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

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

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.

CCS Concepts: • **Software and its engineering → Domain specific languages**; **Programming by example**.

Additional Key Words and Phrases: Program Synthesis, Probabilistic models, Domain-specific languages

ACM Reference Format:
Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-Time Learning for Bottom-Up Enumerative Synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 227 (November 2020), 29 pages. https://doi.org/10.1145/3428295

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.

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 synthesis is scaling the search to complex programs: as the synthesizer considers longer programs,

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

2475-1421/2020/11-ART227
https://doi.org/10.1145/3428295

Input		Output
s	n	
"1/17/16-1/18/17"	1	"1/17/16"
"1/17/16-1/18/17"	2	"1/18/17"
"01/17/2016-01/18/2017"	1	"01/17/2016"
"01/17/2016-01/18/2017"	2	"01/18/2017"

Fig. 1. Input-output example specification for the `pick-date` benchmark (adapted from [eup 2018]).

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. 2013]:

```
(substr s (indexof (concat "-" s) "-" (- n 1)) (indexof s "-" n))
```

This solution extracts the correct substring of `s` by computing its starting index `(indexof (concat "-" s) "-" (- n 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 `indexof` and `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 datasets 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 examples—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 `(substr s 0 (indexof s "-" 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 `indexof` and `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 34 seconds.

[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.

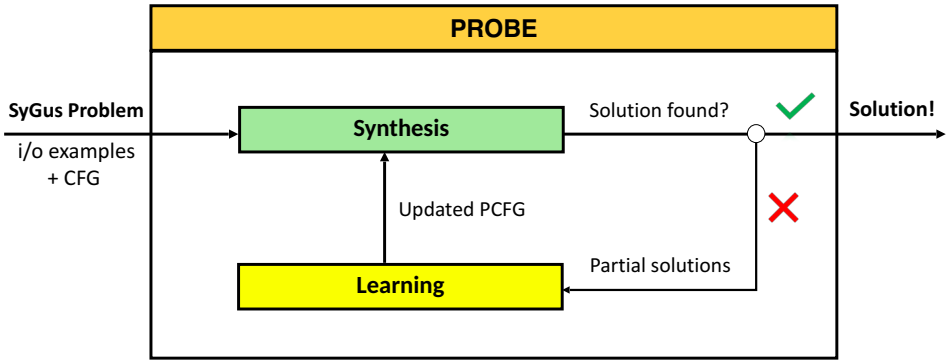


Fig. 2. Overview of the Probe system

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 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 problem in SyGuS format, *i.e.* a context-free grammar of the DSL and a set of input-output examples[2]; 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 91 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

[2]Probe 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.

ID	Input	Output
e_0	`"a < 4 and a > 0"`	`"a 4 and a 0"`
e_1	`"<open and <close>"`	`"open and close"`
e_2	`"<Change> <string> to <a> number"`	`"Change string to a number"`

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

S	$\rightarrow$	`arg` | `""` | `"<"` | `">"`	input string and string literals
	|	(`replace` S S S)	`replace s x y` replaces first occurrence of `x` in `s` with `y`
	|	(`concat` S S)	`concat x y` concatenates `x` and `y`

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

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 9 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:

(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. 2013]. 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

[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 (replace arg "<" "") "<" "") "<" "") ">" "") ">" "") ">" "")	$\{e_0, e_1, e_2\}$

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

Height	# Programs	Bank
0	4	arg, "", "<", ">"
1	15	(concat arg arg), (concat arg "<"), (concat arg ">"), (concat "<" "<"), (concat "<" ">"), . . . (replace arg "<" arg), (replace arg "<" ""), (replace arg "<" ">"), (replace arg ">" "<"), . . .
2	1023	(concat arg (concat arg arg)), (concat arg (concat ">" ">")), . . . (concat "<" (concat arg arg)), (concat "<" (replace arg "<" arg)), (concat ">" (concat "<" "<")), (concat ">" (replace arg ">" "<"))
3	$\sim 30M$	(concat arg (concat (replace arg "<" arg) arg)), (concat arg (concat (replace arg "<" arg) "<")) (concat arg (concat (replace arg "<" arg) ">")), (concat arg (concat (replace arg "<" "") arg)) ...

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

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 refer to any one of them as "the shortest solution". The figure also shows two shorter programs, replace-2 and replace-3, which satisfy different subsets of the semantic specification and which we refer to throughout this and next section.

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

Size	# Programs	Bank
1	4	`arg, "", "<", ">"`
2	0	None
3	9	`(concat arg arg), (concat arg "<"), (concat arg ">"), (concat "<" arg),` `(concat "<" "<"), (concat "<" ">"), (concat ">" arg), (concat ">" "<"), (concat ">" ">")`
4	6	`(replace arg "<" arg), (replace arg "<" "), (replace arg "<" ">")` `(replace arg ">" arg), (replace arg ">" ""), (replace arg ">" "<")`
⋮	⋮	⋮
8	349	`(concat (concat (replace arg "<" arg) arg) arg), (concat (replace arg ">" (concat ">" arg)) ">"),` `(replace (concat arg "<") (concat ">" "<") "") ... (replace (concat ">" arg) (concat ">" "<") ">")`
9	714	`(concat (concat arg arg) (concat (concat arg arg) arg)), (concat (concat "<" "<") (concat (concat ">" "<") "<")), ...` `(replace (replace arg "<" "") "<" (concat ">" ">")), (replace (replace arg ">" (concat ">" ">")) "<" ">")`
10	2048	`(concat "<" (replace (concat arg arg) (concat ">" arg) "<")),` `... (concat arg (concat (replace arg "<" (concat ">" ">")) ">"))`

Fig. 7. Programs generated for `remove-angles-short` from the grammar in Fig. 4 in the order of size.

