in model $\mathcal{O}_V$, and generate random inputs for oracles in model $\mathcal{O}_R$ by setting each input variable to a random integer according to a uniform distribution over $[-50, 50]$.

---

$^3$There is also a CLIA track in SyGuS-Comp. We use the dataset of the general track here because (1) all benchmarks in the CLIA track are included in the general track, (2) the general track includes additional benchmarks that are collected from a varies of domains and can also be solved by programs in the CLIA syntax.
Table 2. The results of comparing PolyGen with baselines.

| Model   | #Solved | #Example | Time Cost | Model   | #Solved | #Example | #Example≥ | Time Cost |
|---------|---------|----------|-----------|---------|---------|----------|-----------|-----------|
| PolyGen | 97      | ×1.000   | ×1.000    | PolyGen | 93      | ×1.000   | ×1.000    |
| Esolver | 9       | ×0.969   | ×3.668    | Esolver | 9       | ×1.065   | ×52.271   |
| Eusolver | 26     | ×0.912   | ×8.233    | Euphony | 65      | ×2.332   | ×6.140    | ×1.639    | ×3.320   | ×12.825  |
| Euphony | 51      | ×2.271   | ×7.417    | Euphony | 53      | ×1.115   | ×3.302    | ×15.067   |

For each execution, we set the time limit as 120 seconds, the memory limit as 8 GB, and the example number limit as $10^4$. Besides, as both PolyGen and oracles in model $O_R$ have randomness, we repeat all related executions 5 times and consider the average performance only.

9.2 Exp1: Comparison of Approaches (RQ1)

**Procedure.** For each oracle model, we compare PolyGen with Esolver, Eusolver and Euphony on all benchmarks in $D$. Among them, the experiment setting for Euphony is slightly different from others, as Euphony requires a labeled training set. We run Euphony in two steps:

- First, for those benchmarks in $D$ where the target program is not explicitly provided, we label them using the program synthesized by PolyGen.
- Second, we run Euphony using 3-fold cross-validation. We divide the dataset $D$ into three subsets. On each subset, we run Euphony with the model learned from the other two subsets. One delicate point is that $D$ contains benchmarks that are almost the same except the number of input variables. We put these benchmarks in the same subset and thus avoid data leaks.

**Results.** The results are summarized in Table 2 while more details are drawn as Figure 2. To compare the generalizability, in each comparison, for each benchmark solved by both PolyGen and the baseline solver, we record the ratio of the number of examples used by the baseline solver to the number of examples used by PolyGen. The geometric mean of these ratios is listed in column #Example. Similarly, to compare the efficiency, in each comparison, we record the ratio of the time cost of the baseline solver to the time cost of PolyGen for those benchmarks solved by both solvers and list the geometric mean of these ratios in column Time Cost.

Comparing with Esolver and Eusolver+, the generalizability of PolyGen is close to both solvers on both oracle models. Recall that the theory of Occam learning used by PolyGen is only an approximation of the principle used by Esolver and Eusolver+: Occam learning relaxes the requirement from finding the smallest valid program to finding a valid program with a bounded size. The experimental result shows that such an approximation does not affect the generalizability too much in practice. Meanwhile, benefiting from the relaxed requirement on the size provided by Occam Learning, PolyGen performs significantly better on efficiency: PolyGen solves much more benchmarks comparing with Esolver and Eusolver+, with significant speed-ups on those commonly solved benchmarks. At last, the results show a tendency that the gap in the generalizability between PolyGen and Esolver (Eusolver+) grows on those difficult benchmarks. This result matches our theoretical results. The generalization bound of PolyGen is polynomial to the size of the simplest program while that of Esolver and Eusolver+ is linear.

We also compare the sizes of the programs synthesized by PolyGen with the smallest valid programs found by Esolver+. On those benchmarks solved by both solvers, the size of the program synthesized by PolyGen is 14.2% larger than the smallest valid program on average, and is exactly equal to the smallest on 20 out of 26 benchmarks. Among others, there is a single benchmark where
PolyGen synthesizes a program at least twice larger than the smallest. On this benchmark, the smallest program uses 2 relatively complex if-terms, while PolyGen synthesizes a larger program with 4 simpler if-terms. PolyGen makes such a decision because term solver T limits the size of each if-term to ensure that the total size of if-terms is bounded (Line 6 in Algorithm 1).

