Just-in-Time Learning for Bottom-up Enumerative Synthesis

SHRADDHA BARKE, UC San Diego
HILA PELEG, UC San Diego
NADIA POLIKARPOVA, UC San Diego

A key challenge in program synthesis is the astronomical size of the search space the synthesizer has to explore. In response to this challenge, recent work proposed to guide synthesis using learned probabilistic models. Obtaining such a model, however, might be infeasible for a problem domain where no high-quality training data is available. In this work we introduce an alternative approach to guided program synthesis: instead of training a model ahead of time we show how to bootstrap one just in time, during synthesis, by learning from partial solutions encountered along the way. To make the best use of the model, we also propose a new program enumeration algorithm we dub guided bottom-up search, which extends the efficient bottom-up search with guidance from probabilistic models.

We implement this approach in a tool called PROBE, which targets problems in the popular syntax-guided synthesis (SyGuS) format. We evaluate PROBE on benchmarks from the literature and show that it achieves significant performance gains both over unguided bottom-up search and over a state-of-the-art probability-guided synthesizer, which had been trained on a corpus of existing solutions. Moreover, we show that these performance gains do not come at the cost of solution quality: programs generated by PROBE are only slightly more verbose than the shortest solutions and perform no unnecessary case-splitting.

ACM Reference Format:

1 INTRODUCTION

Consider the task of writing a program that satisfies examples in Fig. 1. The desired program must return the substring of the input string s on different sides of the dash, depending on the input integer n. The goal of inductive program synthesis is to perform this task automatically, i.e. to generate programs from observations of their behavior.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>n</td>
</tr>
<tr>
<td>&quot;1/17/16-1/18/17&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;1/17/16-1/18/17&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;01/17/2016-01/18/2017&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;01/17/2016-01/18/2017&quot;</td>
<td>2</td>
</tr>
</tbody>
</table>

Fig. 1. A set of input-output examples for a string transformation pick-date (adapted from [eup 2018]).

Inductive synthesis techniques have made great strides in recent years [Feng et al. 2017a,b; Feser et al. 2015; Gulwani 2016; Osera and Zdancewic 2015; Shi et al. 2019; Wang et al. 2017a], and are powering practical end-user programming tools [Gulwani 2011; Inala and Singh 2018; Le and Gulwani 2014]. These techniques adopt different approaches to perform search over the space of all programs from a domain-specific language (DSL). The central challenge of program

Authors’ addresses: Shraddha Barke, UC San Diego, sbarke@eng.ucsd.edu; Hila Peleg, UC San Diego, hpeleg@eng.ucsd.edu; Nadia Polikarpova, UC San Diego, npolikarpova@eng.ucsd.edu.

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/
synthesis is scaling the search to complex programs: as the synthesizer considers longer programs, the search space grows astronomically large, and synthesis quickly becomes intractable, despite clever pruning strategies employed by state-of-the-art techniques.

For example, consider the following solution to the pick-date problem introduced above, using the DSL of a popular synthesis benchmarking platform SyGuS [Alur et al. 2013b]:

\[
\text{(substr } s \text{ (indexof (concat "-" } s \text{) "-" (- } n \text{ 1)) (indexof } s \text{ "-" } n))
\]

This solution extracts the correct substring of \( s \) by computing its starting index \( \text{indexof (concat "-" } s \text{) "-" (+ } n \text{ -1)) } \) to be either zero or the position after the dash, depending on the value of \( n \). At size 14, this is the shortest SyGuS program that satisfies the examples in Fig. 1. Programs of this complexity already pose a challenge to state-of-the-art synthesizers: none of the SyGuS synthesizers we tried were able to generate this or comparable solution within ten minutes\(^1\).

Guiding synthesis with probabilistic models. A promising approach to improving the scalability of synthesis is to explore more likely programs first. Prior work [Balog et al. 2016; Ellis et al. 2018; Lee et al. 2018; Menon et al. 2013] has proposed guiding the search using different types of learned probabilistic models. For example, if a model can predict, given the input-output pairs in Fig. 1, that \text{indexof} and \text{substr} are more likely to appear in the solution than other string operations, then the synthesizer can focus its search effort on programs with these operations and find the solution much quicker. Making this approach practical requires solving two major technical challenges: (1) how to obtain a useful probabilistic model? and (2) how to guide the search given a model?

Learning a model. Existing approaches [Bielik et al. 2016; Lee et al. 2018; Raychev et al. 2014] are able to learn probabilistic models of code automatically, but require significant amounts of high-quality training data, which must contain hundreds of meaningful programs per problem domain targeted by the synthesizer. Such data sets are generally difficult to obtain.

To address this challenge, we propose just-in-time learning, a novel technique that learns a probabilistic context-free grammar (PCFG) for a given synthesis problem “just in time”, i.e. during synthesis, rather than ahead of time. Previous work has observed [Peleg and Polikarpova 2020; Shi et al. 2019] that partial solutions—i.e. programs that satisfy a subset of input-output pairs—are often syntactically similar to the final solution. Our technique leverages this observation to collect partial solutions it encounters during search and update the PCFG on the fly, rewarding syntactic elements that occur in these programs. For example, when exploring the search space for the pick-date problem, unguided search quickly stumbles upon the short program \( \text{(substr } s \text{ 0 (indexof } s \text{ "-" } n) \), which is a partial solution, since it satisfies two of the four input-output pairs (with \( n = 1 \)). At this point, just-in-time learning picks up on the fact that \text{indexof} and \text{substr} seem to be promising operations to solve this problem, boosting their probability in the PCFG. Guided by the updated PCFG, our synthesizer finds the full solution in only 63 seconds.

Guiding the search. The state of the art in guided synthesis is weighted enumerative search using the \( A^* \) algorithm, implemented in the EuPHONY synthesizer [Lee et al. 2018] (see Sec. 7 for an overview of other guided search techniques). This algorithm builds upon top-down enumeration, which works by gradually filling holes in incomplete programs. Unfortunately, top-down enumeration is not a good fit for just-in-time learning: in order to identify partial solutions, the synthesizer needs to evaluate the programs it generates, while with top-down enumeration the majority of synthesizer’s time is spent generating incomplete programs that cannot (yet) be evaluated.

To overcome this difficulty, we propose guided bottom-up search, a new synthesis algorithm that, unlike prior work, builds upon bottom-up enumeration. This style of enumeration works by

---

\(^1\)CVC4 [Reynolds et al. 2019] is able to generate a solution within a minute, but its solution overfits to the examples and has size 73, which makes it hard to understand.
repeatedly combining small programs into larger programs; every generated program is complete and can be evaluated on the input examples, which enables just-in-time learning to rapidly collect a representative set of partial solutions. In addition, bottom-up enumeration leverages dynamic programming and a powerful pruning technique known as observational equivalence [Albarghouthi et al. 2013; Udupa et al. 2013], which further improves efficiency of synthesis. Our algorithm extends bottom-up search with the ability to enumerate programs in the order of decreasing likelihood according to a PCFG, and to our knowledge, is the first guided version of bottom-up enumeration. While guided bottom-up search enables just-in-time learning, it can also be used with an independently obtained PCFG.

The PROBE tool. We implemented guided bottom-up search with just-in-time learning in a synthesizer called PROBE. A high-level overview of PROBE is shown in Fig. 2. The tool takes as input an inductive synthesis problems in SyGuS format, i.e. a context-free grammar of the DSL and a set of input-output examples; it outputs a program from the DSL that satisfies all the examples. Optionally, PROBE can also take as input initial PCFG probabilities suggested by a domain expert or learned ahead of time.

We have evaluated PROBE on 140 SyGuS benchmarks from three different domains: string manipulation, bit-vector manipulation, and circuit transformation. PROBE is able to solve a total of 87 problems within a timeout of ten minutes, compared to only 44 problems for the baseline bottom-up synthesizer and 50 problems for EuPhony. Note that PROBE outperforms EuPhony despite requiring no additional training data, which makes it applicable to new domains where large sets of existing problems are not available. We also compared PROBE with CVC4 [Reynolds et al. 2019], the winner of the 2019 SyGuS competition. Although CVC4 solves more benchmarks than PROBE, its solutions are less interpretable and tend to overfit to the examples: CVC4 solutions are 8 times larger than PROBE solutions on average, and moreover, on the few benchmarks where larger datasets are available, CVC4 achieves only 68% accuracy on unseen data (while PROBE achieves perfect accuracy).

Contributions. To summarize, this paper makes the following contributions:

---

2PROBE also supports universally-quantified first-order specifications and reduces them to input-output examples using counter-example guided inductive synthesis (CEGIS) [Solar-Lezama et al. 2006]. Since this reduction is entirely standard, in the rest of the paper we focus on inductive synthesis, but we use both kinds of specifications in our evaluation.
<table>
<thead>
<tr>
<th>ID</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_0$</td>
<td>&quot;a &lt; 4 and a &gt; $\emptyset$&quot;</td>
<td>&quot;a 4 and a $\emptyset$&quot;</td>
</tr>
<tr>
<td>$e_1$</td>
<td>&quot;&lt;open and &lt;close&gt;&quot;</td>
<td>&quot;open and close&quot;</td>
</tr>
<tr>
<td>$e_2$</td>
<td>&quot;&lt;Change&gt; &lt;string&gt; to &lt;a&gt; number&quot;</td>
<td>&quot;Change string to a number&quot;</td>
</tr>
</tbody>
</table>

Fig. 3. Input-output example specification for the remove-angles benchmark (adapted from [eup 2018]).

\[
S \rightarrow \text{arg} | "" | "<" | ">" \quad \text{input string and string literals}
\]
\[
| (\text{replace } S S S) \quad \text{replace } s \times y \text{ replaces first occurrence of } x \text{ in } s \text{ with } y
\]
\[
| (\text{concat } S S) \quad \text{concatenate } x \text{ and } y
\]

Fig. 4. A simple CFG for string expressions and the informal semantics of its terminals.

(1) **Guided bottom-up search**: a bottom-up enumerative synthesis algorithm that explores programs in the order of decreasing likelihood defined by a PCFG (Sec. 4).

(2) **Just-in-time learning**: a new technique for updating a PCFG during synthesis by learning from partial solutions (Sec. 5).

(3) **Probe**: a prototype implementation of guided bottom-up search with just-in-time learning and its evaluation on benchmarks from prior work (Sec. 6).

2 BACKGROUND

In this section, we introduce the baseline synthesis technique that Probe builds upon: bottom-up enumeration with observational equivalence reduction [Albarghouthi et al. 2013; Udupa et al. 2013]. For exposition purposes, hereafter we use a simpler running example than the one in the introduction; the specification for this example, dubbed remove-angles, is given in Fig. 3. The task is to remove all occurrences of angle brackets from the input string.

