SynGuar: Guaranteeing Generalization in Programming by Example

Bo Wang
National University of Singapore
Singapore
bo_wang@u.nus.edu

Teodora Baluta
National University of Singapore
Singapore
teodora.baluta@u.nus.edu

Aashish Kolluri
National University of Singapore
Singapore
e0321280@u.nus.edu

Prateek Saxena
National University of Singapore
Singapore
prateeks@comp.nus.edu.sg

ABSTRACT

Programming by Example (PBE) is a program synthesis paradigm in which the synthesizer creates a program that matches a set of given examples. In many applications of such synthesis (e.g., program repair or reverse engineering), we are to reconstruct a program that is close to a specific target program, not merely to produce some program that satisfies the seen examples. In such settings, we wish that the synthesized program *generalizes* well, i.e., has as few errors as possible on the unobserved examples capturing the target function behavior. In this paper, we propose the first framework (called SynGuar) for PBE synthesizers that guarantees to achieve low generalization error with high probability. Our main contribution is a procedure to dynamically calculate how many additional examples suffice to theoretically guarantee generalization. We show how our techniques can be used in 2 well-known synthesis approaches: PROSE and STUN (synthesis through unification), for common string-manipulation program benchmarks. We find that often a few hundred examples suffice to provably bound generalization error below 5% with high (≥ 98%) probability on these benchmarks. Further, we confirm this empirically: SynGuar significantly improves the accuracy of existing synthesizers in generating the right target programs. But with fewer examples chosen arbitrarily, the same baseline synthesizers (without SynGuar) *overfit* and lose accuracy.

CCS CONCEPTS

• **Software and its engineering** → **Software notations and tools**; **General programming languages**.

KEYWORDS

Program Synthesis, Generalization, Sample Complexity

ACM Reference Format:

Bo Wang, Teodora Baluta, Aashish Kolluri, and Prateek Saxena. 2021. SynGuar: Guaranteeing Generalization in Programming by Example. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468621

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468621

1 INTRODUCTION

Program synthesis is the goal of automatically generating computer programs for a given task. This vision has existed for over at least four decades [52, 56, 57]. One of the mainstream approaches towards this goal is programming by example (or PBE) [27]. In its simplest form, a PBE synthesizer is given access to an *oracle* that can generate correct input-output (I/O) examples for the unknown *target* program. The synthesizer has to create a candidate program as close as possible to the target program from a pre-specified *hypothesis space*, i.e., the space of all possible candidate programs that the synthesizer can reason about. The number of given I/O examples can vary depending on the end application, but the fewer the better. Therein lies the challenge of *generalization*: If examples are too few, then many possible candidate functions satisfy them, and picking one arbitrarily might yield a solution that works well only on the seen examples. In other words, the solution overfits to the seen examples and may not generalize well. How does a synthesizer create programs that are provably close to the target program? This has been a fundamental question for PBE-based program synthesis.

There are several domain-specific solutions to generalization. In program repair as well as in inductive synthesis, for example, inferring additional specifications from observed examples that must be satisfied by the program is shown to help with generalization [6, 24, 30, 33]. Allowing the synthesizer to use more powerful oracles that adaptively craft examples or logical invariants help to synthesize correct programs [22, 31]. In neural-guided program synthesis [13, 17, 45, 54], machine learning techniques to avoid over-fitting such as regularization or structural risk minimization, are employed implicitly [59]. In domains where we have prior knowledge about the likely distribution to which the target program belongs, synthesizers can rank solutions [55], guide program search [33], and use generative models for program representation or distributional priors [21]. Some synthesizers favor short programs as per Occam's razor [25].

All of the above approaches, while useful, require additional knowledge or implicit assumptions about the target program beyond that captured by the original PBE problem setup. Without such assumptions, these approaches do *not* provide any formal guarantee

that the produced program will be correct or generalize well on unseen examples. It is natural to ask: Can we guarantee generalization without making any additional domain-specific assumptions?

In this paper, we study generalization in PBE from the perspective of *sample complexity*: How many I/O examples should the synthesizer have to see to be confident that its selected solution is close to the target program? To answer this question, the PAC learning theory provides a starting point [10, 58]. A synthesizer generalizes well when the synthesized program is close to the target program with high confidence. The notion of confidence and closeness to the target program can be made formal using PAC learning theory. Specifically, the synthesized program generalizes if it will make no more than a small fraction ϵ of errors on unseen examples taken independently, with high probability (at least $1-\delta$).

Our approach works on any distribution that the I/O examples are sampled from. To formally guarantee that generalization is achieved on the distribution, we need a principled design for PBE synthesizers. Existing PBE synthesizers are not designed to provide generalization guarantees; therefore, they pick the number of I/O examples to work with in an ad-hoc fashion. For instance, they may synthesize a program after seeing only $2-4$ examples [26, 55]. This paper seeks to answer the following questions:

RQ1. How many I/O examples would a synthesizer need to see in order to provably generalize?

RQ2. Do existing synthesizers overfit with, say, $2-4$ examples?

As a conceptual contribution, we present the first principled framework, SynGuar[1], to provide generalization guarantees about the synthesized programs. We propose a procedure that computes the size of the hypothesis space *dynamically* during synthesis, which is then used to calculate the sample size required to provably generalize. The challenge is therefore two-fold. First, while efficiently computing the size of the hypothesis space is easy in some existing PBE synthesizers (such as the PROSE framework [46]), for others it requires careful design. An example of the latter is the synthesis through unification (STUN) [5] approach. As our main technical contribution, we present an example of integrating SynGuar into synthesizers based on PROSE and STUN frameworks. Specifically, we provide two PBE synthesizers for string manipulation programs that provably generalize, one implemented in the PROSE framework (SynGuar-PROSE) and one based on the STUN approach (SynGuar-STUN). To the best of our knowledge, no prior PBE synthesizer claims such strong generalization guarantees about the synthesized programs.

From an empirical perspective, our work provides the first experimental evidence for the number of examples sufficient to guarantee generalization in practice for simple string-manipulation tasks. We run SynGuar-PROSE and SynGuar-STUN on two benchmarks in this domain: 1) manually designed data-wrangling tasks similar to those used in FlashFill [46]; and 2) the standard SyGuS 2019 benchmark [1], respectively. We find that on their respective benchmarks, the tools produce programs which are provably within at most 5% generalization error with a modest number of examples—around 197 samples and 357 examples on average, respectively—with a high probability ($\geq$ 98%). This observation also suggests that it is unlikely for PBE synthesizers to generalize from just $2-4$ examples without using some implicit or explicit additional knowledge. We confirm this observation by running the vanilla versions, i.e., versions with SynGuar disabled, on the same benchmarks with 4 randomly chosen examples or the given seed examples in the benchmark ($2-10$ in size). We find that SynGuar-PROSE generates the correct target program for 14/16 cases from the data-wrangling task benchmark and SynGuar-STUN for 53/59 cases from the SyGuS benchmark. In contrast, the vanilla versions generate correct programs for 0/16 and 36/59 cases, respectively. This shows that without enough examples, synthesized programs often overfit.

Though we focus on string-manipulating programs, our approach makes minimal additional assumptions and thus can be extended to other application domains, such as program repair [24], invariant discovery [9] and so on. The generalization guarantee fits well in applications such as data cleaning and transformation [16], where a provable accuracy matters, or automatic stub writing in symbolic execution [38, 53], where the goal is to learn a symbolic constraint that is *approximately* close enough to the target. Our experiments suggest that the sample size to achieve generalization is task and benchmark dependent, which leaves the question of how well our presented approach works in other domains or benchmarks open. These are grounds for promising future work.

[1]The tool is available with DOI number 10.5281/zenodo.4883273. The latest version can be found at https://github.com/HALOCORE/SynGuar

```
language StrPROSE;
@input string x;
@start string program := recTerm;
string recTerm := catTerm | Concat(catTerm, recTerm);
string catTerm := ConstStr(cs) | convTerm;
string convTerm := term | UpperCase(term) | LowerCase(
    term);
string term := SubStr(x, pos1, pos2);
int? pos := AbsPos(x, ka) | RegPos(x, kr);
string cs; //constant string
int ka; //absolute position
Tuple<Regex,Regex,int,int> kr; // kr=(r1,r2,k,offset)
```

Figure 1: A DSL for string transformation programs. `Concat` returns the string produced by concatenating two strings `catTerm` and `recTerm`, `SubStr` returns the substring between `pos1` and `pos2`. The `UpperCase` and `LowerCase` return the string in upper case and lower case, respectively. `AbsPos` returns the absolute position of string `x`. The `RegPos` operator outputs the k^{th} position plus an `offset` where the boundaries of the strings returned by applying regular expressions `r1` and `r2`, respectively, on string `x` match.