Comparing with Eusolver and Euphony, PolyGen performs significantly better on both generalizability and efficiency. The experimental result on the generalizability is consistent with our theoretical analysis, as PolyGen is an Occam solver but Eusolver, Euphony are not. Comparing with O\textsubscript{V}, the advantage of PolyGen on oracle model O\textsubscript{R} seems less attractive. The reason is that on O\textsubscript{R}, the cost of distinguishing an incorrect program increases, and thus the effect of synthesis algorithms is weakened. For example, in benchmark qn_neg_1.sl, it is hard to distinguish between the target program is \( p^* (x) = (\text{if } (x < 0) \text{ then } 1 \text{ else } 0) \) and a wrong program with a slightly different if-condition \( p'(x) = (\text{if } (x \leq 0) \text{ then } 1 \text{ else } 0) \) in model O\textsubscript{R}. The probability for a random input to distinguish them is smaller than 1% when the input is in the range \([-50, 50]\).

Please note that the comparison in Column \#Example suffers from a survivorship bias. On many benchmarks that PolyGen solves while Eusolver (Euphony) does not, PolyGen is likely to have better generalizability. To validate this point, we perform an extra experiment on model O\textsubscript{R}. We iteratively rerun Eusolver (Euphony) on those benchmarks where PolyGen solves but Eusolver (Euphony) does not. Starting from only 1 random example, we invoke Eusolver (Euphony) with an enlarged...
Table 3. The results of comparing PolyGen with weakened solvers.

| Model              | \(O_Y\) | \(O_R\) |
|--------------------|---------|---------|
| Solver             | #Solved | #Example | Time Cost | #Solved | #Example | Time Cost |
| PolyGen            | 97      | \(\times 1.000\) | \(\times 1.000\) | 93      | \(\times 1.000\) | \(\times 1.000\) |
| PolyGen\_T         | 73      | \(\times 1.154\) | \(\times 2.001\) | 75      | \(\times 0.960\) | \(\times 2.733\) |
| PolyGen\_U         | 71      | \(\times 1.644\) | \(\times 1.950\) | 69      | \(\times 1.288\) | \(\times 1.928\) |

time-limit of 30 minutes. If the solver synthesizes an incorrect program, we record the current number of examples as a lower bound of the generalizability, and then continue to the next iteration by doubling the number of examples. The iteration stops when Eusolver (Euphony) successfully synthesizes a correct program or times out. After adding these lower bounds, the geometric mean of the ratios between the lower bounds and the number of examples used by PolyGen is reported in column \(\#\text{Example}\leq\) of Table 2. The result justifies the existence of the survivorship bias as the ratios increase from \(\times 1.115 – \times 1.649 \rightarrow \times 3.320 – \times 3.302\). Please note that the new experiment still favors Eusolver and Euphony because (1) the iteration only provides a coarse lower bound on the number of required examples, (2) the survivorship bias still exists as Eusolver and Euphony still time out on 27 and 28 out of 93 benchmarks, respectively.

In terms of efficiency, PolyGen solves almost all benchmarks in \(\mathcal{D}\) on both oracle models. We investigate those benchmarks where PolyGen times out, and conclude two major reasons:

- As the time cost of \(T_{\text{poly}}\) grows quickly when the number of if-terms increases, PolyGen may time out when a large term set is used. For example, PolyGen fails in finding a valid term set for array_search\_15.sl where 16 different if-terms are used.

- As \(U_{\text{poly}}\) considers conditions in the increasing order of the size, PolyGen may time out when a large condition is used. For example, PolyGen times out on mpg_example3.sl where if-condition \(2x + 2y – z – 7 \leq 0\) is used. This defect can be improved by combining PolyGen with techniques on feature synthesis [Padhi and Millstein 2017].

To solve these hard benchmarks, another way is to further relax the requirement of Occam learning and thus make it possible to design more efficient synthesizers. Comparing with the previous strategy that requires synthesizing the smallest correct program, the main advantage of Occam learning is that it relaxes the constraints on the synthesizer and allows us to design a more efficient synthesizer with a theoretical guarantee on generalizability. Therefore, in our vision, a hierarchical theory on generalizability can be established, where the constraints of Occam learning may be further relaxed to enable more efficient synthesizers, and there may be different guarantees on generalizability according to the level and the direction of the relaxation. It will be one direction of future work.

At last, a noteworthy result is that Euphony performs even worse than Eusolver in our evaluation, implying that the model used in Euphony plays a negative role. One possible reason is that the target programs in our dataset are diverse and are difficult to be predicted by a simple probabilistic model considering only the dependency between program elements.