2.1 Syntax Guided Synthesis

We formulate our search problem as an instance of **syntax-guided synthesis** (SyGuS) [Alur et al. 2013a]. In this setting, synthesizers are expected to generate programs in a simple language of S-expressions with built-in operations on integers (such as + or −) and strings (such as concat and replace). The input to a SyGuS problem is a syntactic specification, in the form of a context-free grammar (CFG) that defines the space of possible programs and a semantic specification that consists of a set of input-output examples\(^3\). The goal of the synthesizer is to find a program generated by the grammar, whose behavior is consistent with the semantic specification.

For our running example remove-angles, we adopt a very simple grammar of string expressions shown in Fig. 4. The semantic specification for this problem is the set of examples $\{e_0, e_1, e_2\}$ from Fig. 3. The program to be synthesized takes as input a string arg and outputs this string with every occurrence of "<" and ">" removed. Because the grammar in Fig. 4 allows no loops or recursion, and the replace operation only replaces the first occurrence of a given substring, the solution involves repeatedly replacing the substrings "<" and ">" with an empty string "". Fig. 5 shows one of the shortest solutions to this problem, which we dub replace-6. Note that this benchmark has multiple solutions of the same size that replace "<" and ">" in different order; for our purposes they are equivalent, so hereafter we will refer to any one of them as "the shortest solution". The

\(^3\)In general, SyGuS supports a richer class of semantic specifications, which can be reduced to example-based specifications using a standard technique, as we explain in Sec. 6

ID | Program | Examples Satisfied
--- | --- | ---
replace-2 | (replace (replace arg "<" ") ">" ") | \{e_{0}\}
replace-3 | (replace (replace (replace arg "<" ") "<" ") ">" ") | \{e_{0}, e_{1}\}
replace-6 | (replace (replace (replace (replace (replace arg "<" ") "<" ") "<" ") ">" ") ">" ") | \{e_{0}, e_{1}, e_{2}\}

Fig. 5. Shortest solutions for different subsets of examples of the remove-angles problem.

<table>
<thead>
<tr>
<th>Height</th>
<th># Programs</th>
<th>Bank</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>{arg, &quot;&lt;&quot;, &quot;&gt;&quot;, }</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>(concat arg arg), (concat arg &quot;&lt;&quot;), (concat arg &quot;&gt;&quot;), (concat &quot;&lt;&quot; &quot;&lt;&quot;), (concat &quot;&lt;&quot; &quot;&gt;&quot;), \ldots</td>
</tr>
<tr>
<td>2</td>
<td>1923</td>
<td>(concat arg (concat arg arg)), (concat arg (concat &quot;&gt;&quot; &quot;&gt;&quot;)), \ldots</td>
</tr>
<tr>
<td>3</td>
<td>\sim 30 M</td>
<td>(concat arg (concat replace arg &quot;&lt;&quot; arg arg)), (concat arg (concat replace arg &quot;&lt;&quot; arg) &quot;&lt;&quot;)</td>
</tr>
</tbody>
</table>

Fig. 6. Programs generated for remove-angles-short from the grammar in Fig. 4 in the order of height.

figure also shows two shorter programs, replace-2 and replace-3, which satisfy different subsets of the semantic specification and which we will refer to throughout the paper.

2.2 Bottom-up Enumeration

Bottom-up enumeration is a popular search technique in program synthesis, first introduced in the tools Transit [Udupa et al. 2013] and Escher [Albarghouthi et al. 2013]. We illustrate this search technique in action using a simplified version of our running example, remove-angles-short, where the semantic specification only contains the examples \{e_{0}, e_{1}\} (the shortest solution to this problem is the program replace-3 from Fig. 5).

**Bottom-up Enumeration.** Bottom-up enumeration is a dynamic programming technique that maintains a bank of enumerated programs and builds new programs by applying production rules to programs from the bank. Fig. 6 illustrates the evolution of the program bank on our running example. Starting with an empty bank, each iteration \(n\) builds and adds to the bank all programs of height \(n\). In the initial iteration, we are limited to production rules that require no subexpressions—literals and variables; this yields the programs of height zero: ", ", ", ", and arg. In each following iteration, we build all programs of height \(n + 1\) using the programs of height up to \(n\) as subexpressions. For example at height one, we construct all programs of the form concat \(x\ y\) and replace \(s\ x\ y\), where \(\langle s, x, y\rangle\) are filled with all combinations of height-zero expressions. The efficiency of bottom-up enumeration comes from reusing solutions to overlapping sub-problems, characteristic of dynamic programming: when building a new program, all its sub-expressions are taken directly from the bank and never recomputed.

**Observational Equivalence Reduction.** Bottom-up synthesizers further optimize the search by discarding programs that are observationally equivalent to some program that is already in the bank. Two programs are considered observationally equivalent if they evaluate to the same output for every input in the semantic specification. In our example, the height-one program (concat arg ")") is not added to the bank because it is equivalent to the height-zero program arg. This optimization shrinks the size of the bank at height one from 80 to 15; because each following iteration uses all combinations of programs from the bank as subexpressions, even a small reduction in bank size at lower heights leads to a significant overall speed-up.

Despite this optimization, the size of the bank grows extremely quickly with height, as illustrated in Fig. 6. In order to get to the desired program replace-3, which has height three, we need to enumerate anywhere between 1024 and \sim 30 M\ programs (depending on the order in which
productions and subexpressions are explored within a single iteration). Because of this search space explosion, bottom-up enumerative approach does not find replace-3 even after 20 minutes.

3 OUR APPROACH

In this section we first modify bottom-up search to enumerate programs in the order of increasing size rather than height (Sec. 3.1) and then generalize it to the order of decreasing likelihood defined by a probabilistic context-free grammar (Sec. 3.2). Finally, we illustrate how the probabilistic grammar can be learned just in time by observing partial solutions during search (Sec. 3.3).

3.1 Size-based Bottom-up Enumeration

Although exploring smaller programs first is common sense in program synthesis, the exact interpretation of "smaller" differs from one approach to another. As we discussed in Sec. 2, existing bottom-up synthesizers explore programs in the order of increasing height; at the same time, synthesizers based on other search strategies [Alur et al., 2013a, 2017b; Koukoutos et al., 2016] tend to explore programs in the order of increasing size—i.e. total number of AST nodes—rather than height, which has been observed empirically to be more efficient.

To illustrate the difference between the two orders, consider a hypothetical size-based bottom-up synthesizer. Fig. 7 shows how the bank would grow with each iteration on our running example. The solution replace-3 that we are looking for has size ten (and height three). Hence, size-based enumeration only has to explore up to 2048 programs to discover this solution (compared with up to \( \sim 30M \) for height-based enumeration). This is not surprising: a simple calculation shows that programs of height three range in size from 8 to 26, and our solution is towards the lower end of this range; in other words, replace-3 is tall and skinny rather than short and bushy. This is not a mere coincidence: in fact, prior work [Shah et al., 2018] has observed that useful programs tend to be skinny rather than bushy, and therefore exploration in the order of size has a better inductive bias.

Extending bottom-up enumeration. Motivated by this observation, we extend the bottom-up enumerative algorithm from Sec. 2.2 to explore programs in the order of increasing size. To this end, we modify the way subexpressions are selected from the bank in each search iteration. For example, to construct programs of size four of the form \( \text{concat } x \ y \), we only replace \( \langle x, y \rangle \) with pairs of programs whose sizes add up to three (the concat operation itself takes up one AST node). This modest change to the search algorithm yields surprising efficiency improvements: our size-based bottom-up synthesizer is able to solve the remove-angles-short benchmark in only one second! (Recall that the baseline height-based synthesizer times out after 20 minutes).

Unfortunately, the number of programs in the bank still grows exponentially with program size, limiting the range of sizes that can be explored efficiently: for example, the solution to the original
Fig. 8. A PCFG for string expressions that is biased towards the solution \texttt{replace-6}. For each production rule \( R \), we show its probability \( p_R \) and its cost \( \text{cost}_R \), which is computed as a rounded negative log of the probability.

Remove-angles benchmark (replace-6) has size 19, and size-based enumeration is unable to find it within the 20 minute timeout. This is where \textit{guided bottom-up search} comes to the rescue.

### 3.2 Guided Bottom-up Search

Previous work has demonstrated significant performance gains in synthesizing programs by exploiting probabilistic models to guide the search \cite{Balog2016, Lee2018, Menon2013}. These techniques, however, do not build upon bottom-up enumeration, and hence cannot leverage its two main benefits: reuse of subprograms and observational equivalence reduction (Sec. 2.2). Our first key contribution is modifying the size-based bottom-up enumeration technique from previous section to guide the search using a \textit{probabilistic context-free grammar} (PCFG). We refer to this modification of the bottom-up algorithm as \textit{guided bottom-up search}.

**Probabilistic context-free grammars.** A PCFG assigns a probability to each production rule in a context-free grammar. For example, Fig. 8 depicts a PCFG for our running example that is biased towards the correct solution: it assigns high probabilities to the rules (operations) that appear in \texttt{replace-6} and a low probability to the rule \texttt{concat} that does not appear in this program. As a result, this PCFG assigns a higher likelihood to the program \texttt{replace-6} than it does to other programs of the same size. Hence, an algorithm that explores programs in the order of decreasing likelihood would encounter \texttt{replace-6} sooner than size-based enumeration would.

**From probabilities to discrete costs.** Unfortunately, size-based bottom-up enumeration cannot be easily adapted to work with real-valued probabilities. We observe, however, that the order of program enumeration need not be exact: enumerating \textit{approximately} in the order of decreasing likelihood still benefits the search. Our insight therefore is to convert rule probabilities into discrete costs, which are computed as their rounded negative logs. According to Fig. 8, the high-probability rules have a low cost of two, and the low-probability rule \texttt{concat} has a higher cost of four. The cost of a program is computed by summing up the costs of its productions, for example:

\[
\text{cost}((\text{concat} \text{ arg } "<") = \text{cost}((\text{concat}) + \text{cost}(\text{arg}) + \text{cost}("<")
\]

\[
= 4 + 2 + 2 = 8
\]

Hence, the order of increasing cost approximately matches the order of decreasing likelihood.