2 OVERVIEW

In practice, it is hard to know how many I/O examples suffice to solve a synthesis task. The number of I/O examples across various target programs, even for the same synthesizer, vary in prior works and are chosen somewhat arbitrarily. For instance, the SyGuS benchmark [1] has a dedicated track for the domain of string-manipulating programs. The benchmark consists of several PBE tasks and each of them is provided with a different number of I/O examples. Some have as low as 2 examples whereas others have 50.

```
// word \w+=(A-Za-z0-9)+, digit \d+=(0-9)+
// *.?= any character
Concat( // return substr until the end of 1st word match
 SubStr(x, AbsPos(x, 0), RegPos(x, (\w+, .*?, 0, 0))),
 Concat(ConstStr(","), // append ","
  Concat( // return substr of 2nd word match until the end
       of the word
   SubStr(x, RegPos(x, (.*?, \w+, 1, 0)),
             RegPos(x, (\w+, .*?, 1, 0))),
   Concat(ConstStr(","),
    Concat( // return the substr of 3rd word match until
         the end of the word
     SubStr(x, RegPos(x, (.*?, \w+, 2, 0)),
               RegPos(x, (\w+, .*?, 2, 0))),
     Concat(ConstStr(","),
      // return the substr from the 4th word until the end
           of the string
       SubStr(x, RegPos(x, (.*?, \w+, 3, 0)),
                 AbsPos(x, -1))))))))
```

```
Concat( // return substr of first two characters
 SubStr(x, AbsPos(x, 0), AbsPos(x, 2)), overfits
 Concat(ConstStr(","),
  Concat( // return the string from the start of the first
       number offset by 1 till the second last separator
   SubStr(x, RegPos(x, (.*?, (-?\d+)(\.\d+)?, 0, 1)),
             RegPos(x, (.*?, [\,\.\;\-\|], -2, 0))), overfits
   Concat(ConstStr(","),
    Concat( // return the substr from second capital letter
         offset +2 to offset +3
     SubStr(x, RegPos(x, (.*?, [A-Z]+, 1, 2)),
               RegPos(x, (.*?, [A-Z]+, 1, 3))), overfits
     Concat(ConstStr(","),
      // return the substr from start of second word with
           offset +3 till the end
       SubStr(x, RegPos(x, (\w+, .*?, 1, 3)),
                 AbsPos(x, -1)))))))) overfits
```

Figure 2: On the left, we show the correct target program t that tokenizes the string x on the boundaries of the first 4 words. On the right, we show the program synthesized on 2 examples which overfits at all the four SubStr operations.

#Ex.	Input	Output	#f
1	"0E-E 2|u7kuZ85"	"0E,E,2,u7kuZ85"	$\approx 10^{42}$
2	" J-3bJ.9;PPm"	" J,3bJ,9,PPm"	$\approx 10^{20}$
3	"tpJ|AV n0d7 6z"	"tpJ,AV,n0d7,6z"	$\approx 10^{12}$
4	"R 3|6VCs Q"	"R,3,6VCs,Q"	304128
5	"M x cSkrw ru6"	"M,x,cSkrw,ru6"	304128
6	"Wk U U nZp X "	"Wk,U,U,nZp X "	864
7	"gsa-ub hn lpa"	"gsa,ub,hn,lpa"	216
8	"RO8I3 R SuM|e "	"RO8I3,R,SuM,e "	144
9	"q E 0 LD0 "	"q,E,0,LD0 "	144
10	"dZPz T.Q s "	"dZPz,T,Q,s "	144
11	"Ny e87e -lO 0w"	"Ny,e87e,lO,0w"	36
12	"FX 1 U P 1fN"	"FX,1,U,P 1fN"	18

Figure 3: The I/O examples provided to the synthesis algorithm, along with the number of consistent candidates (#f). This number drops from $\approx 10^{42}$ to 18 in just 12 samples.

To illustrate the problem of overfitting in PBE-based synthesis, let us consider a data-wrangling task of tokenizing a given passage of text into individual words. The tokenization task involves recognizing the boundaries of the first 4 words and replacing the characters used to separate them with commas. The user wishes to use a program synthesis tool to learn a program that performs this task. Here, we use our tool SynGuar-PROSE which is based on the PROSE framework [46] for synthesizing a program. Program synthesizers output programs in the syntax of some pre-specified *target language* or domain-specific language (DSL). In order to support our task, we implement a string transformation DSL in PROSE that is similar to the FlashFill DSL [46], see Figure 1. We give a reference implementation of our modified DSL in the supplementary material [2]. The user provides an oracle, which can be queried by the synthesizer for I/O examples exemplifying the behavior of the target program. Each I/O example is a pair of strings formed from lower and upper case letters, digits, spaces and separators.

Given the problem setup as described above, the goal of the synthesizer reduces to learning the 4 correct substrings from the provided examples. Let us examine how well the synthesizer performs on a few I/O examples, say 2. After running it on the first 2 examples given in Figure 3, as expected, the output program overfits. For instance, instead of trying to get the first substring until the end of the first word, it just picks two characters for the first word since both the examples have only two characters until their respective first separators. In fact, the synthesizer continues to overfit even after being provided 9 examples.

However, it turns out that for our running example, we can theoretically assert a program close to the target will be picked with high probability after 149 examples! Our proposed algorithmic framework SynGuar is able to calculate this quantity on the fly and stop when enough examples are seen. To understand how it works, consider Figure 3 that shows the estimated size[2] of the hypothesis space that is *consistent* (or matches) with all the seen examples up to a certain point. SynGuar computes this quantity internally, which serves as our main technical insight. Notice that the space of the consistent programs shrinks progressively as more examples are provided. Before seeing any example, the hypothesis space is the set of all the programs our target language can represent and its size can be infinite as the DSL grammar is recursive. Now suppose the synthesizer sees the first I/O example, then the number of candidate programs which are *consistent* with the first example reduces considerably. The reduction depends critically on the example provided. For instance, the first I/O example shown in Figure 3 will reduce the space of consistent programs to 10^{42}. This is still quite large—if we arbitrarily pick one program, without using any auxiliary assumptions or prior knowledge about the target program, the odds of picking the correct program are negligibly small. However, after seeing 12 examples, the consistent program space reduces to

[2]A sound upper bound of the actual size is calculated.

Algorithm 1 Meta-synthesis Algorithm

```
1:  procedure MetaSyn(ϵ, δ)
2:      while stopping condition do
3:          Query user for k examples
4:          Update the hypothesis space
5:          return None if empty hypothesis space
6:      Compute the m examples for (ϵ, δ) guarantee
7:      Query user for m examples
8:      Update the hypothesis space
9:      if empty hypothesis space then
10:         return None
11:     return f in hypothesis space
```

18 for our running example. Note that choosing a program out of these 18 at random does not have any guarantees on its closeness to the target program. For a program space of 18, to choose a program that is close to the target program with provably high confidence we require 137 more examples. SynGuar can provably assert that it has seen enough examples to stop and return a program close to the target program after 12 + 137 = 149 examples.

Problem Setup. Similar to the setup used in existing PBE-based synthesizers, we are given an oracle to query I/O examples and a DSL for representing the output program. Additionally, we are given user-specified (ϵ, δ) parameters that capture the desired generalization guarantee. The synthesizer queries the oracle for as many I/O examples as it needs and terminates with either *None* or a synthesized function f. The probability that the synthesizer returns a function f that might not generalize should be under the given small δ. Let $S = \{(x,\ t(x))\}$ be the set of I/O examples seen by one invocation of the synthesizer. Here, each x is an input drawn independently and identically distributed (i.i.d) from an unknown distribution D that the oracle captures. We assume that f will satisfy all given I/O examples, $\forall x \in S, f(x) = t(x)$. Note that this is different from the "best-effort" [42] or approximate synthesis approaches [53] where the program f is allowed to differ from t on some examples in S. In this setup, therefore, S is a random variable. The synthesizer, denoted as $\mathcal{A}(S)$, is also a random variable defined over I/O samples (S) drawn from D.

We seek to design PBE synthesizers that achieve generalization given by a rigorous PAC-style guarantee [58] while computing the required sample complexity. For an (ϵ, δ)-synthesizer, the generalization error $error(f) = Pr_{x\sim D}[f(x) \neq t(x)]$ is bounded by ϵ. The probability of generating f with $error(f) > \epsilon$ is bounded by δ.

Definition 2.1 ((ϵ, δ)-synthesizer). A synthesis algorithm $\mathcal{A}$ with hypothesis space H is an (ϵ, δ)-synthesizer with respect to a target class of functions C iff for any input distributions D, for all $t \in C$, $\epsilon \in (0, 1)$, $\delta \in (0, 1)$, given example set S drawn i.i.d from the D,

$$Pr[\mathcal{A}(S) \text{ outputs } f \in H \text{ such that } error(f) > \epsilon] < \delta$$

3 THE SYNGUAR FRAMEWORK

We propose a framework with a similar algorithmic meta-structure as that of existing PBE engines. The overall procedure is shown in Algorithm 1. Instead of synthesizing a program after seeing a pre-determined number of I/O examples the procedure queries an oracle for new examples until it synthesizes a program that generalizes. There are two key new features in our framework: a stopping criterion and a dynamically calculated count of the number of samples to be seen. The following classical result gives us a starting point to compute the count precisely.

A Starting Point. The number of examples provably sufficient to achieve the (ϵ, δ)-generalization is given by Blumer et al. [10]. We restate this result, which computes sample complexity as a function of (ϵ, δ) and the capacity (or size) of any given hypothesis space H.