9.3 Exp2: Comparison of the Term Solver and the Unifier (RQ2)

**Procedure.** In this experiment, we test how \(T_{\text{poly}}\) and \(U_{\text{poly}}\) affect the performance of PolyGen.

Here, we implement two weakened solvers PolyGen\_T and PolyGen\_U: PolyGen\_T replaces term solver \(T_{\text{poly}}\) with the term solver \(T_E\) used in Eusolver, and PolyGen\_U replaces unifier \(U_{\text{poly}}\) with the unifier \(U_E\) used in Eusolver. For each oracle model, we run these solvers on all benchmarks in \(\mathcal{D}\).

**Results.** The results are summarized in Table 3 while more details are drawn as Figure 2.
Table 4. The results of comparing PolyGen with different values of $c$ and $c_0$.

| Model     | $O_Y$ #Solved | $O_Y$ #Example | $O_Y$ Time Cost | $O_R$ #Solved | $O_R$ #Example | $O_R$ Time Cost |
|-----------|----------------|----------------|-----------------|----------------|----------------|-----------------|
| PolyGen   | 97             | 1.000          | 1.000           | 93             | 1.000          | 1.000           |
| PolyGen$_1$ | 94             | 1.075          | 1.073           | 87             | 1.013          | 0.867           |
| PolyGen$_3$ | 94             | 0.994          | 0.940           | 90             | 0.934          | 0.752           |

As shown in Table 3, the unifier $U_{\text{poly}}$ improves a lot on both efficiency and generalizability. In contrast, $T_{\text{poly}}$ majorly contributes to the efficiency, as the generalizability of PolyGen changes little when $T_{\text{poly}}$ is replaced. The reason is that the number of examples required by the unifier usually dominates the number of examples required by the term solver because an example for synthesizing \textit{if}-conditions, where the output type is Boolean, provides much less information than an example for synthesizing \textit{if}-terms, where the output is an integer.

Besides the generalizability, one important advantage of $T_{\text{poly}}$ comparing with $T_E$ is that $T_{\text{poly}}$ can quickly find some complex \textit{if}-terms. For example, on benchmark mpg_example4.sl, $T_{\text{poly}}$ is able to find a complex term set $\{ 10x+20y+15z-99, 9y+25w-11, 11x+15y+30z+22w+11, 16x+18z+5w-55 \}$. In contrast, $T_E$ cannot handle such a scale as it enumerates terms from small to large.

9.4 Exp3: Comparison of Values of $c$ and $c_0$ (RQ3)

\textbf{Procedure.} In this experiment, we test how the values of $c$ and $c_0$ affect the performance of PolyGen. Here, we compare the default implementation of PolyGen, where both $c$ and $c_0$ are set to 2, with two modified versions, PolyGen$_1$ and PolyGen$_3$, where $c$ and $c_0$ are both set to 1 and 3 respectively.

\textbf{Results.} The results are summarized in Table 4. These results demonstrate that the generalizability and efficiency of PolyGen are not sensitive to the values of $c$ and $c_0$. Note that though PolyGen$_1$ and PolyGen$_3$ perform faster on those jointly solved benchmarks than PolyGen, they solve fewer benchmarks in contrast.

10 CONCLUSION

In this paper, we adopt a concept from computational learning theory, Occam Learning, to study the generalizability of the STUN framework. On the theoretical side, we provide a sufficient set of conditions for individual components in STUN to form an Occam solver and prove that Eusolver, a state-of-the-art STUN solver, is not an Occam solver. Besides, we design an Occam solver PolyGen for the STUN framework. On the practical side, we instantiate PolyGen on the domains of CLIA and evaluate it against state-of-the-art PBE solvers on 100 benchmarks and 2 common oracle models. The evaluation shows that (1) PolyGen significantly outperforms existing STUN solvers on both efficiency and generalizability, and (2) PolyGen keeps almost the same generalizability with those solvers that always synthesize the smallest program, but achieves significantly better efficiency.

ACKNOWLEDGEMENT

We sincerely thank the anonymous OOPSLA reviewers for their valuable feedback on this work. This work is supported in part by National Key Research and Development Program of China No. 2019YFE0198100 and National Natural Science Foundation of China under Grant No. 61922003.

REFERENCES

David J. Aldous and Umesh V. Vazirani. 1995. A Markovian Extension of Valiant’s Learning Model. \textit{Inf. Comput.} 117, 2 (1995), 181–186. https://doi.org/10.1006/inco.1995.1037

Proc. ACM Program. Lang., Vol. 5, No. OOPSLA, Article 167. Publication date: October 2021.