**Extending size-based enumeration.** With the discrete costs at hand, guided bottom-up search is essentially the same as the size-based search detailed in Sec. 3.1, except that it takes the cost of the top-level production into account when constructing a new program. Fig. 9 illustrates the working of this algorithm. For example, at cost level 8, we build all programs of the form \texttt{concat x y}, where the costs of \( x \) and \( y \) sum up to \( 8 - 4 = 4 \). The cost of our solution \texttt{replace-6} is 38, which places it within the first 175K programs the search encounters; on the other hand, its size is 19, placing it within the first \( \sim 3M \) programs in the order of size. As a consequence, size-based enumeration

\[\text{Proc. ACM Program. Lang., Vol. 1, No. OOPSLA, Article 1. Publication date: January 2020.}\]
cannot find this program within 20 minutes, but guided enumeration, given the PCFG from Fig. 8, is able to discover replace-6 within 5 seconds.

### 3.3 Just-in-Time Learning

In the previous section we have seen that guided bottom-up search can find solutions efficiently, given an appropriately biased PCFG. But how can we obtain such a PCFG for each synthesis problem? Prior approaches have proposed learning probabilistic models either from a corpus of existing solutions [Lee et al., 2018; Menon et al., 2013] or from synthetic data [Balog et al., 2016]. While achieving impressive results, these approaches are computationally expensive and, more importantly, require high-quality training data, which is generally hard to obtain (see Sec. 7 for a detailed discussion). Can we benefit from guided search when training data is not available?

Our second key contribution is a new approach to learning probabilistic models of programs, which we dub just-in-time learning. This approach is inspired by an observation made in prior work [Peleg and Polikarpova, 2020; Shi et al., 2019] that partial solutions—programs that satisfy a subset of the semantic specification—often share syntactic similarity with the full solution. We can leverage this insight to iteratively bias the PCFG during synthesis, rewarding productions that occur in partial solutions we encounter.

**Enumeration with just-in-time learning.** We illustrate just-in-time learning on our running example remove-angles. We begin enumeration with a uniform PCFG, which assigns the same probability to each production. In this initial PCFG every production has cost 3 (see Fig. 10).

With a uniform PCFG, our search starts off exactly the same as size-based search of Sec. 3.1. At size 7 (cost level 21), the search encounters the program replace-2, which satisfies the example $e_0$. Since this program contains productions replace, arg, "", ">", and ",", we **reward** these productions by decreasing their cost, as indicated in Fig. 10; after this update, the cost of the production concat does not change, so our solution is now cheaper relative to other programs of the same size. With the new PCFG at hand, the enumeration soon encounters another partial solution, replace-3, which covers the examples $e_0$ and $e_1$. Since this program uses the same productions as replace-2 and satisfies even more examples, the difference in cost between the irrelevant production concat and the relevant ones increases even more: in fact, we have arrived at the same biased PCFG we used in Sec. 3.2 to illustrate the guided search algorithm.

**Challenge: selecting promising partial solutions.** As this example illustrates, the more partial solutions we encounter that are similar to the final solution, the more biased the PCFG becomes, gradually steering the search in the right direction. The key challenge with this approach is that the search might encounter hundreds or thousands of partial solutions, and many of them have irrelevant syntactic features. In our running example, there are in fact more than 3100 programs that satisfy at least one of the examples $e_0$ or $e_1$. For instance, the program

\[(\text{replace} (\text{replace} (\text{replace} \text{arg} "<" "") ">" ") "<" (\text{concat} (\text{concat} \text{arg} \text{arg}) \text{arg}))\]

5The algorithm can also be initialized with a pre-learned PCFG if one is available.
satisfies $e_0$, but contains the concat production, so if we use this program to update the PCFG, we would steer the search away from the final solution. Hence, the core challenge is to identify promising partial solutions, and only use those to update the PCFG.

A closer look at this program reveals that it has the same behavior as the shorter program `replace-2`, but it contains an irrelevant subexpression that appends "<" to arg only to immediately replace it with an empty string! In our experience, this is a common pattern: whenever a partial solution $p'$ is larger than another partial solution $p$ but solves the same subset of examples, then $p'$ often syntactically differs from $p$ by an irrelevant subexpression, which happens to have no effect on the inputs solved by the two programs. Following this observation, we only consider a partial solution $p$ promising—and use it to update the PCFG—when it is one of the shortest solutions that covers a given subset of examples.

Powered by just-in-time learning, PROBE is able to find the solution `replace-6` within 36 seconds, starting from a uniform PCFG: only a slight slowdown compared with having a biased PCFG from the start. Note that EuPhony, which uses a probabilistic model learned from a corpus of existing solutions, is unable to solve this benchmark even after 10 minutes.

4 GUIDED BOTTOM-UP SEARCH

In this section, we describe our guided bottom-up search algorithm. We first formulate our problem of guided search as an instance of an inductive SyGuS problem. We then present our algorithm that enumerates programs in the order of decreasing likelihood.

4.1 Preliminaries

Context-free Grammar. A context-free grammar (CFG) is a quadruple $G = (N, \Sigma, S, R)$, where $N$ denotes a finite, non-empty set of non-terminal symbols, $\Sigma$ denotes a finite set of terminals, $S$ denotes the starting non-terminal, and $R$ is the set of production rules. In our setting, each terminal $t \in \Sigma$ is associated with an arity $\text{arity}(t) \geq 0$, and each production rule $R \in R$ is of the form $N \rightarrow (t N_1 \ldots N_k)$, where $N, N_1, \ldots, N_k \in N$, $t \in \Sigma$, and $\text{arity}(t) = k^6$. We denote with $R(N)$ the set of all rules $R \in R$ whose left-hand side is $N$. A sequence $\alpha \in (N \cup \Sigma)^*$ is called a sentential form and a sequence $s \in \Sigma^*$ is a called a sentence. A grammar $G$ defines a (leftmost) single-step derivation relation on sentential forms: $sNa \Rightarrow s\beta(a)$ if $N \rightarrow \beta \in R$. The reflexive transitive closure of this relation is called (leftmost) derivation and written $\Rightarrow^*$. All grammars we consider are unambiguous, i.e., every sentential form has at most one derivation.

Programs. A program $P$ is a sentence derivable from some $N \in N$; we call a program whole if it is derivable from $S$. The set of all programs is called the language of the grammar $G$: $L(G) = \{ s \in \Sigma^* \mid N \Rightarrow^* s \}$. The trace of a program $\text{tr}(P)$ is the sequence of production rules $R_1, \ldots, R_n$ used in its derivation $(N \Rightarrow a_1 \Rightarrow \ldots \Rightarrow a_{n-1} \Rightarrow P)$. The size of a program $|P|$ is the length of its trace. We assign semantics $[P] : \text{Val}^* \rightarrow \text{Val}$ to each program $P$, where Val is the set of run-time values.

---

6 An astute reader might have noticed that we can formalize this grammar as a regular tree grammar instead; we decided to stick with the more familiar context-free grammar for simplicity.
Inductive syntax-guided synthesis. An inductive syntax-guided synthesis (SyGuS) problem is defined by a grammar $G$ and a set of input-output examples $E = \{(i, o)\}$, where $i \in \text{Val}^*$, $o \in \text{Val}^7$. A solution to the problem is a program $P \in \mathcal{L}(G)$ such that $\forall (i, o) \in E$, $[P](i) = o$. Without loss of generality, we can assume that only whole programs can evaluate to the desired outputs $o$, hence our formulation need not explicitly require that the solution be whole.

Probabilistic Context-free Grammar. A probabilistic context-free grammar (PCFG) $G_p$ is a pair of a CFG $G$ and a function $p: \mathcal{R} \rightarrow [0, 1]$ that maps each production rule $R \in \mathcal{R}$ to its probability. Probabilities of all the rules for given non-terminal $N \in \mathcal{N}$ sum up to one: $\forall N, \sum_{R \in \mathcal{R}(N)} p(R) = 1$. A PCFG defines a probability distribution on programs: a probability of a program is the product of probabilities of all the productions in its trace $p(P) = \prod_{R_i \in \text{tr}(P)} p(R_i)$.

Costs. We can define the real cost of a production as $\text{rcost}(R) = -\log(p(R))$; then the real costs of a program can be computed as $\text{cost}(P) = -\log(p(P)) = \sum_{R_i \in \text{tr}(P)} \text{rcost}(R_i)$. For the purpose of our algorithm, we define discrete costs, which are real costs rounded to the nearest integer: $\text{cost}(R) = \lfloor \text{rcost}(R) \rfloor$. The cost of a program $P$ is defined as the sum of costs of all the productions in its trace: $\text{cost}(P) = \sum_{R_i \in \text{tr}(P)} \text{cost}(R_i)$.

4.2 Guided Bottom-up Search Algorithm

Algorithm 1 presents our guided bottom-up search algorithm. The algorithm takes as input a PCFG $G_p$ and a set of input-output examples $E$, and enumerates programs in the order of increasing discrete costs according to $G_p$, until it finds a program $P$ that satisfies the entire specification $E$ or reaches a certain cost limit. The algorithm maintains a search state that consists of (1) the current cost level; (2) program bank $B$, which stores all enumerated programs indexed by their cost; (3) evaluation cache $E$, which stores evaluation results of all programs in $B$ (for the purpose of checking observational equivalence); and (4) the set PSol, which stores all enumerated partial solutions. Note that the algorithm returns the current search state and optionally takes a search state as input; we make use of this in Sec. 5 to resume search from a previously saved state.

Every iteration of the loop in lines 3–14 enumerates all programs whose costs is equal to level. New programs with a given cost are constructed by the auxiliary procedure NewPrograms, which we describe below. In line 5, every new program $P$ is evaluated on the inputs from the bank that have the right costs and the right types.

To construct a program of cost level, the procedure iterates over all production rules $R \in \mathcal{R}$. Once $R$ is chosen as the top-level production in the derivation of the new program, we have a budget of level $-\text{cost}(R)$ to allocate between the subexpressions; line 21 iterates over all possible subexpression costs that add up to this budget. Once the subexpression costs $c_1, \ldots, c_k$ have been fixed, line 22 iterates over all $k$-tuples of programs from the bank that have the right costs and the right types to serve as subexpressions: $N_i \Rightarrow^* P_i$.
means that $P_i$ can replace the nonterminal $N_i$ in the production rule $R$. Finally, line 23 builds a program from the production rule $R$ and the subexpressions $P_i$.

4.3 Guarantees

**Soundness.** The procedure GUIDED-SEARCH is sound: given $\mathcal{G}_p = (\mathcal{G}, p)$ and $\mathcal{E}$, if the procedure returns $(P, \bot)$, then $P$ is a solution to the inductive SyGuS problem $(\mathcal{G}, \mathcal{E})$. It is straightforward to show that $P$ satisfies the semantic specification $\mathcal{E}$, since we check this property directly in line 6. Furthermore, $P \in \mathcal{L}(\mathcal{G})$, since $P$ is constructed by applying a production rule $R$ to programs derived from appropriate non-terminals (see check in line 22).