Theorem 3.1 (Sample Complexity for (ϵ, δ)-synthesis). *For all $\epsilon \in (0, 1)$, $\delta \in (0, 1)$, and hypothesis space H, a synthesis algorithm $\mathcal{A}(S)$ which outputs functions consistent with m i.i.d samples is an (ϵ, δ)-synthesizer, if*

$$m > \frac{1}{\epsilon}(\ln |H| + \ln \frac{1}{\delta})$$

The above theorem is intuitively based on the following analysis. Let us say we have some initial hypothesis space H. After seeing one new I/O example, each hypothesis that is "ϵ-far" from t (generalization error $> \epsilon$) becomes inconsistent with some non-zero probability, and is eliminated. Therefore after seeing sufficiently many new examples, the probability of any "ϵ-far" hypothesis being output falls below δ. For details, please see the analysis [10].

3.1 Key Observations & Challenges

We observe that Theorem 3.1 can be used at any point of the synthesis procedure. After seeing say the first S examples, let the space of programs consistent with the examples be H_S. We can plug $|H_S|$ into Theorem 3.1 to compute how many more examples are sufficient to achieve the guarantee provided in Definition 2.1. But, there are several key technical challenges in utilizing the classical result of Theorem 3.1 in providing end-to-end generalization guarantees.

First, applying this result requires being able to compute $|H_S|$. We point out that this has not been an explicit algorithmic goal when designing existing synthesizers. Consequently, computing $|H_S|$ is non-trivial in some of the existing synthesizers. To tackle this, we design our own PBE synthesizer based on the STUN approach and bottom-up explicit search with the ability to compute $|H_S|$ (see Section 4.2). Further, we show how to integrate SynGuar in PROSE, a synthesis framework where the size of the hypothesis space can be easily computed.

Second, $|H_S|$ can be large and plugging in its values at the beginning leads to vacuously high sample bounds in practice. For instance, initially the hypothesis size in our running task is infinite, and even after seeing one example, the size is 10^{42}. Therefore, instead of naively plugging in values of parameters at the beginning, we use the idea that if $|H_S|$ decreases as the synthesizer sees more examples then the estimated sample complexity for generalization reduces as well. Therefore, in SynGuar's design, $|H_S|$ is computed on the fly as the synthesizer sees more examples and the stopping condition ensures that the synthesized program generalizes.

Lastly, the PAC learning theory offers no recourse to predict how fast $|H_S|$ reduces with more samples in practice. This question, then, becomes an empirical one: *For which programs do we observe that a*

Algorithm 2 SynGuar Synthesis returns a program with error smaller than ϵ with probability higher than $1-\delta$

```
1:  procedure SynGuar(ε, δ)
2:      k = 1 // tunable parameter
3:      g ← PickStoppingCond
4:      S' ← ∅, s ← 0
5:      size_H ← ComputeSize(H)
6:      n ← g(size_H)
7:      while s ≤ n do
8:          S' ← S' ∪ sample(k)
9:          H_S' ← UpdateHypothesis(S')
10:         size_H_S' ← ComputeSize(H_S')
11:         return None if size_H_S' = 0
12:         s ← s + k
13:         n ← min(n, s + g(size_H_S'))
14:     m_H_S' = (1/ε)(ln size_H_S' + ln (1/δ))
15:     T ← sample(m_H_S'(ε, δ))
16:     S ← S' ∪ T
17:     return program f in H_S or None if H_S = ∅
```

small number of examples are sufficient to generalize? Our empirical evaluation shows that for several common string manipulation tasks, the required number of examples turn out to be modest.

Remark. It can be seen that our solution is a modification to the existing PBE synthesis loop, which can be instantiated for several program synthesis engines. Our proofs and analysis utilize classical sample complexity arguments, together with bounds for hypothesis space size. It may therefore appear surprising then that such calculations are not routine in prior program synthesis works already, despite accuracy / generalization being a natural objective. We believe that the challenges stated above offer an explanation as to why applying previous theoretical results is not straightforward, and requires a principled approach. We point out that the subtlety is in our problem formulation itself, namely, the use of dynamic calculation of the remaining sample size needed for generalization.

3.2 SynGuar Algorithm

We start by addressing the second challenge assuming that $|H_S|$ is computable. Recall that, our synthesis algorithm takes as input the error tolerance parameters ϵ and the confidence δ (Algorithm 2). The algorithm follows the structure of Algorithm 1 and consists of two phases: the sampling phase (lines 7 – 13) and the validation phase (lines 14 – 17). The sampling phase addresses the second challenge by trying to shrink the hypothesis space as much as possible. In each iteration k samples are taken before updating the hypothesis space that satisfies all k samples seen. The crucial part of the sampling phase is deciding when the number of examples seen so far (stored in the set S' and whose cardinality is s) is "enough". SynGuar stops sampling when the number of examples seen so far exceeds a stopping threshold, represented by variable n. In each iteration, the stopping threshold (which depends on finite $|H|$ initially) either remains the same or it gradually shrinks with each update of the hypothesis space (line 13). Hence, SynGuar *dynamically* updates the threshold based on the change in the hypothesis size. To control how much the threshold variable shrinks with respect to the size of the updated hypothesis space $H_{S'}$, SynGuar picks a function g (line 3). Our framework allows using any $g : \mathbb{N} \mapsto \mathbb{Z}$ that is monotonically non-decreasing, and we provide a sample complexity analysis for such functions. We propose a particular choice of function g as the default: $g(x) = \max\{0,\ \frac{1}{\epsilon}(\ln(x) - \ln(\frac{1}{\delta}))\}$. This g has a useful property—the required number of samples it entails in the worst case cannot be more than twice the number of samples that an optimal choice of g will take. We will formally state and prove this optimality claim in Section 3.3.

In the second phase, in addition to the samples S', SynGuar samples a fixed number of samples according to Theorem 3.1. SynGuar then can return a program f with provable (ϵ, δ) guarantees (line 14). Algorithm 2 calls sub-procedures UpdateHypothesis (line 9) to find a program space consistent with S' and ComputeSize (line 10) to compute the size of the consistent program space. The sub-procedure UpdateHypothesis can be implemented by any existing PBE synthesis algorithm which return hypotheses consistent with S'. Section 4 details how to implement ComputeSize, which is specific to the underlying UpdateHypothesis sub-procedure.

Running Example. Consider the example given in Section 2. First the user inputs $\epsilon = 5\%, \delta = 2\%$ respectively. In the sampling phase, the user is queried for one example in each iteration. After the first iteration, i.e., seeing one example, the sample size for generalization ($m_{H_{S'}}$) is 2018. Instead, SynGuar's sampling phase stops after seeing $n = 12$ examples and the additional sample size for generalization (Line 14) reduces to $m_{H_{S'}} = 137$. The total sample size of both phases sums up to 149 examples, which is 10× less than the sample complexity after the first iteration. This is a direct consequence of SynGuar dynamically estimating the sample size for generalization.

3.3 Analysis of the Algorithm

SynGuar's design is motivated by being able to give a formal generalization guarantee and a bounded sample complexity. For this purpose, we state and prove the following properties:

(P1: Termination) SynGuar always terminates for a finite $|H|$.

(P2: (ϵ, δ) guarantees) The probability of SynGuar returning an f that is ϵ-far is smaller than δ.

(P3: Sample complexity) SynGuar's sample complexity is always within 2× of the optimal for $k \leq \frac{1}{2\epsilon} \ln \frac{1}{\delta}$.

Theorem 3.2 (P1). SynGuar *always terminates for a finite* $|H|$.

Proof. It suffices to prove that the sampling phase (lines 7 – 13) of SynGuar terminates in order to show that SynGuar terminates. In each iteration of the sampling phase, let S_i be the queue storing the user-provided examples after each iteration, z_t be the t^{th} example, $S_{i+1} = S_i \cup \{z_{ik+1}, \ldots z_{ik+k}\}$ and $S_0 = \varnothing$. For each S_i, H_{S_i} determines the set of consistent hypothesis that satisfy S_i. Let N_i be the limit of the number of I/O examples n for the sampling phase after iteration i. For iterations i and j where $i < j$ and $\forall g : \mathbb{N} \to \mathbb{Z}$ such that g is monotonically non-decreasing, the following holds:

$$S_i \subset S_j \Rightarrow |H_{S_j}| \leq |H_{S_i}| \Rightarrow g(|H_{S_j}|) \leq g(|H_{S_i}|)$$

$$N_j \leq \min\{N_i, |S_j| + g(|H_{S_j}|)\} \leq N_i \text{ (see line 13 in Alg. 2)}$$

Therefore, if $N_0 \leq g(|H|)$ then the loop will terminate at some iteration p such that $N_p < |S_p| \leq N_p + k \leq N_0 + k$. □

Theorem 3.3 (P2). *The probability of* SynGuar *returning a synthesized program f that is ϵ-far is smaller than δ.*

Proof. By Theorem 3.2, we know that the sampling phase terminates with S' samples (see line 14). In lines 14 − 16 SynGuar samples an additional number of I/O examples required to generalize and then synthesizes a program after seeing the additional samples. Therefore, Theorem 3.3 follows from Theorem 3.1. □

In order to prove the last property, we define a new quantity $\omega(Q)$. It is the smallest sample size taken by SynGuar (ϵ, δ) for any non-decreasing g used for a sequence of I/O examples Q.