**Completeness.** The procedure GUIDED-SEARCH is complete: if $P^*$ is a solution to the inductive SyGuS problem $(\mathcal{G}, \mathcal{E})$, such that $\text{cost}(P^*) = C$, and $C \leq \text{level}_0 + \text{lim}$, the the algorithm will return $(P, \bot)$, where $\text{cost}(P) \leq C$. Completeness follows by observing that each level of the bank is complete up to observational equivalence: if $P \in \mathcal{L}(\mathcal{G})$ and $\text{cost}(P) \leq C$, then at the end of the iteration with level $C$, either $P \in B$ or $\exists P' \in B \text{ s.t. } \text{cost}(P') \leq \text{cost}(P)$ and $\forall (i, o) \in \mathcal{E}, [P](i) = [P'](i)$. This in turn follows from the completeness of NEW-PROGRAMS (it considers all combinations of costs of $R$ and the subexpressions that add up to level), monotonicity of costs (replacing a subexpression with a more expensive one yields a more expensive program) and compositionality.
Algorithm 2 The Probe algorithm

Input: CFG $G$, set of input-output examples $E$
Output: A solution $P$ or ⊥
1: procedure Probe($G, E$) 
2: $G_p$ ← ⟨$G, p_u$⟩  
3: level, B, E ← 0, ∅, ∅  
4: while not timeout do 
5:  $P, (\text{level}, B, E, \text{PSol})$ ← guided-search ($G_p, E, (\text{level}, B, E, \emptyset)$) 
6:  if $P \neq ⊥$ then 
7:    return $P$  
8:  PSol ← Select(PSol, E)  
9:  if PSol ≠ ∅ then 
10:    $G_p$ ← Update($G_p, \text{PSol}, E$)  
11:    level, B, E ← 0, ∅, ∅  
12: return ⊥

of program semantics (replacing a subexpression with an observationally equivalent one yields an observationally equivalent program).

Prioritization. We would also like to claim that Guided-Search enumerates programs in the order of decreasing likelihood. This property would hold precisely if we were to enumerate programs in order of increasing real cost $rcost$: since the log function is monotonic, $p(P_1) < p(P_2)$ iff $rcost(P_1) < rcost(P_2)$. Instead Guided-Search enumerates programs in the order of increasing discrete cost $cost$, so this property only holds approximately due to the rounding error. Empirical evaluation shows, however, that this approximate prioritization is effective in practice (Sec. 6).

5 JUST IN TIME LEARNING

In this section, we introduce a new technique we call just-in-time learning that updates the probabilistic model used to guide synthesis by learning from partial solutions. We first present the overall Probe algorithm in Sec. 5.1 and then discuss the three steps involved in updating the PCFG in the remainder of the section.

5.1 Algorithm summary

The overall structure of the Probe algorithm is presented in Algorithm 2. The algorithm iterates between the following two phases until timeout is reached:

1. Synthesis phase searches over the space of programs in order of increasing discrete costs using the procedure Guided-Search from Sec. 4.
2. Learning phase updates the PCFG using the partial solutions found in the synthesis phase.

Probe takes as input an inductive SyGuS problem $G$, $E$. It starts by initializing the PCFG with CFG $G$ and a uniform distribution $p_u$, which assigns every production rule $R = N \rightarrow \beta$ the probability $p(R) = 1/|R(N)|$. Each iteration of the while-loop corresponds to one synthesis-learning cycle. In each cycle, Probe first invokes Guided-Search with the current search state. If the search finds a solution, Probe terminates successfully (line 7); otherwise it enters the learning phase, which consists of three steps. First, procedure Select selects promising partial solutions (line 8); if no such solutions have been found, the search simply resumes from the current state. Otherwise (line 9), the second step is to use the promising partial solutions to Update the PCFG, and the third step is to restart the search. These three steps are detailed in the rest of this section.
### 5.2 Selecting Promising Partial Solutions

The procedure `Select` takes as input the set of partial solutions `PSol` returned by `guided-search`, and selects the ones that are (promising) and should be used to update the PCFG. We illustrate this process using the synthesis problem in Fig. 11; some partial solutions generated for this problem are listed in Fig. 12. The shortest full solution for this problem is:

\[
\text{(substr arg (indexof (concat " " arg) " " 3) 3)}
\]

**Objectives.** An effective selection procedure must balance the following two objectives.

(a) **Avoid rewarding irrelevant productions:** The reason we cannot simply use all generated partial solutions to update the PCFG is that partial solutions often contain irrelevant subprograms, which do not in fact contribute to solving the synthesis problem; rewarding productions from these irrelevant subprograms derails the search. For example, consider `P_0` and `P_1` in Fig. 12: intuitively, these two programs solve the examples \{e_0, e_1\} in the same way, but `P_1` also performs an extraneous character replacement, which happens to not affect its behavior on these examples. Hence we would like to discard `P_1` from consideration to avoid rewarding the irrelevant production `replace`.

Observe that `P_0` and `P_1` satisfy the same subset of examples but `P_1` has a higher cost; this suggests discarding partial solutions that are subsumed by a cheaper program.

(b) **Reward different approaches:** On the other hand, different partial solutions might represent inherently different approaches to solving the task at hand. For example, consider partial solutions `P_0` and `P_4` in Fig. 12; intuitively, they represent different strategies for computing the starting position of the substring: fixed index vs. search (indexof). We would like to consider `P_4` promising: indeed, `indexof` turns out to be useful in the final solution. We observe that although `P_4` solves the same number of examples and has a higher cost than `P_0`, it solves a different subset of examples, and hence should be considered promising.

Our goal is to find the right trade-off between the two objectives. Selecting too many partial solutions might lead to rewarding irrelevant productions and more frequent restarts (recall that search is restarted only if new promising partial solutions were found in the current cycle). On the other hand, selecting too few partial solutions might lead the synthesizer down the wrong path or simply not provide enough guidance, especially when the grammar is large.

**Selection schemes.** Based on these objectives, we designed three selection schemes, which make different trade-offs and are described below from most to least selective. Note that all selection

---

**Fig. 11.** A set of input-output examples for a string transformation (adapted from [eup 2018]).

<table>
<thead>
<tr>
<th>ID</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_0</td>
<td>&quot;+83 973-757-831&quot;</td>
<td>&quot;973&quot;</td>
</tr>
<tr>
<td>e_1</td>
<td>&quot;+72 001-050-856&quot;</td>
<td>&quot;001&quot;</td>
</tr>
<tr>
<td>e_2</td>
<td>&quot;+172 027-507-632&quot;</td>
<td>&quot;027&quot;</td>
</tr>
</tbody>
</table>

**Fig. 12.** Partial solutions and the corresponding subset of examples satisfied for the problem in Fig. 11.
schemes need to preserve information about promising partial solutions between different synthesis-learning cycles, to avoid rewarding the same solution again after synthesis restarts. We evaluate the effectiveness of these schemes in comparison to the baseline (using all partial solutions) in Sec. 6.

1. **LARGEST SUBSET**: This scheme selects a single cheapest program (first enumerated) that satisfies the largest subset of examples encountered so far across all synthesis cycles. Consequently, the number of promising partial solutions it selects is always smaller than the size of E. Among partial solutions in Fig. 12, this scheme picks a single program \(P_0\).

2. **FIRST CHEAPEST**: This scheme selects a single cheapest program (first enumerated) that satisfies a unique subset of examples. The partial solutions \(\{P_0, P_2, P_4\}\) from Fig. 12 are selected by this scheme. This scheme still rewards a small number of partial solutions, but allows different approaches to be considered.

3. **ALL CHEAPEST**: This scheme selects all cheapest programs (enumerated during a single cycle) that satisfy a unique subset of examples. The partial solutions \(\{P_0, P_2, P_3, P_4\}\) are selected by this scheme. Specifically, \(P_2\) and \(P_3\) satisfy the same subset of examples; both are considered since they have the same cost. This scheme considers more partial solutions than FIRST CHEAPEST, which refines the ability to reward different approaches.

### 5.3 Updating the PCFG

Procedure **UPDATE** uses the set of promising partial solution PSol to compute the new probability for each production rule \(R \in \mathcal{R}\) using the formula:

\[
p(R) = \frac{p_u(R)(1 - \text{fit})}{Z}
\]

where \(\text{fit} = \max_{\{P \in \text{PSol} | R \in \text{tr}(P)\}} |\mathcal{E} \cap \mathcal{E}[P]| / |\mathcal{E}|\)

where \(Z\) denotes the normalization factor, and \(\text{fit}\) is the highest proportion of input-output examples that any partial solution derived using this rule satisfies. Recall that \(p_u\) is the uniform distribution for \(G\). This rule assigns higher probabilities to rules that occur in partial solutions that satisfy many input-output examples.

### 5.4 Restarting the Search

Every time the PCFG is updated during a learning phase, **PROBE** restarts the bottom-up enumeration from scratch, i.e. empties the bank \(B\) (and the evaluation cache \(E\)) and resets the current cost level to zero. At a first glance this seems like a waste of computation: why not just resume the enumeration from the current state? The challenge is that any update to the PCFG renders the program bank outdated, and updating the bank to match the new PCFG requires the amount of computation and/or memory that does not pay off in relation to the simpler approach of restarting the search. Let us illustrate these design trade-offs with an example.

Consider again the synthesis problem in Fig. 11, and two programs encountered during the first synthesis cycle: the program \(\theta\) with cost 5 and the program \((\text{indexof arg } "+" )\) with cost 15. Note that both programs evaluate to 0 on all three example inputs, i.e. they belong to the same observational equivalence class \([0, 0, 0]\); hence the latter program is discarded by observational equivalence reduction, while the former, discovered first, is chosen as the representative of its equivalence class and appears in the current bank \(B\).

Now assume that during the subsequent learning phase the PCFG changed in such a way that the new costs of these two programs are \(\text{cost}(\theta) = 15\) and \(\text{cost}((\text{indexof arg } "+" )) = 5\). Let us examine different options for the subsequent synthesis cycle.

1. **Restart from scratch**: If we restart the search with an empty bank, the program \((\text{indexof arg } "+" )\) is now encountered before the program \(\theta\) and selected as the representative of its equivalence class.
class. In other words, the desired behavior under the new PCFG is that the class \([0, 0, 0]\) has cost 5. How can we achieve this behavior without restarting the search?