Definition 3.4 (Smallest dynamic sample size). For any infinite sampled sequence of examples Q, let Prefix(Q, g) be the prefix of Q at which SynGuar (ϵ, δ) terminates. Then,

$$\omega(Q) = \inf\{|m_g| \ : \ \forall g,\ m_g = \textsc{Prefix}(Q, g)\}$$

Theorem 3.5 (P3). SynGuar *uses no more than* $2\omega(Q)$ *examples on any Q when the result is not None with* $g(x) = \max\{0,\ \frac{1}{\epsilon}(\ln(x) - \ln(\frac{1}{\delta}))\}$ *and* $k \leq \frac{1}{2\epsilon}\ln\frac{1}{\delta}$.

Due to space limit, we provide the proof of this theorem in the supplementary material [2].

4 RETROFITTING SYNGUAR INTO EXISTING SYNTHESIZERS

We now show how to compute $|H_S|$, the size of the consistent program space. A sound upper bound of $|H_S|$ is safe to use, since in this case, our analysis shows SynGuar to take more examples than those needed to guarantee generalization. We show how to compute $|H_S|$ bounds for two well-known PBE synthesis approaches.

4.1 SynGuar for the PROSE Framework

We first apply SynGuar on top of the PROSE framework [46], a state-of-the-art PBE meta-synthesis framework that generates an inductive synthesizer for a given DSL. PROSE allows developers to write DSLs and specify *witness functions* that capture (a subset of) inverse semantics for the DSL operators. These witness functions are the drivers for the "deductive backpropagation" because they specify the inputs or properties of the input given an I/O example.

We implement a synthesizer named StrPROSE with the DSL in Figure 1 on top of PROSE by specifying executable semantics and witness functions for its operators. Our DSL shares most operators with the DSL of FlashFill [46]. For the operators that differ, we detail their executable semantics and witness functions in the supplementary material [2]. Note that SynGuar works with any DSL expressible in PROSE, as long as each operator has its semantics and an associated witness function specified.

PROSE uses an internal succinct representation of the program space using a data structure called version-space algebra (VSA) which makes it convenient to calculate $|H_S|$ [26, 34, 35, 39]. A VSA is a directed graph where each node corresponds to a set of programs. The leaf nodes explicitly represent a set of programs that can be enumerated. There are two types of internal nodes: *union* nodes that represent a set-theoretic union and *join* nodes that represent k-ary operators which are defined by the DSL.

Computing $|H_S|$ Using VSA. PROSE readily computes $|H_S|$ using a bottom-up graph traversal on its VSA. For each leaf node, it enumerates and counts the set of programs directly. For every union node, to compute the corresponding number of programs it adds up the count of all child nodes. For every join node, the number of programs is a cross product of all applications of the k-ary operator to k parameter programs. This soundly upper bounds $|H_S|$. In our implementation, we reuse the Size API available in PROSE, resulting in the sizes shown in Figure 2.

Scaling to Large Sample Size. Building VSA on a large number of examples can be time-consuming. Therefore, we build the VSA on a subset of the examples which lead to the same set of programs. More specifically, we take the examples one by one and drop the examples that do not decrease the VSA size.

4.2 SynGuar in StrSTUN

STUN is a well-known synthesis approach [5]. It was originally proposed as an extension of the counter-example guided inductive synthesis (CEGIS) approach to synthesize program from the specification. The high-level idea is to synthesize partial solutions satisfying parts of the inputs and unify them. As an instantiation of STUN for synthesizing conditional programs under PBE settings, we work with top-level[3] if-then-else unification operator where the condition can be any boolean expression in the hypothesis space. The subsequent synthesis algorithms following this approach do *not* compute $|H_S|$ directly, or make it straightforward to compute it. We design a synthesis algorithm based on this approach, and a procedure to soundly compute an upper bound on $|H_S|$. We choose our target language as the SyGuS string-manipulating program DSL [1]. Our synthesis algorithm is referred to as StrSTUN.

Vanilla StrSTUN: Overview. StrSTUN instantiates the previously proposed approach of bottom-up explicit search with observational equivalence reduction [3]. Its algorithm consists of two phases at a high level: an enumeration phase and a unification phase. In the first phase, the synthesizer enumerates candidate programs only by repeatedly using function application. It clusters all candidate programs which have the same I/O behavior on the given examples and saves only one program representative of each cluster. Such enumerated programs may only be consistent with subsets of the given I/O examples. In the unification phase, StrSTUN composes enumerated programs with an if-then-else unification operator. The final synthesized program P, therefore, can be a straight-line program (obtained by repeated function application) or a program with nested compositions of the form `if` P_1 `then` P_2 `else` P_3, where P_2 and P_3 can be nested programs themselves. The nesting depth is bounded internally to limit the search space. We explain the constructional details of these phases next, and explain how to compute $|H_S|$ from the internal data-structures later. In what follows, we denote inputs and outputs of the given I/O examples as vectors **w** and **o** respectively.

Vanilla StrSTUN: Enumeration. StrSTUN enumerates all candidate programs in a bottom-up fashion by generating programs through function application. We start with the smallest syntactic programs, which are just single components (or syntactic terminals)

[3] Such if-else constructs is restricted to being at the top of the function's AST.

in the target language, working up to programs with more than one component. For instance, `concat(input0, input1)` is a candidate program with three components. The total hypothesis space without conditionals is fixed based on a user-provided constraint on the maximum component size, specified as the maximum number of components the straight-line program contains. For each program created in the enumeration phase, we compute the outputs of the program on the given I/O examples. Note that this step does not require explicitly individually creating and running all candidate programs—it is possible to evaluate outputs during the bottom-up construction of the programs [3].

We compute 2 useful data structures internally during enumeration. The first is a *consistency vector* $\mathbf{c}$ which captures whether an enumerated program P is consistent with the given I/O examples represented by vector $\mathbf{w}$ and vector $\mathbf{o}$. Specifically, the consistency vector $\mathbf{c}$ for program P has the i^{th} element set to ✓ if the $P(\mathbf{w}[i]) = \mathbf{o}[i]$, namely, the output of P on the i^{th} given input example matches the corresponding given output example. Otherwise, $\mathbf{c}[i]$ is set to '×'. This data structure speeds up the search in two ways, conceptually. First, it is calculated on top of *observational equivalence* [3]. If many candidate programs generate the same outputs on the given input examples, then they are all observationally equivalent, and we only need to keep one such program that has the outputs for completeness [3], thus the consistency vector is only calculated once for those programs. Secondly, even if two programs are observationally not equivalent, and both give different incorrect outputs for an input example, they will have the same value (×) in their consistency vectors. Thus, we can effectively cluster many programs that are observationally non-equivalent but have the same consistency vector to speed up the next phase.

The second useful data structure is cluster map ϕ. It maps consistency vectors to sets of programs. Each distinct consistency vector $\mathbf{c}$ computed during the enumeration phase is mapped to a set of programs $\phi(\mathbf{c})$ that have I/O behavior captured by $\mathbf{c}$.

Vanilla StrSTUN: Unification. In this phase, StrSTUN synthesizes programs with nested if-then-else structures. The goal is to create programs that are consistent with larger subsets of I/O examples than enumerated programs, and ideally, correct on the full set of I/O examples. The cluster map ϕ allows us to quickly find programs which match certain subsets of all the given I/O examples, specified by a consistency vector value. When a program $P :=$ `if` P_1 `then` P_2 `else` P_3 is synthesized during unification, we must carefully construct a semantically correct consistency vector for P, using those for sub-programs P_2 and P_3. Here, note that P_1 needs to be a program that evaluates to a boolean value on a given input example, say $\mathbf{w}[i]$. If it evaluates to `true`, then the program P_2 must be correct on $\mathbf{w}[i]$ for P to be correct on $\mathbf{w}[i]$. Therefore, in this case, we mark P as consistent with $\mathbf{w}[i]$ if and only if $\mathbf{c}[i]$ for P_2 has a ✓. Analogously, if P_1 evaluates to `false` on $\mathbf{w}[i]$, then $\mathbf{c}[i]$ for P_3 should be set to ✓ for P to be marked consistent with $\mathbf{w}[i]$; otherwise P is marked inconsistent with $\mathbf{w}[i]$.

The above-described unification procedure synthesizes all programs with a nesting depth of up to a pre-configured maximum (default of 2). The nesting depth controls the hypothesis space desired by the user. The cluster map ϕ is updated continuously with new consistency vectors discovered in the unification phase. A successful solution is a program that matches all given examples, i.e., has a consistency vector with a ✓ for all values.

Computing the $|H_S|$. The vanilla StrSTUN algorithm can be slightly modified and augmented with rules shown in Table 1 to compute the $|H_S|$ soundly. Notice that in vanilla StrSTUN, when employing the observational equivalence, a program with larger components size might be discarded if there is a smaller program that has the same *value vector* [4] [3]. But we need to count all programs at different components sizes. To do so, we store multiple counting values (and representative programs) for different component sizes along with each value vector.

During the enumeration phase, programs are synthesized bottom-up from smallest components size to larger ones. Let t be the components size of a program. We keep track of a $\texttt{Count}(\mathbf{v}, t)$ for each value vector $\mathbf{v}$ computed for programs with size t. The $\texttt{Count}(\mathbf{v}, t)$ for $t = 1$ (smallest base components) can be directly enumerated (rule 1 in Table 1), since these are program input arguments or constant components in our target language. For $t > 1$, $\texttt{Count}(\mathbf{v}, t)$ can return 0 if there is no enumerated program with components size t that outputs value vector $\mathbf{v}$. Same value vectors $\mathbf{v}$ may have different counts for different components sizes t, thus we enumerate on tuple $(\mathbf{v}, t)$ rather than just $\mathbf{v}$. When StrSTUN uses function application to generate a new program P' from programs P_i (rule 2.1 in Table 1), the count for the value vector of the resulting P' is updated by adding the product of all the counts of its arguments P_i at their respective components sizes (rule 2.2 in Table 1). This completes all the ways programs are compositionally created in the enumeration phase from component size 1 to the maximum component size.

After the enumeration on value vectors is finished, we have also clustered observationally non-equivalent programs based on a consistency vector $\mathbf{c}$ in ϕ. Define $\psi[\mathbf{c}]$ as the set of all the value vectors that corresponds to a consistency vector $\mathbf{c}$, we sum up the $\texttt{Count}(\mathbf{v}, t)$ for every $\mathbf{v}$ in $\psi[\mathbf{c}]$ (rule 3 in Table 1). This way, we compute counts for consistency vectors.

During unification, programs of the form $P :=$ `if` P_1 `then` P_2 `else` P_3 are composed. Here, counts of the consistency vectors of P_2 and P_3 have been computed after enumeration if P_2 and P_3 are programs with nesting depth zero. In this case, the $\texttt{Count}(\mathbf{c}, 1)$ of the program P is the the product of P_1, all possible P_2 (0 condition) with P_1 as the condition, and all possible P_3 (0 condition) with P_1 as the condition, summed over all possible P_1. Thus, we have computed `Count` for the consistency vectors of programs with nesting depth of 1. Using this, we can recursively compute counts for consistency vectors of programs with nesting depth 2 or more (rule 5 in Table 1). To optimize, we memorize counts of individual consistency vectors as well as for sets of consistency vectors. For example, the set comprising two consistency vectors $\langle ✓, ×, × \rangle$ and $\langle ✓, ×, ✓ \rangle$ is succinctly represented as $\langle ✓, ×, \top \rangle$, and their sum of counts is memorized (rule 4 in Table 1). We prove that the rules in Table 1 provide a sound upper bound of the size of the hypothesis space $|H_S|$ in supplementary material [2].

[4] The value vector $\mathbf{v}$ for a program P is simply the outputs of P on the given input examples, i.e., $\mathbf{v}[i] := P(\mathbf{w}[i])$ for all i.

Table 1: Count rules for StrSTUN. We define the set of all boolean value vectors as $\mathcal{B}$, and we use C to represent a succinct representation of consistency vectors, which can be a singleton or a set of consistency vectors. The Count value is calculated differently for value vector v, consistency vector c, or their succinct representation C.

Count Rules for Enumeration
Starting point of enumeration:
$\text{Count}(\mathbf{v}, 1)$ = number of single components that output $\mathbf{v}$
$\text{Count}(\mathbf{v}, t) = 0$, for $t > 1$. (1)

When program P' with value vector $\mathbf{v}'$ is enumerated at component size t':
Let, $P' = f(P_1, P_2, ..., P_n)$ for some function component f and programs P_i.
If t_i is the component size for P_i,
and $\mathbf{v}_i$ is the value vector of (P_i, t_i), $i \in \{1, ..., n\}$,
then $1 + \sum t_i = t'$ holds, and (2.1)

$$\text{Count}(\mathbf{v}', t') \leftarrow \text{Count}(\mathbf{v}', t') + \prod_{i=1}^{n} \text{Count}(\mathbf{v}_i, t_i) \quad (2.2)$$

Count Rules for Clustering

$$\text{Count}(c) = \sum_{\mathbf{v} \in \psi[c]} \sum_t (\text{Count}(\mathbf{v}, t)) \quad (3)$$

Count Rules for Unification
$C_{\text{goal}} = \langle \checkmark, \checkmark, ..., \checkmark \rangle$,
the count of hypothesis space up to k conditions is $\sum_{i=0}^{k} \text{Count}(C_{goal}, i)$

$$\text{Count}(C, 0) = \sum_{c \in C} \text{Count}(c) \quad (4)$$

$$\text{Count}(C, i) = \sum_{\mathbf{b} \in \mathcal{B}} \sum_{j=0}^{i-1} (\text{Count}(\Gamma_{\text{then}}(C, \mathbf{b}), j) \times \text{Count}(\Gamma_{\text{else}}(C, \mathbf{b}), i - j - 1) \times \sum_t \text{Count}(\mathbf{b}, t)) \quad (5)$$

where $\Gamma_{\text{then}}(C, b)$ and $\Gamma_{\text{else}}(C, b)$ are also succinct representation of consistency vectors, and

$$\Gamma_{\text{then}}(C, b)_i = \begin{cases} \checkmark \text{ if } b_i = T \\ \top \text{ otherwise} \end{cases}, \Gamma_{\text{else}}(C, b)_i = \begin{cases} \checkmark \text{ if } b_i = F \\ \top \text{ otherwise} \end{cases}$$

StrSTUN **Augmented with** SynGuar**.** We call StrSTUN augmented with SynGuar as described in Section 4 as SynGuar-STUN. The maximum number of nested conditions during the unification phase of StrSTUN is 2, and without much loss of expressiveness, we only allow the `else` branch to have nesting. With this setting, the hypothesis space of StrSTUN is a union of:

- H_0 (The set of straight-line programs),
- H_1 (The programs of form `if` P_1 `then` P_2 `else` P_3),
- H_2 (The programs of form `if` P_1 `then` P_2 `else if` P_3 `then` P_4 `else` P_5).

From H_0 to H_2, the hypothesis space is increasingly expressive. SynGuar-STUN invokes the SynGuar loop with H_0 first, and if it returns None then with H_1, and so on in that order. This has the nice property that it will return f consistent with existing examples from H_i where i is the smallest possible. For correctness, each invocation with a new hypothesis H_i uses a failure probability of $\frac{\delta}{3}$, so the total failure probability is bounded by δ (union bound).

5 EVALUATION

We have shown that when an existing PBE synthesizer using SynGuar returns a synthesized program, the program generalizes, i.e., it is close to the target with high probability. Our evaluation focuses on two empirical utility goals in string-manipulation tasks:

(1) **Accuracy:** Do our theoretical generalization guarantees improve the end accuracy of existing PBE synthesizers?
(2) **Sample Size:** How many examples does SynGuar require to achieve provable generalization?

Recall that SynGuar primarily extends existing synthesizers to control how many examples the synthesizer sees before stopping. We evaluate (a) SynGuar-PROSE, which builds on the PROSE framework, and (b) SynGuar-STUN, which is implemented on the StrSTUN synthesizer we designed. The vanilla StrSTUN synthesizer is around 4000 lines of C++ code. These vanilla versions of PBE synthesizers (without the SynGuar augmentation) serve as our baselines to measure improvements due to SynGuar.

We point out that PBE synthesizers for string programs often compete on computational overheads reported for producing *any* program that fits a given set of examples. Our accuracy criterion and our objective are completely different—we want to check when a synthesizer produces a program close to a fixed target program (the number of examples is not fixed). This is why we do not compare to other baseline solvers which may be computationally faster [48, 49], but are not designed to generalize to a target program.

Benchmarks. For SynGuar-PROSE, we considered 16 common string-related programming tasks as target functions to synthesize. These are of similar style and complexity as those reported in FlashMeta paper [46] such as changing the date format, extracting numbers or abbreviating words. Henceforth, we refer to these programs as PROSE-benchmark, details of which are in the supplementary material [2]. For SynGuar-STUN, we take the euphony benchmark from the PBE-Strings track of the SyGuS 2019 benchmark [1][5] which contains 100 PBE tasks with $2 - 16$ examples.

The target programs are not available for those 100 tasks, so we manually wrote them from the given examples from the benchmark. Out of the 100 tasks, 10 are for tasks that output boolean values which are not in the scope of our considered DSL. Further, we experimentally observed that our StrSTUN implementation scales up to component size 9 (size of the longest straight-line programs before unification) within a reasonable computation of a day for all benchmarks to finish on our experimental setup (larger component size increases the program search space). So for the remaining 90, we filtered out the ones that could not be manually constructed under component size 9. This finally results in 59 SyGuS benchmarks which we call SyGuS-STUN.