(2) **Keep the bank unchanged**: Resuming the enumeration with \(B\) unchanged would be incorrect: in this case the representative of \([0, 0, 0]\) is still the program \(\emptyset\) with cost 5. As a result, any program we build in the new cycle that uses this equivalence class as a sub-program would have a wrong cost, and hence the enumeration order would be different from that prescribed by the new PCFG.

(3) **Re-index the bank**: Another option is to keep the programs stored in \(B\) but re-index it with their updated costs: for example, index the program \(\emptyset\) with cost 15. This does not solve the problem, however: now class \([0, 0, 0]\) has cost 15 instead the desired cost 5, because it still has a wrong representative in \(B\). Therefore, in order to enforce the correct enumeration order in the new cycle we need to update the equivalence class representatives stored in the bank.

(4) **Update representatives**: To be able to update the representatives, we need to store the redundant programs in the bank instead of discarding them. To this end, prior work [Phothilimthana et al. 2016; Wang et al. 2017b,c] has proposed representing the bank as a **finite tree automaton**, i.e. a hypergraph where nodes correspond to equivalence classes (such as \([0, 0, 0]\)) and edges correspond to productions (with the corresponding arity). The representative program of an equivalence class can be computed as the shortest hyper-path to the corresponding node from the set of initial nodes (inputs and literals); the cost of the class is the length of such shortest path. When the PCFG is updated, leading to modified costs of hyper-edges, shortest paths for all nodes in this graph need to be recomputed. Algorithms for doing so [Gao et al. 2012] have super-linear complexity in the number of affected nodes. Since in our case most nodes are likely to be affected by the update, and since the number of nodes in the hypergraph is the same as the size of our bank \(B\), this update step is roughly as expensive as rebuilding the bank from scratch. In addition, for a search space as large as the one PROBE explores for the SyGuS string benchmarks, the memory overhead of storing the entire hypergraph is also prohibitive.

Since restarting the search is expensive, PROBE does not return from the guided search immediately once a partial solution is found and instead keeps searching until a fixed cost limit and returns partial solutions in batches. There is a trade-off between restarting synthesis too often (wasting time exploring small programs again and again) and restarting too infrequently (wasting time on unpromising parts of the search space when an updated PCFG could guide the search better). In our implementation, we found that setting the cost limit to \(6 \cdot C\) works best empirically, where \(C\) is the maximum production cost in the initial PCFG (this roughly corresponds to enumerating programs in size increments of six with the initial grammar).

### 6 EXPERIMENTS

We have implemented the PROBE synthesis algorithm in Scala\(^8\). In this section, we empirically evaluate how PROBE compares to the baseline and state-of-the-art synthesis techniques. We design our experiments to answer the following research questions:

- **Q1** How effective is the just-in-time learning in PROBE? We examine this question in two parts:
  1. by comparing PROBE to baseline unguided bottom-up enumerative techniques, and
  2. by comparing different schemes for partial solution selection.

- **Q2** Is PROBE faster than state-of-the-art SyGuS solvers?

- **Q3** Is the quality of PROBE solutions comparable with state-of-the-art SyGuS solvers?

\(^8\)https://github.com/shraddhabarke/probe.git

6.1 Experimental Setup
We evaluate Probe on three different application domains: string manipulation (STRING), bit-vector manipulation (BitVec), and circuit transformations (Circuit). We perform our experiments on a set of total 140 benchmarks, 82 of which are STRING benchmarks, 27 are BitVec manipulation benchmarks and 31 are Circuit benchmarks. The grammars containing the available operations for each of these domains appear in the extended version of this paper [Barke et al. 2020].

STRING Benchmarks. The 82 STRING benchmarks are taken from the testing set of EuPhony [eup2018]. The entire EuPhony String benchmark suite consists of 205 problems, from the PBE-String track of the 2017 SyGuS competition and from string-manipulation questions from popular online forms. EuPhony uses 82 out of these 205 benchmarks as their testing set based on the criteria that EUSolver [Alur et al. 2017b] could not solve them within 10 minutes. STRING benchmark grammars have a median of 16 operations, 11 literals, and 1 variable. All these benchmarks use input-output examples as semantic specification, and the number of examples ranges from 2 to 400.

BitVec Benchmarks. The 27 BitVec benchmarks originate from the book Hacker’s Delight [Warren 2013], commonly referred to as the bible of bit-twiddling hacks. We took 20 of them verbatim from the SyGuS competition suite: these are all the highest difficulty level (d5) Hacker’s Delight benchmarks in SyGuS. We then found 7 additional loop-free benchmarks in synthesis literature [Gulwani et al. 2011; Jha et al. 2010] and manually encoded them in the SyGuS format. BitVec benchmark grammars have a median of 17 operations, 3 literals, and 1 variable. The semantic specification of BitVec benchmarks is a universally-quantified first-order that is functionally equivalent to the target program.

Note that in addition to Hacker’s Delight benchmarks, the SyGuS bitvector benchmark set also contains EuPhony bitvector benchmarks. We decided to exclude these benchmarks from our evaluation because they have very peculiar solutions: they all require extensive case-splitting, and hence are particularly suited to synthesizers that perform condition abduction [Albarghouthi et al. 2013; Alur et al. 2017b; Kneuss et al. 2013]. Since Probe (unlike EuPhony) does not implement condition abduction, it is bound to perform poorly on these benchmarks. At the same time, condition abduction is orthogonal to the techniques introduced in this paper; hence Probe’s performance on these benchmarks would not be informative.

Circuit Benchmarks. The 31 Circuit benchmarks are taken from the EuPhony testing set. These benchmarks involve synthesizing constant-time circuits that are cryptographically resilient to timing attacks. Circuit benchmark grammars have a median of 4 operations, 0 literals, and 6 variables. The semantic specification is a universally-quantified boolean formula functionally equivalent to the circuit to be synthesized.

Reducing first-order specifications to examples. As discussed above, only the string domain uses input-output examples as the semantic specification, while the other two domains use a more general SyGuS formulation where the specification is a (universally-quantified) first-order formula. We extend Probe to handle the latter kind of specifications in a standard way (see e.g. [Alur et al. 2017b]), using counter-example guided inductive synthesis (CEGIS) [Solar-Lezama et al. 2006]. CEGIS proceeds in iterations, where each iteration first synthesizes a candidate program that works on a finite set of inputs, and then verifies this candidate against the full specification, adding any failing inputs to the set of inputs to be considered in the next synthesis iteration. We use Probe for the synthesis phase of the CEGIS loop. At the start of each CEGIS iteration, we initialize an independent instance of Probe starting from a uniform grammar.

Baseline Solvers. As the state-of-the-art in research questions (Q2) and (Q3) we use EuPhony and CVC4, which are the state-of-the-art SyGuS solvers in terms of performance and solution quality.
EuPhony [Lee et al. 2018] also uses probabilistic models to guide its search, but unlike Probe they are pre-learned models. We used the trained models that are available in EuPhony’s repository [eup 2018]. CVC4 [Reynolds et al. 2019] has been the winner of the PBE-Strings track of the SyGuS Competition [Alur et al. 2017a] since 2017. We use the CVC4 version 1.8 (Aug 6 2020 build).

Experimental setup. All experiments were run with a 10 minute timeout for all solvers, on a commodity Lenovo laptop with a i7 quad-core CPU @ 1.90GHz with 16GB of RAM.

Fig. 13. Number of benchmarks solved by Probe and unguided search techniques (size-based and height-based enumeration) for String, BitVec and Circuit domains. Timeout is 10 min, graph scale is linear.

6.2 Q1.1: Effectiveness of Just-in-time learning
To assess the effectiveness of the just-in-time learning approach implemented in Probe, we first compare it to two unguided bottom-up search algorithms: height-based and size-based enumeration. We implement these baselines inside Probe, as simplifications of guided bottom-up search.

Results for String Domain. We measure the time to solution for each of the 82 benchmarks in the String benchmark set, for each of the three methods: Probe, size-based, and height-based enumeration. The results are shown in Fig. 13a. Probe, size-based and height-based are able to
solve 50, 42 and 9 problems, respectively. Additionally, at every point after one second, PROBE has solved more benchmarks than either size-based or height-based enumeration.

**Just-in-time learning and grammar size.** In addition to our regular benchmark suite, we created a version of the STRING benchmarks (except 12 outliers that have abnormally many string literals) that uses an extended string grammar, which includes all operations and literals from all STRING benchmarks. In total this grammar has all available string, integer and boolean operations in the SyGuS language specification and 48 string literals and 11 integer literals. These 70 extended-grammar benchmarks allow us to test the behavior of PROBE on larger grammars and thereby larger program spaces.

Within a timeout of 10 minutes, PROBE solves 25 benchmarks (half the original amount) whereas height-based and size-based enumeration solved 1 (11% of original) and 9 (21% of original) benchmarks respectively as shown in Fig. 13b. We find this particularly encouraging, because the size of the grammar usually has a severe effect on the synthesizer (as we can see for size-based enumeration), so much so that carefully constructing a grammar is considered to be part of synthesizer design. While the baseline synthesizers need the benefit of approaching each task with a different, carefully chosen, grammar, PROBE’s just-in-time learning is much more robust to additional useless grammar productions. Even with a larger grammar, PROBE’s search space does not grow as much: once it finds a partial solutions, it hones in on the useful parts of the grammar.

**Results for BitVec Domain.** The results for the BitVec benchmarks are shown in Fig. 13c. Out of the 27 BitVec benchmarks, size-based and PROBE are able to solve 20 benchmarks each, whereas height-based solves 13 benchmarks. Although size-based enumeration and PROBE solve the same number of benchmarks, we can see from the graph that PROBE is faster in terms of synthesis time.

**Results for Circuit Domain.** The results for the Circuit benchmarks are shown in Fig. 13d. Out of the 31 Circuit benchmarks, PROBE solves 17 whereas size-based and height-based enumeration each solve 22 benchmarks. The reason PROBE performs worse that both baselines on this domain is that the Circuit grammar is very small (only four operations in the median case) and the solutions tend to use most of productions from the grammar. Thus, rewarding specific productions in the PCFG does not yield significant benefits, but in fact the search is slowed down due to the restarting overhead incurred by PROBE.

**Summary of results.** Out of the 210 benchmarks from three different domains and the extended STRING grammar, PROBE solves 112, size-based solves 93 and height-based solves 45. We conclude that overall, PROBE outperforms both baseline techniques, and is therefore an effective synthesis technique.

### 6.3 Q1.2: Selection of partial solutions

In this section, we empirically evaluate the schemes for selecting promising partial solutions. We compare four different schemes: the three described in Sec. 5.2 and the baseline of using ALL generated partial solutions. The results are shown in Fig. 14.

With a timeout of 10 minutes, the ALL baseline scheme performs consistently worse than the other schemes on all three domains (and also worse than unguided size-based enumeration). For the circuit domain (Fig. 14c), the ALL scheme solves none of the benchmarks. The performance of the remaining schemes is very similar, indicating that the general idea of leveraging small and semantically unique partial solutions to guide search is robust to minor changes in the selection criteria. We select First Cheapest as the scheme used in PROBE since it provides a balance between rewarding few partial solutions while still considering syntactically different approaches. The performance of First Cheapest varies slightly depending on the domain. For instance, on Circuit
benchmarks, PCFG updates are actually detrimental to performance, so the LARGEST SUBSET scheme, which selects the fewest solutions, performs the best.

6.4 Q2: Is PROBE faster than the state of the art?

We compare PROBE’s time to solution on the benchmarks in our suite against two state-of-the-art SyGuS solvers, EuPHONY and CVC4. The results for all three domains are shown in Fig. 15.

**STRING Domain.** Results for the STRING domain are shown in Fig. 15a. Of the 82 benchmarks in the STRING suite, PROBE solves 50 benchmarks, with an average time of 45.1s and a median time of 10s. EuPHONY solves 23 benchmarks, with average of 15.4s and a median of 0.7s. CVC4 solves 75 benchmarks, with an average of 51.8s and a median of 10.2s.

The performance of EuPHONY is close to that reported originally by Lee et al. [2018]: they report 27 of the 82 benchmarks solved with a 60 minute timeout. Even with the reduced timeout, PROBE vastly outperforms EuPHONY.
When only examining time to solution, CVC4 outperforms PROBE: not only it solves more benchmarks faster, but it still solves new benchmarks long after PROBE and EUPhONY have plateaued. However, these solutions are not necessarily usable, as we show in Sec. 6.5.

**BitVec Domain.** Out of the 27 BitVec benchmarks, PROBE solves 20 benchmarks, EUPhONY solves 14 and CVC4 solves 13 benchmarks as shown in Fig. 15b. PROBE outperforms both CVC4 and EUPhONY on these benchmarks with an average time of 6.4s and median time of 2.3s. EUPhONY’ average time is 52s and median is 4.6s while CVC4 takes an average of 58.1s and a median of 15.4s. PROBE not only solves the most benchmarks overall, it also solves the highest number of benchmarks compared to EUPhONY and CVC4 at each point in time.

We should note that the EUPhONY model we used for this experiment was trained on the EUPhONY set of bit-vector benchmarks (the ones we excluded because of the case-splits) rather than the Hacker’s Delight benchmarks. Although EUPhONY does very well on its own bit-vector benchmarks, it does not fare so well on Hacker’s Delight. These results shed some light on how
brittle pre-trained models are in the face of subtle changes in syntactic program properties, even within a single bit-vector domain; we believe this makes a case for just-in-time learning.

**Circuit Domain.** Out of the 31 Circuit benchmarks, Probe solves 17 benchmarks with an average time of 235.1s and median time of 128.6s, as shown in Fig. 15c. EuPhony solves 13 benchmarks with an average time of 193.6s and median time of 36.1s. CVC4 outperforms Probe by solving 19 benchmarks with average and median times of 89s and 41s. But similarly to String domain, CVC4 generates much larger solutions than Probe, as discussed in Sec. 6.5.

**Summary of results.** Of the total 140 benchmarks, Probe solves 87 within the 10-minute timeout, EuPhony solves 50, and CVC4 solves 107. Probe outperforms EuPhony’s pre-learned models in all three domains, and while CVC4 outperforms Probe in both the String and Circuit domains, the next subsection will discuss the quality of the results it generates.

### 6.5 Q3: Quality of synthesized solutions

So far, we have tested the ability of solvers to arrive at a solution, without checking what the solution is. When a PBE synthesizer finds a program for a given set of examples, it guarantees nothing but the behavior on those examples. Indeed, the SyGuS Competition scoring system\(^9\) awards the most points (five) for simply returning any program that matches the given examples. It is therefore useful to examine the *quality* of the solutions generated by Probe and its competition.

Size is a common surrogate measure for program simplicity: e.g., the SyGuS Competition awards an additional point to the solver that returns the smallest program for each benchmark. Program size reflects two sources of complexity: (i) unnecessary operations that do not influence the result, and, perhaps more importantly, (ii) case splitting that overfits to the examples. It is therefore reasonable to assume that a smaller is more interpretable and generalizes better to additional inputs beyond the initial input-output examples.

Based on these observations, we first estimate the quality of results for all three domains by comparing the sizes of solutions generated by Probe and other tools. We next focus on the String benchmarks, as this is the only domain where the specification is given in the form of input-output examples, and hence is prone to overfitting. For this domain, we additionally measure the amount of case splitting in generated solutions and test their generalization accuracy on unseen inputs.

#### Size of generated solutions.

Fig. 16 shows the sizes of Probe solutions in AST nodes, as compared with size-based enumeration (which always returns the smallest solution by definition), and well as EuPhony and CVC4. Each comparison is limited to the benchmarks both tools can solve.

**String domain.** First, we notice in Fig. 16a that Probe sometimes finds larger solutions than size-based enumeration, but the difference is small. Likewise, Fig. 16b shows that EuPhony and Probe return similar-sized solutions. Probe returns the smaller solutions for 10 benchmarks, but the difference is not large. On the other hand, CVC4 solutions (Fig. 16c) are larger than Probe’s on 43 out of 47 benchmarks, sometimes by as much as *two orders of magnitude*. For the remaining four benchmarks, solution sizes are equal. On one of the benchmarks not solved by Probe (and therefore not in the graph), CVC4 returns a result with over 7100(!) AST nodes.

**Other domains.** Fig. 16d shows that on the BitVec domain Probe always finds the minimal solution. Solutions by EuPhony (Fig. 16e) and CVC4 (Fig. 16f) are slightly larger\(^10\) in one (resp. three) cases. Hence, on this domain Probe outperforms its competitors with respect to *both* synthesis time and solution size. For the Circuit benchmarks, Probe again finds minimal solutions, as shown

---


\(^10\)Note that we use linear scale for BitVec and Circuit as opposed to logarithmic scale for String.
in Fig. 16g. Both EuPhony (Fig. 16h) and CVC4 (Fig. 16i) generate larger solutions for all of the commonly solved benchmarks.

**Case splitting.** So why are the CVC4 String programs so large? Upon closer examination, we determined that they perform over-abundant case splitting, which hurts both readability and generality. To confirm our intuition, we count the number of if-then-else operations (ite) in the programs synthesized by Probe and by CVC4. The results are plotted in Fig. 17a. The number of ites is normalized by number of examples in the task specification. Probe averages 0.01 ite per example (for all but one benchmark Probe solutions do not contain an ite), whereas CVC4
Just-in-Time Learning for Bottom-up Enumerative Synthesis

(a) Number of ite operations per examples

<table>
<thead>
<tr>
<th>Test Benchmark</th>
<th>Training Examples</th>
<th>Testing Examples</th>
<th>PROBE Accuracy</th>
<th>CVC4 Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>initials-long</td>
<td>4</td>
<td>54</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>phone-5-long</td>
<td>7</td>
<td>100</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>phone-6-long</td>
<td>7</td>
<td>100</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>phone-7-long</td>
<td>7</td>
<td>100</td>
<td>100%</td>
<td>7%</td>
</tr>
<tr>
<td>phone-10-long</td>
<td>7</td>
<td>100</td>
<td>100%</td>
<td>57%</td>
</tr>
<tr>
<td>phone-9-long</td>
<td>7</td>
<td>100</td>
<td>N/A</td>
<td>7%</td>
</tr>
<tr>
<td>univ_4-long</td>
<td>8</td>
<td>20</td>
<td>N/A</td>
<td>73.6%</td>
</tr>
<tr>
<td>univ_5-long</td>
<td>8</td>
<td>20</td>
<td>N/A</td>
<td>68.4%</td>
</tr>
<tr>
<td>univ_6-long</td>
<td>8</td>
<td>20</td>
<td>N/A</td>
<td>100%</td>
</tr>
<tr>
<td><strong>Avg Accuracy</strong></td>
<td></td>
<td></td>
<td><strong>100%</strong></td>
<td><strong>68.1%</strong></td>
</tr>
</tbody>
</table>

(b) Generalization accuracy on unseen inputs

Fig. 17. Fig. 17a displays the number of ite operations per example for STRING benchmarks solved by PROBE and CVC4. CVC4 has a large number of case splits as indicated. Fig. 17b shows the generalization accuracy on unseen inputs for the 9 test benchmarks.

averages 0.41 ites per example. When also considering benchmarks PROBE cannot solve, some benchmarks have more than two ites per example.

**Generalization Accuracy.** Finally, we test the generality of the synthesized programs—whether they generalize well to additional examples, or in other words, whether synthesis returns reusable code. Concretely, we measure generalization accuracy [Alpaydin 2014], the percentage of unseen inputs for which a generated program produces the correct output. To this end, we find a solution using PROBE and CVC4, and then test it on additional examples for the same program.

Since most benchmarks in our suite contain only a few input-output examples, splitting these examples into a training and testing set would render most benchmarks severely underspecified. Instead we turn to the subset of STRING benchmarks from the SyGuS Competition PBE-Strings suite. These are benchmark pairs where each task appears in a "short" form with a small number of examples and a “long” form with with additional examples, but both represent the same task and share the same grammar. There are nine such benchmark pairs in this suite.