The generalization error tolerance for all experiments is set to 5% ($\epsilon = 0.05$) and confidence parameter to 98% ($\delta = 0.02$) by default. When comparing sample size for different ϵ, we also run the benchmark on ($\epsilon = 0.02, \delta = 0.02$) and ($\epsilon = 0.1, \delta = 0.02$). The default step size k for the sampling phase is set to 1 for SynGuar-PROSE and 20 for SynGuar-STUN. With these parameters, it is guaranteed with probability at least 98% that when SynGuar stops, its synthesized program is going to have a generalization error of at most 5%. We ran all experiments on Amazon EC2 Ubuntu 16.04 instance with 512GB RAM, 64-core 3.1GHz Intel Xeon processors where each benchmark runs 1 core. All our experiments finished within 24 hours. For SynGuar-PROSE, 79% runs finished in 1 minute

[5] downloaded from https://github.com/SyGuS-Org/benchmarks/tree/master/comp/2019/PBE_SLIA_Track/euphony

and 100% within 1 hour. For SynGuar-STUN, 75% runs finished in 10 minutes and 97% within 2 hours.

SynGuar works with any input distribution for creating I/O examples. We choose a distribution that is easy to generate and not specialized to each target program. Specifically, we simulate a black-box fuzzer for string inputs. All our SynGuar-PROSE evaluation reports an average over 32 trials of each target program, and for each trial, we sample a string input as follows:

- the string length is chosen uniformly at random from $8-16$;
- each character in the string is either chosen uniformly at random from the character set C = `"A-Za-z0-9,.-;|"`, or chosen as white-space with probability 15× larger than the probability of any character in C.

We run each such input created on the target program to create the output. The input-output pairs are given to the synthesizer.

We evaluate SynGuar-STUN over 3 trials for each target, as these programs are computationally heavier to synthesize. For each trial, we simulate a basic mutation-based fuzzer as the sampling distribution. Specifically, we take the input strings provided in the SyGuS benchmark as seeds, and mutate them randomly as follows:

- add a string of randomly chosen length up to size 10 at a randomly chosen location of the seed string, with each character being a printable value[6]; or
- remove a randomly chosen character from the seed string.

For programs with more than one input argument, we additionally add rules specifying whether one of the inputs is a substring of another input argument. We then randomly choose a substring from that input argument when synthesizing examples. For integer inputs, we randomly chose either an integer bounded by the length of one of the string input arguments or a randomly chosen integer from (0, 1000). This ensures that the target program can be run on the mutated inputs without resulting in type errors or failure.

5.1 Accuracy Improvement

To evaluate whether our theoretical generalization guarantees translate into improved correctness, we check that the synthesized program is *correct*, i.e., syntactically or semantically equivalent to the target program. Our syntactic equivalence is confirmed automatically, and for semantic equivalence, we resort to manual inspection. While SynGuar might produce programs that are close to the target as per its ϵ-close guarantee, it is difficult to estimate closeness objectively. Therefore, we take a conservative approach and only report whether the synthesized program is correct. Programs that are "almost" or "close to" correct are reported as incorrect.

SynGuar-PROSE synthesizes 14/16 programs semantically equivalent to the target program in all 32 runs for $\epsilon = 0.05, \delta = 0.02$ which shows that SynGuar-PROSE is useful to synthesize common string manipulation programs. For one of the remaining benchmarks, the synthesized program is correct on 26/32 runs. For 1 benchmark, the synthesized program is correct on 7/32 runs. In total, SynGuar-PROSE produces correct programs in 481/512 (93.95%) runs.

We observe that the vanilla StrPROSE synthesizer (without SynGuar) consistently overfits when sample sizes are chosen arbitrarily smaller than mandated by SynGuar-PROSE. For instance, when we use exactly 4 randomly chosen examples in each trial, the vanilla StrPROSE synthesizer produces incorrect programs on most of the 32 runs for all target programs. Most of the synthesized programs overfit the examples: it is only correct on 176/512 (34.38%) runs in total. This confirms the importance of SynGuar's main objective: taking enough examples until the synthesized program is guaranteed to generalize with high probability.

SynGuar-STUN synthesizes 53/59 correct programs from the SyGuS-STUN for $\epsilon = 0.05, \delta = 0.02$ for all 3 runs. In total, this leads to 159/177 (89.83%) correct runs. As a point of comparison, the vanilla version StrSTUN synthesizer, evaluated on the examples provided in the SyGuS benchmark, produces correct programs for 36/59 of the target benchmark. SynGuar-STUN shows a 29% improvement over the vanilla StrSTUN on the SyGuS benchmark, synthesizing correctly an additional 17 programs in all trials. Further, this suggests that a significant number of SyGuS-STUN programs in the benchmark do *not* have enough examples in the benchmark to provably generalize. Synthesizers, therefore, may need additional hints or assumptions to solve them correctly.

To analyze vanilla StrSTUN under the same input distribution as SynGuar-STUN, we further evaluate it on a fixed number of randomly chosen examples in 3 trials. We use sample size of 4 per trial following prior work [26, 55]. We find that vanilla StrSTUN synthesizes only 33/59 correct programs in all 3 runs (in total correct on 121/177 runs), confirming that it often overfits.

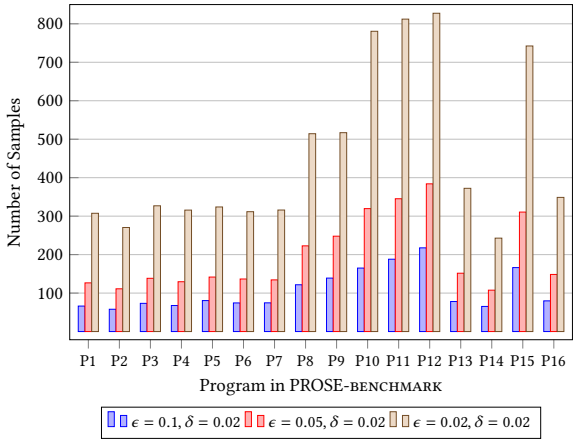

Figure 4: For most programs in the PROSE-benchmark, SynGuar-PROSE synthesizes programs with provable generalization under 400 examples for $\epsilon = 0.02, \delta = 0.02$. For $\epsilon = 0.1, \delta = 0.02$, the #samples drop to $58-218$ (average 107.22).

5.2 Sample Size Sufficient for Generalization

Our work provides empirical evidence that a modest number of examples suffice for provable generalization for the evaluated tasks. Figure 4 shows that we need between 100-400 examples (about 197 on average) to achieve ($\epsilon = 0.05, \delta = 0.02$) generalization for the SynGuar-PROSE on the 16 target programs evaluated. For a smaller error tolerance ($\epsilon = 0.02, \delta = 0.02$), the observed range of sample size becomes 200-900. Note that sample size is sensitive to ϵ as is theoretically expected—distinguishing between two functions

[6] In Python, we use `string.printable` and remove the white space characters

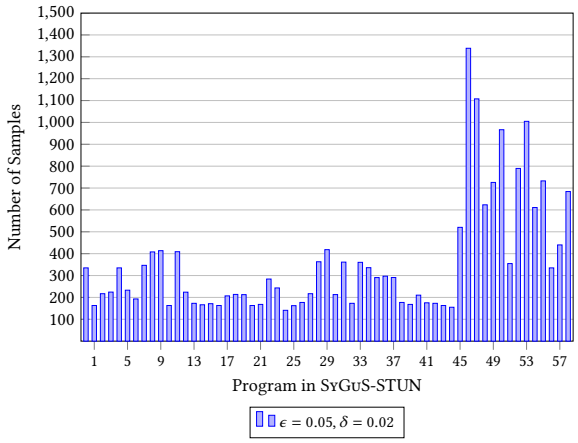

Figure 5: For most programs in the SyGuS-STUN, SynGuar-STUN synthesizes programs with provable generalization under 500 examples for $\epsilon = 0.05, \delta = 0.02$. Only 11 of these programs require $500 - 1400$ examples.

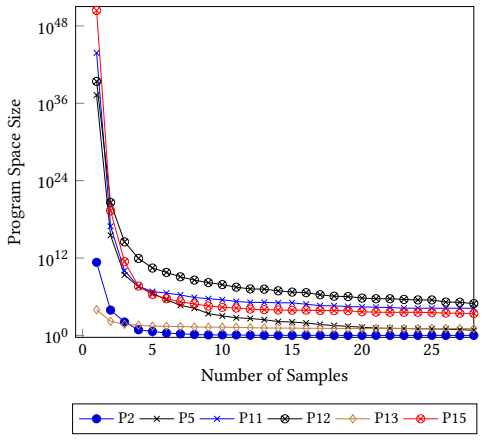

Figure 6: In PROSE-benchmark, the program space shrinks $3.7 \cdot 10^2 - 10^{44}\times$ on average with the first 5 examples, explaining why SynGuar-PROSE can provably generalize in modest number of samples for most programs in the benchmark.

that behave almost identically using random sampling will require many samples.

Figure 5 shows that we need between 140-1400 examples (about 357 on average) to achieve ($\epsilon = 0.05, \delta = 0.02$) generalization for the 59 target programs evaluated with SynGuar-STUN. Most of the programs in the SyGuS benchmark require under 500 samples, with only 11 programs requiring $500 - 1400$ I/O examples.