We compare the generalization accuracy of CVC4 and PROBE by using the short benchmark of each pair to synthesize a solution, and, if a solution is found, we test it on the examples of the long version of the benchmark to see how well it generalizes. The results are shown in Fig. 17b.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Solution generated</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>stackoverflow1.sl</td>
<td>(substr arg 0 (indexof arg (concat &quot; &quot; &quot;Inc&quot;) 1))</td>
<td>1.4s</td>
</tr>
<tr>
<td>stackoverflow3.sl</td>
<td>(substr arg (- (to.int (concat &quot;1&quot; &quot;9&quot;)) 2) (len arg))</td>
<td>2.1s</td>
</tr>
<tr>
<td>stackoverflow8.sl</td>
<td>(replace (substr arg (+ 2 (+ 4 4)) (len arg)) &quot;/&quot; &quot;)</td>
<td>10.5s</td>
</tr>
<tr>
<td>stackoverflow10.sl</td>
<td>(substr arg (indexof (replace arg &quot;&quot; (to.str (len arg))) 4) 4)</td>
<td>17.6s</td>
</tr>
<tr>
<td>exceljet1.sl</td>
<td>(substr arg (+ (indexof arg &quot;.&quot; 1)) (len arg))</td>
<td>1.5s</td>
</tr>
<tr>
<td>exceljet2.sl</td>
<td>(substr arg (indexof (concat &quot;.&quot; arg) &quot;.&quot;) (len arg) (indexof arg &quot;.&quot; 1)) (len arg))</td>
<td>25.1s</td>
</tr>
<tr>
<td>initials.sl</td>
<td>(concat (concat (at name 0) &quot;.&quot;) (concat (at name (+ (indexof name &quot;.&quot; 0)) &quot;.&quot;)) )</td>
<td>126.9s</td>
</tr>
<tr>
<td>phone-6-long.sl</td>
<td>(substr name (- (indexof name &quot;.&quot; 4) 3) 3)</td>
<td>11.6s</td>
</tr>
<tr>
<td>43606446.sl</td>
<td>(substr arg (- (len arg) (+ (+ 1 1) (+ 1 1)) ) (+ (+ 1 1) 1))</td>
<td>12.6s</td>
</tr>
<tr>
<td>11604909.sl</td>
<td>(substr (concat &quot;.&quot; arg) (indexof arg &quot;.&quot; 1) (+ (+ 1 1) 1)) )</td>
<td>20.3s</td>
</tr>
</tbody>
</table>

Fig. 18. PROBE solutions for 10 randomly selected benchmarks out of the 50 benchmarks PROBE solves from the [eup 2018] STRING testing set. Time indicates the synthesis time in seconds.
Fig. 19. PROBE solutions for 5 randomly selected benchmarks out of the 20 benchmarks PROBE solves from the Hacker’s Delight BitVec set, Time indicates the synthesis time in seconds.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Solution generated</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hd-11.sl</td>
<td>(bvult y (bvand x (bvnot y)))</td>
<td>2.4s</td>
</tr>
<tr>
<td>hd-09.sl</td>
<td>(bvsub x (bvshl (bvand (bvashr x #x000000000000001f) x) #x0000000000000001))</td>
<td>8.1s</td>
</tr>
<tr>
<td>hd-15.sl</td>
<td>(bvsub (bvor x y) (bvlsr (bvxor x y) #x0000000000000001))</td>
<td>16.8s</td>
</tr>
<tr>
<td>hd-18.sl</td>
<td>(bvult (bvxor (bvneg x) x) (bvneg x))</td>
<td>2.3s</td>
</tr>
<tr>
<td>hd-13.sl</td>
<td>(bvor (bvashr x #x0000000000000001f) (bvlsr (bvneg x) #x0000000000000001f))</td>
<td>3.5s</td>
</tr>
</tbody>
</table>

Fig. 20. PROBE solutions for 5 randomly selected benchmarks out of the 17 benchmarks PROBE solves from the [eup 2018] Circuit set, Time indicates the synthesis time in seconds.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Solution generated</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CrCy_10-sbox2-D5-sIn14.sl</td>
<td>(xor (xor (and (xor LN17 LN16) LN4) LN200) LN61)</td>
<td>6.4s</td>
</tr>
<tr>
<td>CrCy_10-sbox2-D5-sIn88.sl</td>
<td>(xor LN73 (and (xor (and LN70 (xor LN236 (xor LN252 LN253))) LN71) LN74))</td>
<td>353.1s</td>
</tr>
<tr>
<td>CrCy_10-sbox2-D5-sIn78.s</td>
<td>(and LN77 (and (xor (and LN70 (xor LN253 (xor LN236 LN252))) LN71))</td>
<td>24.8s</td>
</tr>
<tr>
<td>CrCy_10-sbox2-D5-sIn80.s</td>
<td>(xor (and (xor (xor LN252 LN236) LN253) LN70) LN71))</td>
<td>2.7s</td>
</tr>
<tr>
<td>CrCy_8-P12-D5-sIn1.s</td>
<td>(xor (xor (xor LN3 LN81) (xor LN7 (xor LN75 LN78))) k4)</td>
<td>193.5s</td>
</tr>
</tbody>
</table>

The first part of the table shows the benchmarks where PROBE finds a solution. As discussed above, PROBE rarely finds solutions with case splits, so it is not surprising that once it finds a program, that program is not at all overfitted to the examples.

Solutions found by CVC4 generalize with 100% accuracy in 4 out of the 9 benchmark pairs. In two of the benchmarks, the accuracy of CVC4 solutions is only 7%, or precisely the 7 training examples out of the 100-example test set, representing a complete overfitting to the training examples. On average, CVC4 has 68% generalization accuracy on these benchmark pairs. Even though this experiment is small, it provides a glimpse into the extent to which CVC4 solutions sometimes overfit to the examples.

**Sample solutions.** Finally, we examine a few sample solutions generated by PROBE in Fig. 18 for STRING domain, Fig. 19 for BitVec domain and Fig. 20 for the Circuit domain. Even though the SyGuS language is unfamiliar to most readers, we believe that these solutions should appear simple and clearly understandable. In comparison, the CVC4 solutions to these benchmarks are dozens or hundreds of operations long.

**Solution quality.** The experiments in this section explored solution quality via three empirical measures: solution size, the amount of case-splitting, and the ability of solutions to generalize to new examples for the same task. These results show conclusively that, while CVC4 is considerably faster than PROBE, and solves more benchmarks, the quality of its solutions is significantly worse.

### 6.6 Conclusions

In conclusion, we have shown that PROBE is faster and solves more benchmarks than unguided enumerative techniques, which confirms that just-in-time learning is an improvement on a baseline synthesizer. We have also shown that PROBE is faster and solves more benchmarks than EuPhony, a probabilistic synthesizer with a pre-learned model, based on top-down enumeration. Finally, we have explored the quality of synthesized solutions via size, case splitting, and generalizability, and found that even though CVC4 solves more benchmarks than PROBE, its solutions to example-based benchmarks overfit to the examples, and are therefore neither readable nor reusable; in contrast, PROBE’s solutions are small and generalize perfectly.
7 RELATED WORK

**Enumerative Program Synthesis.** Despite their simplicity, enumerative program synthesizers are known to be very effective: ESolver [Alur et al. 2013a] and EUSolver [Alur et al. 2017b] have been past winners of the SyGuS competition [Alur et al. 2016, 2017a]. Enumerative synthesizers typically explore the space of programs either top-down, by extending a partial program tree from the node towards the leaves [Alur et al. 2017b; Kalyan et al. 2018; Koukoutos et al. 2017; Lee et al. 2018], or bottom-up, by gradually building up a program tree from the leaves towards the root [Albarghouthi et al. 2013; Alur et al. 2013a; Peleg and Polikarpova 2020; Udupa et al. 2013]. These two strategies have complementary strengths and weaknesses, similarly to backward chaining and forward chaining in proof search.

One important advantage of bottom-up enumeration is the ability to prune the search space using observational equivalence (OE), i.e. discard a program that behaves equivalently to an already enumerated program on the set of inputs from the semantic specification. OE was first proposed in [Albarghouthi et al. 2013; Udupa et al. 2013] and since then has been successfully used in many bottom-up synthesizers [Alur et al. 2018; Peleg and Polikarpova 2020; Wang et al. 2017a], including Probe. Top-down enumeration techniques cannot fully leverage OE, because incomplete programs they generate cannot be evaluated on the inputs. Instead, these synthesizers prune the search space based on other syntactic and semantic notions of program equivalence: [Frankle et al. 2016; Gvero et al. 2013; Osera and Zdancewic 2015] only produce programs in a normal form; [Feser et al. 2015] apply a constant set of semantic equivalence reductions; finally, EuPHONY [Lee et al. 2018] employs a weaker version of OE for incomplete programs, which compares their complete parts observationally and their incomplete parts syntactically.

**Guiding Synthesis with Probabilistic Models.** Recent years have seen proliferation of probabilistic models of programs [Allamanis et al. 2018], which can be used, in particular, to guide program synthesis. The closest to ours are the techniques for guiding enumerative search using probabilistic grammars, such as PCFGs [Balog et al. 2016; Koukoutos et al. 2017; Menon et al. 2013], and PHOGs [Lee et al. 2018]. DeepCoder [Balog et al. 2016] proposes several simple schemes for using a PCFG to prune and prioritize the search, but these schemes are ad-hoc in the sense that they do not guarantee completeness and/or enumeration in the order of (approximately) decreasing likelihood. The other techniques use the model to guide top-down enumeration, using either Dijkstra’s shortest path algorithm [Menon et al. 2013] or its A* variation [Koukoutos et al. 2017; Lee et al. 2018]. In contrast to these approaches, Probe uses a PCFG to guide bottom-up enumeration with observational equivalence reduction, and it empirically performs better than the guided top-down search technique of [Lee et al. 2018].

**Learning Probabilistic Models.** Approaches to learning probabilistic models of programs can be classified into broad categories: pre-training using large amounts of data and learning on the fly. Pre-trained models require a large existing corpus of data for training. [Menon et al. 2013] learn a parameterized PCFG using a training corpus of input-output examples and their gold-standard solutions. [Bielik et al. 2016; Lee et al. 2018] increase the context of PCFGs by annotating them with a program learned from the training data. [Balog et al. 2016] learn a probability distribution over the space of possible productions by assigning weights using a training corpus of task specifications. [Kalyan et al. 2018] also assigns weights to production rules by invoking the neural network on the partial search state. While these approaches require training a model from a large corpus of preprocessed and high-quality data, Probe learns a probabilistic model just-in-time from partial solutions and needs no additional training data.

[Chen et al. 2020; Ellis et al. 2020] are related to the just-in-time approach of Probe in the sense that they update their probabilistic model on the fly. [Chen et al. 2020] employs a reinforcement learning approach to update the model with new data.
learning framework where the initial policy (model) is trained off-line and then updated on the fly during search using feedback from a logical deduction engine. PROBE can also be initialized with a weighted PCFG obtained by training over a corpus of data, but it is effective even without one; in the case of [Chen et al. 2020], it is unclear how important the initial training is for performance. [Ellis et al. 2020] learns a probabilistic model from full solutions to a subset of synthesis problems from a corpus; whereas PROBE learns a problem-specific model from partial solutions to a single synthesis problem.

**Leveraging Partial Solutions to Guide Synthesis.** [Perelman et al. 2014; Shi et al. 2019] are component-based synthesis techniques that leverage information from partial solutions to generate new programs. [Perelman et al. 2014] explicitly requires the user to arrange input-output examples in the order of increasing difficulty, and then synthesizes a sequence of programs, where \( i \) th program passes the first \( i \) examples. Each following program is not synthesized from scratch, but rather by modifying the previous program; hence intermediate programs serve as “stepping stones” for synthesis. PROBE puts less burden on the user: it does not require the examples to be arranged in a sequence, and instead identifies partial solutions that satisfy any subset of examples.

Similarly to PROBE, FrANGEL [Shi et al. 2019] leverages partial solutions that satisfy any subset of the example specification. FrANGEL generates new programs by randomly combining fragments from partial solutions. Our approach is similar to [Perelman et al. 2014; Shi et al. 2019] in that it guides the search using syntactic information learned from partial solutions, but we achieve that by updating the weights of useful productions in a probabilistic grammar and using it to guide bottom-up enumerative search.

Our previous work [Peleg and Polikarpova 2020] proposes a technique to accumulate multiple partial solutions during bottom-up enumerative synthesis with minimum overhead. PROBE is a natural extension of [Peleg and Polikarpova 2020] since it leverages these partial solutions found during enumeration to guide search.

During top-down enumeration, [Koukoutos et al. 2017] employs an optimization strategy where the cost of an incomplete (partial) program is lowered if it satisfies some of the examples. This optimization encourages the search to complete a partial program that looks promising, but unlike PROBE, offers no guidance on which are the likely productions to complete it with.

### 8 CONCLUSION AND FUTURE WORK

We have presented a new program synthesis algorithm we dub *guided bottom-up search with just-in-time-learning*. This algorithm combines the pruning power of observational equivalence with guidance from probabilistic models. Moreover, our just-in-time learning is able to bootstrap a probabilistic model during synthesis by leveraging partial solutions, and hence does not require training data, which can be hard to obtain.

We have implemented this algorithm in a tool called PROBE that works with a popular SyGuS input format. We evaluated PROBE on 140 synthesis benchmarks from three different domains. Our evaluation demonstrates that PROBE is more efficient than unguided enumerative search and a state-of-the-art guided synthesizer EUFONY, and while PROBE is less efficient than CVC4, our solutions are of higher quality.

In future work, we are interested in instantiating PROBE in new application domains. We expect just-in-time learning to work for other programs over structured data structures, e.g. lists and tree transformations. Just-in-time learning also requires that example specifications cover a range from simple to more complex, so that PROBE can discover short partial solutions and learn from them. Luckily, users seem to naturally provide examples that satisfy this property, as indicated by SyGuS benchmarks whose specifications are taken from StackOverflow. Generalizing these observations
is an exciting direction for future work. Another interesting direction is to consider PROBE in the context of program repair, where similarity to the original faulty program can serve as a prior to initialize the PCFG.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their feedback on the draft of this paper. This work was supported by the National Science Foundation under Grants No. 1955457, 1911149, and 1943623.

REFERENCES


APPENDIX

\[
\begin{align*}
\text{Start} & \rightarrow S \\
S & \rightarrow \text{arg0} | \text{arg1} | \ldots \\
& | \ \text{lit}-1 | \text{lit}-2 | \ldots \quad \text{string variables and literals} \\
& | (\text{replace } S S S) \quad \text{replace } s \times y \text{ replaces first occurrence of } x \text{ in } s \text{ with } y \\
& | (\text{concat } S S) \quad \text{concat } x \ y \text{ concatenates } x \text{ and } y \\
& | (\text{substr } S I I) \quad \text{substr } x \ y \ z \text{ extracts substring of length } z, \text{ from index } y \\
& | (\text{ite } B S S) \quad \text{ite } x \ y \ z \text{ returns } y \text{ if } x \text{ is true, otherwise } z. \\
& | (\text{int.to.str } I) \quad \text{int.to.str } x \text{ converts } x \text{ to a string} \\
& | (\text{at } S I) \quad \text{at } x \ y \text{ returns the character at index } y \text{ in string } x \\
B & \rightarrow \text{true} | \text{false} \quad \text{bool literals} \\
& | (= I I) \quad = x \ y \text{ returns true if } x \text{ equals } y \\
& | (\text{contains } S S) \quad \text{contains } x \ y \text{ returns true if } x \text{ contains } y \\
& | (\text{suffixof } S S) \quad \text{suffixof } x \ y \text{ returns true if } x \text{ is the suffix of } y \\
& | (\text{prefixof } S S) \quad \text{prefix } x \ y \text{ returns true if } x \text{ is the prefix of } y \\
I & \rightarrow \text{arg0} | \text{arg1} | \ldots \\
& | \ \text{lit}-1 | \text{lit}-2 | \ldots \quad \text{int variables and literals} \\
& | (\text{str.to.int } S) \quad \text{str.to.int } x \text{ converts } x \text{ to a int} \\
& | (+ I I) \quad + x \ y \text{ sums } x \text{ and } y \\
& | (- I I) \quad - x \ y \text{ subtracts } y \text{ from } x \\
& | (\text{length } S) \quad \text{length } x \text{ returns length of } x \\
& | (\text{ite } B I I) \quad \text{ite } x \ y \ z \text{ returns } y \text{ if } x \text{ is true, otherwise } z. \\
& | (\text{indexof } S S I) \quad \text{indexof } x \ y \ z \text{ returns index of } y \text{ in } x, \text{ starting at index } z
\end{align*}
\]

Fig. 21. The full \textit{SyGuS} \textit{STRING} grammar of the \textit{EuPhony} benchmark suite. Integer and string variables and constants change per benchmark. Some benchmark files contain a reduced grammar.
\[ \text{Start} \rightarrow BV \]

\[ BV \rightarrow \text{arg0} | \text{arg1} \ldots \]

- \( \text{lit-1} | \text{lit-2} \ldots \) \hspace{0.5cm} \text{bit-vector variables and literals}
- \( \text{xor BV BV} \) \hspace{0.5cm} \text{xor } x \text{ y performs bitwise xor between } x \text{ and } y
- \( \text{and BV BV} \) \hspace{0.5cm} \text{and } x \text{ y performs bitwise and operation between } x \text{ and } y
- \( \text{or BV BV} \) \hspace{0.5cm} \text{or } x \text{ y performs bitwise or operation between } x \text{ and } y
- \( \text{neg BV} \) \hspace{0.5cm} \text{neg } x \text{ returns the two’s complement of } x.
- \( \text{not BV} \) \hspace{0.5cm} \text{not } x \text{ returns the one’s complement of } x.
- \( \text{add BV BV} \) \hspace{0.5cm} \text{add } x \text{ y adds } x \text{ and } y
- \( \text{mul BV BV} \) \hspace{0.5cm} \text{mul } x \text{ y multiplies } x \text{ and } y
- \( \text{udiv BV BV} \) \hspace{0.5cm} \text{udiv } x \text{ y returns the unsigned quotient of dividing } x \text{ by } y
- \( \text{urem BV BV} \) \hspace{0.5cm} \text{urem } x \text{ y returns the unsigned remainder of dividing } x \text{ by } y
- \( \text{lshr BV BV} \) \hspace{0.5cm} \text{lshr } x \text{ y returns the result of logically shifting } x \text{ to the right by } y \text{ bits.}
- \( \text{ashr BV BV} \) \hspace{0.5cm} \text{ashr } x \text{ y arithmetically shift } x \text{ to the right by } y \text{ bits.}
- \( \text{shl BV BV} \) \hspace{0.5cm} \text{shl } x \text{ y returns the result of logically shifting } x \text{ to the left by } y \text{ bits.}
- \( \text{sdv BV BV} \) \hspace{0.5cm} \text{sdv } x \text{ y returns the signed quotient of dividing } x \text{ by } y
- \( \text{srem BV BV} \) \hspace{0.5cm} \text{srem } x \text{ y returns the signed remainder of dividing } x \text{ by } y
- \( \text{sub BV BV} \) \hspace{0.5cm} \text{sub } x \text{ y subtracts } y \text{ from } x
- \( \text{ite B BV BV} \) \hspace{0.5cm} \text{ite } x \text{ y } z \text{ returns } y \text{ if } x \text{ is true, otherwise } z.

\[ B \rightarrow \text{true} | \text{false} \]

- \( \text{true} \)
- \( \text{false} \)
- \( \text{=} BV BV \) \hspace{0.5cm} \text{=} x \text{ y returns true if } x \text{ equals } y
- \( \text{ult BV BV} \) \hspace{0.5cm} \text{ult } x \text{ y returns true if } x \text{ is unsigned less than } y
- \( \text{ule BV BV} \) \hspace{0.5cm} \text{ule } x \text{ y returns true if } x \text{ is unsigned less than equal to } y
- \( \text{slt BV BV} \) \hspace{0.5cm} \text{slt } x \text{ y returns true if } x \text{ is signed less than } y
- \( \text{sle BV BV} \) \hspace{0.5cm} \text{sle } x \text{ y returns true if } x \text{ is signed less than equal to } y
- \( \text{ugt BV BV} \) \hspace{0.5cm} \text{ugt } x \text{ y returns true if } x \text{ unsigned greater than } y
- \( \text{redor BV} \) \hspace{0.5cm} \text{redor } x \text{ y performs bit-wise or reduction of } x
- \( \text{and BV BV} \) \hspace{0.5cm} \text{and } x \text{ y returns the logical and of } x \text{ and } y
- \( \text{or BV BV} \) \hspace{0.5cm} \text{or } x \text{ y returns the logical or of } x \text{ and } y
- \( \text{not BV} \) \hspace{0.5cm} \text{not } x \text{ returns the logical not of } x \text{ and } y

Fig. 22. The full SyGuS BrvVec grammar; variables and constants change per benchmark. Some of the Hacker’s Delight benchmarks contain a reduced grammar; required constants are provided. The EuPhony benchmark suite provides a subset of this grammar; it does not contain the comparison operators ult, ule, slt, sle, ugt and also the reduction operator redor. These benchmarks provide bit-vector constants from 0 to 16.
Fig. 23. The full SyGuS 

\[
\text{Start} \rightarrow B
\]
\[
B \rightarrow (\text{and } B \ B) \quad \text{and } x \ y \text{ returns the logical and of } x \text{ and } y
\]
\[
| \quad (\text{not } B) \quad \text{not } x \text{ returns the one’s complement of } x.
\]
\[
| \quad (\text{or } B \ B) \quad \text{or } x \ y \text{ returns the logical or of } x \text{ and } y
\]
\[
| \quad (\text{xor } B \ B) \quad \text{xor } x \ y \text{ returns the logical xor of } x \text{ and } y
\]
\[
| \quad \text{lit-1} \ | \ \text{lit-2} \ | \ldots \quad \text{boolean literals}
\]

CIRCUIT grammar of the EuPHONY benchmark suite. Variables and the depth of the grammar change per benchmark.