5.3 Reduction in Program Search Space

Note that Theorem 3.1 does *not* predict how fast the procedure will converge to a generalized program, i.e., how fast the space of consistent programs will shrink after each example. This varies empirically given the task and examples seen. But, SynGuar internally estimates (conservatively) how many programs remain consistent after seeing each example. From this, two empirical findings which explain our other observations emerge. First, the program space shrinks *drastically* with the first few examples for nearly all benchmarks. Second, the number of examples required to generalize depends significantly on the expressiveness of the chosen hypothesis space.

Figure 6 shows the size of the consistent program space, for 6 representative programs in the PROSE-benchmark computed by SynGuar-PROSE. The Y-axis is a logarithmic scale. The full evaluation of the rest of the programs is in the supplementary material [2]. We choose these programs as they represent the largest, smallest, and average cases of the program space size after 25 examples averaged over 32 runs, as well as the largest and smallest program space size decrease after 5 examples. In particular, P15 has the largest decrease on average in the program space size after 5 examples from 10^{50} to 10^6 while P13 has the smallest decrease after 5 examples from $9.7 \cdot 10^3$ to 27. This observation explains why a small number of examples turn out to be sufficient for generalization in this benchmark. It also shows that reducing the consistent hypothesis space further, after the initial quick reduction, becomes increasingly difficult with unbiased sampling.

The SyGuS benchmark has target programs of different complexity (different number of conditionals). For SynGuar-STUN, the number of samples required to generalize depends on target program's complexity—more complex programs require considering a larger original hypothesis space to be represented. However, the algorithm does not know the original hypothesis space. To use fewer samples, the algorithm chooses the smallest hypothesis space that still contains programs consistent with samples, because outputting more complex programs (more conditionals) requires choosing a larger hypothesis space for which more samples are needed.

We use the target programs in H_0 as an example to show this phenomenon in Figure 7. The figure shows for each target program the sample size sufficient for $(\epsilon, \delta/3)$-generalization calculated by the 3 parallel SynGuar instances on H_0, H_1, and H_2. We show that if SynGuar-STUN chooses a program in H_0, the number of examples is 141–419. If the target program is in H_0 but the synthesizer chooses a program in H_1, the number of samples is larger by 1166.22 on average. For example, for program 20 in our 3 runs, 213 samples are sufficient to pick a program in H_0, but to return a program from H_1 or H_2 SynGuar-STUN requires around 1300 and 2400 examples, respectively. This quantitatively shows that the sample size can vary by a large margin when considering more complex programs. Moreover, this result explains why choosing a simpler program first can require a smaller number of examples.

6 RELATED WORK

The overfitting problem in learning programs from examples is known. Many different approaches have been proposed to tackle it (see Section 1). One line of work proposes conditioning the search with program traces rather than just I/O examples [13, 21, 54]. Another line of work improves the input specification [18, 33] using domain-specific knowledge about the hypothesis space. Singh et al. propose to rank the synthesized functions based on distributional priors with machine learning [55]. Similarly, several inductive synthesis techniques use deep learning to improve their

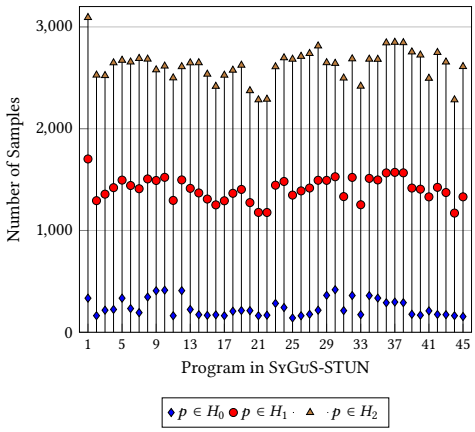

Figure 7: For target programs with no condition ($t \in H_0$), choosing a program $p \in H_0$, versus a program with one condition (H_1) or two conditions (H_2) leads to provable generalization in less number of samples.

search [7, 45, 61]. Broadly speaking, these approaches are complementary to ours as they either require domain-specific priors or they learn program patterns from a dataset of similar programs [47]. Moreover, none of them claims any generalization guarantees.

Several works suggest that larger test harnesses lead to promising improvements in synthesized programs [33, 44], which is also observed in related domains of program repair [23, 24] and invariant learning [9]. Our work, motivated by these, provides the formal bridge between test harness size and provable generalization. We also show that it translates into significantly improved accuracy on string-manipulating tasks, directly due to reduced overfitting.

Establishing generalization guarantees for synthesizers has been studied for over three decades. The PAC learnability framework has been introduced by Valiant [58] for analyzing generalization from a computational perspective. Under this theory, the sample complexity required for generalization has been established for learning in propositional logic domain [29, 51, 58]. The results have been extended to learning logic programs i.e., predicate logic domain [14, 15, 20]. Some of the above bounds are limited to certain types of hypothesis spaces. Instead, Blumer et al. have given two sample complexity bounds 1) union bound [10] and 2) Vapnik-Chervonenkis Dimension bound [11] that are more general. The union bound is used when the hypothesis space is finite and when the VC dimension of a model is difficult to estimate [8, 34]. Whilst the above works have established the bounds in theory, their applicability in real-world synthesizers have been very limited. A recent work uses the VC dimension argument for synthesizing linear arithmetic functions for holes given in a sketch [19]. For many programming domains, such as for string-manipulating programs, it is difficult to compute VC dimensions.

Besides the PAC framework, another line of work towards generalization is through active learning. Some interactive synthesis systems have question selection mechanisms to find distinguishing input [37, 60]. Ji et al. further approximate optimal questions to resolve ambiguity in less number of samples [31]. However, these approaches do not give generalization guarantees without assuming the existence of the target programs in the hypothesis space or a prior distribution over target programs, so they are orthogonal to our approach which works under minimal assumptions.

Outside of program synthesis, generalization has been extensively studied in machine learning. Our work bridges the two lines of inquiry that have evolved in parallel. Apart from PAC-style definitions based on sample complexity, generalization can be achieved using algorithmic stability [12]. Bounds have been established for both convex optimizations and non-convex optimization algorithms, i.e., low sensitivity to small changes in inputs [28, 36, 40, 43, 50]. These works leverage the properties of algorithms like stochastic gradient descent (SGD) and stochastic gradient Langevin dynamic (SGLD) in order to estimate the generalization bounds for a given number of samples. Adapting the framework of algorithmic stability to PBE-based synthesis is promising future work, but it is challenging. A direct adaptation, for example, would restrict learnt programs to be stable, for which small changes in outputs for small changes in inputs. Generalization has been explored from other perspectives such as by bounding network capacity [41] and overparametrization [4, 32, 62] in machine learning literature, which are also alternative starting points for studying generalization in program synthesis.

7 CONCLUSION

In this work, we exploit the theoretical connection between generalization and the numbers of examples used in programming-by-example synthesis. We provide the first principled approach that guarantees generalization with a modest number of examples in this regime. Key to this result is our mechanism for computing sample complexity on the fly. We show experimentally that this significantly reduces overfitting and improves accuracy for synthesizing string-manipulation programs, compared to approaches that use arbitrarily fewer examples.

ACKNOWLEDGMENTS

We thank Shiqi Shen, Shruti Hiray, and the anonymous reviewers for helpful feedback on this work. This work was supported by Crystal Centre at National University of Singapore, a Singapore Ministry of Education Academic Research Fund Tier 1 (WBS number R-252-000-B50-114), and a research grant with WBS number R-252-000-B14-281. All opinions in this work are solely those of the authors.

REFERENCES

[1] 2019. *SyGuS-Comp 2019.* https://sygus.org/comp/2019/
[2] 2021. *Supplementary Material.* https://github.com/HALOCORE/SynGuar
[3] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *Computer-Aided Verification (CAV).* https://doi.org/10.1007/978-3-642-39799-8_67
[4] Zeyuan Allen-Zhu, Yuanzhi Li, and Yingyu Liang. 2019. Learning and Generalization in Overparameterized Neural Networks, Going Beyond Two Layers. In *Conference on Neural Information Processing Systems (NeurIPS).*
[5] Rajeev Alur, Pavol Černý, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *Computer-Aided Verification (CAV).* https://doi.org/10.1007/978-3-319-21668-3_10
[6] Shengwei An, Rishabh Singh, Sasa Misailovic, and Roopsha Samanta. 2019. Augmented example-based synthesis using relational perturbation properties. In *Principles of Programming Languages (POPL).* https://doi.org/10.1145/3371124

[7] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. Deepcoder: Learning to write programs. In *International Conference on Learning Representations (ICLR)*.

[8] Guy Blanc, Jane Lange, and Li-Yang Tan. 2020. Top-Down Induction of Decision Trees: Rigorous Guarantees and Inherent Limitations. In *Innovations in Theoretical Computer Science (ITCS)*. https://doi.org/10.4230/LIPIcs.ITCS.2020.44

[9] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *USENIX Security*.

[10] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. 1987. Occam's razor. *Information processing letters* 24, 6 (1987), 377–380. https://doi.org/10.1016/0020-0190(87)90114-1

[11] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. 1989. Learnability and the Vapnik-Chervonenkis Dimension. *Journal of the ACM (JACM)* 36 (1989). https://doi.org/10.1145/76359.76371

[12] Olivier Bousquet and André Elisseeff. 2002. Stability and Generalization. *Journal of Machine Learning Research (JMLR)* 2 (2002).

[13] Xinyun Chen, Chang Liu, and Dawn Song. 2019. Execution-Guided Neural Program Synthesis. In *International Conference on Learning Representations (ICLR)*.

[14] William W. Cohen. 1994. Pac-learning recursive logic programs: efficient algorithms. *Journal of Artificial Intelligence Research (JAIR)* 2, 1 (1994), 501–539. https://doi.org/10.1613/jair.97

[15] William W. Cohen. 1994. Pac-learning recursive logic programs: negative results. *Journal of Artificial Intelligence Research (JAIR)* 2, 1 (1994), 541–573. https://doi.org/10.1613/jair.1917

[16] Tamraparni Dasu and Theodore Johnson. 2003. *Exploratory Data Mining and Data Cleaning*. Vol. 479. John Wiley & Sons. https://doi.org/10.1002/0471448354

[17] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: neural program learning under noisy I/O. In *International Conference on Machine Learning (ICML)*.

[18] Dana Drachsler-Cohen, Sharon Shoham, and Eran Yahav. 2017. Synthesis with Abstract Examples. In *Computer-Aided Verification (CAV)*. https://doi.org/10.1007/978-3-319-63387-9_13

[19] Samuel Drews, Aws Albarghouthi, and Loris D'Antoni. 2019. Efficient Synthesis with Probabilistic Constraints. In *Computer-Aided Verification (CAV)*. https://doi.org/10.1007/978-3-030-25540-4_15

[20] Sašo Džeroski, Stephen Muggleton, and Stuart Russell. 1992. PAC-learnability of determinate logic programs. In *Annual Workshop on Computational Learning Theory (COLT)*. 128–135. https://doi.org/10.1145/130385.130399

[21] Kevin Ellis and Sumit Gulwani. 2017. Learning to Learn Programs from Examples: Going Beyond Program Structure. In *International Joint Conferences on Artificial Intelligence Organization (IJCAI)*. https://doi.org/10.24963/ijcai.2017/227

[22] P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P. Madhusudan. 2018. Horn-ICE Learning for Synthesizing Invariants and Contracts. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*. https://doi.org/10.1145/3276501

[23] Xiang Gao, Shraddha Barke, Arjun Radhakrishna, Gustavo Soares, Sumit Gulwani, Alan Leung, Nachiappan Nagappan, and Ashish Tiwari. 2020. Feedback-Driven Semi-Supervised Synthesis of Program Transformations. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*. https://doi.org/10.1145/3428287

[24] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* (2019). https://doi.org/10.1145/3318162

[25] Peter Grünwald. 2007. *The Minimum Description Length Principle*. https://doi.org/10.7551/mitpress/4643.001.0001

[26] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Principles of Programming Languages (POPL)*. 317–330. https://doi.org/10.1145/1926385.1926423

[27] Sumit Gulwani. 2016. Programming by Examples - and its applications in Data Wrangling. *Verification and Synthesis of Correct and Secure Systems* (2016). https://doi.org/10.3233/978-1-61499-627-9-137

[28] Moritz Hardt, Benjamin Recht, and Yoram Singer. 2016. Train Faster, Generalize Better: Stability of Stochastic Gradient Descent. In *International Conference on Machine Learning (ICML)*.

[29] David Haussler. 1992. Decision theoretic generalizations of the PAC model for neural net and other learning applications. *Information and Computation* 100 (1992). https://doi.org/10.1016/0890-5401(92)90010-D

[30] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/1806799.1806833

[31] Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. 2020. Question Selection for Interactive Program Synthesis. In *Programming Language Design and Implementation (PLDI)*. 1143–1158. https://doi.org/10.1145/3385412.3386025

[32] K. Kawaguchi and J. Huang. 2019. Gradient Descent Finds Global Minima for Generalizable Deep Neural Networks of Practical Sizes. In *Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. https://doi.org/10.1109/ALLERTON.2019.8919696

[33] Larissa Laich, Pavol Bielik, and Martin Vechev. 2020. Guiding Program Synthesis by Learning to Generate Examples. In *International Conference on Learning Representations (ICLR)*.

[34] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53, 1 (2003), 111–156. https://doi.org/10.1023/A:1025671410623

[35] Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. 2000. Version Space Algebra and its Application to Programming by Demonstration. In *International Conference on Machine Learning (ICML)*.

[36] Ben London. 2017. A PAC-Bayesian Analysis of Randomized Learning with Application to Stochastic Gradient Descent. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[37] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *User Interface Software & Technology (UIST)*. 291–301. https://doi.org/10.1145/2807442.2807459

[38] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. Symbolic execution with existential second-order constraints. In *ESEC/FSE*. 389–399. https://doi.org/10.1145/3236024.3236049

[39] Tom M Mitchell. 1982. Generalization as search. *Artificial Intelligence* 18, 2 (1982), 203–226. https://doi.org/10.1016/0004-3702(82)90040-6

[40] Wenlong Mou, Liwei Wang, Xiyu Zhai, and Kai Zheng. 2018. Generalization Bounds of SGLD for Non-convex Learning: Two Theoretical Viewpoints. In *Conference on Learning Theory, PMLR*. 605–638.

[41] Behnam Neyshabur, Srinadh Bhojanapalli, David McAllester, and Nathan Srebro. 2018. A PAC-Bayesian Approach to Spectrally-Normalized Margin Bounds for Neural Networks. In *International Conference on Learning Representations (ICLR)*.

[42] Hila Peleg and Nadia Polikarpova. 2020. Perfect is the Enemy of Good: Best-Effort Program Synthesis. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.4230/LIPIcs.ECOOP.2020.2

[43] A. Pensia, V. Jog, and P. Loh. 2018. Generalization Error Bounds for Noisy, Iterative Algorithms. In *International Symposium on Information Theory (ISIT)*. https://doi.org/10.1109/ISIT.2018.8437571

[44] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-Driven Synthesis. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2594291.2594297

[45] Illia Polosukhin and Alexander Skidanov. 2018. Neural Program Search: Solving Programming Tasks from Description and Examples. In *ICLR workshop*.

[46] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, Vol. 50. 107–126. https://doi.org/10.1145/2858965.2814310

[47] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2676726.2677009

[48] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. 2019. cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis. In *Computer-Aided Verification (CAV)*. https://doi.org/10.1007/978-3-030-25543-5_5

[49] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer-Aided Verification (CAV)*. https://doi.org/10.1007/978-3-319-21668-3_12

[50] Omar Rivasplata, Emilio Parrado-Hernández, John Shawe-Taylor, Shiliang Sun, and Csaba Szepesvári. 2018. PAC-Bayes Bounds for Stable Algorithms with Instance-Dependent Priors. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[51] Ronald L. Rivest. 1987. Learning Decision Lists. *Machine Learning* 2 (1987). https://doi.org/10.1023/A:1022607331053

[52] Jean E. Sammet. 1966. The Use of English as a Programming Language. *Commun. ACM* (1966). https://doi.org/10.1145/365230.365274

[53] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. 2019. Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints. In *Network and Distributed Systems Security (NDSS)*. https://doi.org/10.14722/ndss.2019.23530

[54] Eui Chul Shin, Illia Polosukhin, and Dawn Song. 2018. Improving Neural Program Synthesis with Inferred Execution Traces. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[55] Rishabh Singh and Sumit Gulwani. 2015. Predicting a Correct Program in Programming by Example. In *Computer-Aided Verification (CAV)*. https://doi.org/10.1007/978-3-319-21690-4_23

[56] David Canfield Smith. 1975. *PYGMALION: A Creative Programming Environment*. Technical Report. Stanford University. https://doi.org/10.21236/ada016811

[57] Phillip D Summers. 1977. A methodology for LISP program construction from examples. *Journal of the ACM (JACM)* (1977). https://doi.org/10.1145/321992.322002

[58] L. G. Valiant. 1984. A Theory of the Learnable. *Commun. ACM* 27 (1984). https://doi.org/10.1145/1968.1972

[59] Vladimir Vapnik. 2000. *The Nature of Statistical Learning Theory*. Springer science & business media. https://doi.org/10.1007/978-1-4757-3264-1

[60] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Interactive Query Synthesis from Input-Output Examples. In *International Conference on Management of Data (SIGMOD)*. 1631–1634. https://doi.org/10.1145/3035918.3058738

[61] Amit Zohar and Lior Wolf. 2018. Automatic Program Synthesis of Long Programs with a Learned Garbage Collector. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[62] Difan Zou, Yuan Cao, Dongruo Zhou, and Quanquan Gu. 2020. Gradient descent optimizes over-parameterized deep ReLU networks. *Machine Learning* 109 (2020), 1–26. https://doi.org/10.1007/s10994-019-05839-6