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 $\sim 30M$ 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. 2013, 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 `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

			p_R	$-\log(p_R)$	cost_R
S	$\rightarrow$	`arg` | `""` | `"<"` | `">"`	0.188	2.41	2
	|	(`replace` S S S)	0.188	2.41	2
	|	(`concat` S S)	0.059	4.09	4

Fig. 8. A PCFG for string expressions that is biased towards the solution `replace-6`. For each production rule R, we show its probability p_R and its cost cost_R, which is computed as a rounded negative log of the probability.

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 `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 *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 [Balog et al. 2016; Lee et al. 2018; Menon et al. 2013]. 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 *probabilistic context-free grammar* (PCFG). We refer to this modification of the bottom-up algorithm as *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 `replace-6` and a low probability to the rule `concat` that does not appear in this program. As a result, this PCFG assigns a higher likelihood to the program `replace-6`[4] than it does to other programs of the same size. Hence, an algorithm that explores programs in the order of decreasing likelihood would encounter `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 *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 `concat` has a higher cost of four. The cost of a program is computed by summing up the costs of its productions, for example:

$$\begin{aligned}\text{cost}(\texttt{concat arg "<"}) &= \text{cost}(\texttt{concat}) + \text{cost}(\texttt{arg}) + \text{cost}(\texttt{"<"}) \\ &= 4 + 2 + 2 = 8\end{aligned}$$

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 `concat` x y, where

[4]The likelihood of a program is the product of the probabilities of all rules involved in its derivation.

Cost	# Programs	Bank
2	4	`arg, "", "<", ">"`
8	15	`(replace arg "<" arg), (replace arg "<" ""), (replace arg "<" ">")` `(replace arg ">" "<"), (concat "<" arg), (concat "<" "<") . . .`
20	1272	`(replace "<" (replace arg (replace arg "<" "") "") ""), (replace "<" (replace arg (replace arg "<" "") "") ">") . . .` `(replace (replace arg ">" "<") (replace arg ">" "") arg), (replace (replace arg ">" "<") (replace arg ">" "") ">")`
⋮	⋮	⋮
38	130K	`(str.replace (replace arg "<" (replace (replace arg ">" "<") ">" arg)) (replace (replace arg "<" "") ">" arg) "<")` `(replace (replace arg "<" (replace (replace arg ">" "<") ">" arg)) (replace (replace arg "<" "") ">" arg) ">") ...`

Fig. 9. Programs generated for `remove-angles` using guided bottom-up search with the PCFG in Fig. 8

the costs of x and y sum up to $8 - 4 = 4$. The cost of our solution `replace-6` is 38, which places it within the first 130K programs the search encounters; on the other hand, its size is 19, placing it within the first $\sim 4M$ programs in the order of size. As a consequence, size-based enumeration 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 from a corpus of existing solutions [Lee et al. 2018; Menon et al. 2013] (see Sec. 7 for a detailed discussion). While achieving impressive results, these approaches are computationally expensive and, more importantly, require high-quality training data, which is generally hard to obtain. 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[5]. 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

[5]The algorithm can also be initialized with a pre-learned PCFG if one is available.

Partial Solution	Examples Satisfied	PCFG costs
	$\emptyset$	arg, "", "<", ">", replace, concat $\mapsto 3$
replace-2	$\{e_0\}$	arg, "", "<", ">", replace $\mapsto 2$; concat $\mapsto 3$
replace-3	$\{e_0, e_1\}$	arg, "", "<", ">", replace $\mapsto 2$; concat $\mapsto 4$

Fig. 10. Just-in-time learning: as the search encounters partial solutions that satisfy new subsets of examples, PCFG costs are adjusted and the relative cost of concat, which is not present in the solution, increases.

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

```
replace (replace (replace (concat arg "<") "<" "") "<" "") ">" ""
```

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 23 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 $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{S}, \mathcal{R})$, where $\mathcal{N}$ denotes a finite, non-empty set of non-terminal symbols, Σ denotes a finite set of terminals, $\mathcal{S}$ denotes the starting non-terminal, and $\mathcal{R}$ is the set of production rules. In our setting, each terminal $t \in \Sigma$ is associated with an *arity* $\mathrm{arity}(t) \geq 0$, and each production rule $\mathrm{R} \in \mathcal{R}$ is of the form $\mathrm{N} \longrightarrow (t\ \mathrm{N}_1\ \ldots\ \mathrm{N}_k)$, where $\mathrm{N}, \mathrm{N}_1, \ldots, \mathrm{N}_k \in \mathcal{N}$, $t \in \Sigma$, and $\mathrm{arity}(t) = k$[6]. We denote with $\mathcal{R}(\mathrm{N})$ the set of all rules $\mathrm{R} \in \mathcal{R}$ whose left-hand side is N. A sequence $\alpha \in (\mathcal{N} \cup \Sigma)^*$ is called a *sentential form* and a sequence $s \in \Sigma^*$ is a called a *sentence*. A grammar $\mathcal{G}$ defines a (leftmost) *single-step derivation* relation on sentential forms: $s\mathrm{N}\alpha \Rightarrow s\beta\alpha$ if $\mathrm{N} \longrightarrow \beta \in \mathcal{R}$. 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.

[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.

Programs. A *program* P is a sentence derivable from some $\mathrm{N} \in \mathcal{N}$; we call a program *whole* if it is derivable from $\mathcal{S}$. The set of all programs is called the *language* of the grammar $\mathcal{G}$: $\mathcal{L}(\mathcal{G}) = \{s \in \Sigma^* \mid \mathrm{N} \Rightarrow^* s\}$. The *trace* of a program $\mathrm{tr}(P)$ is the sequence of production rules $\mathrm{R}_1, \ldots, \mathrm{R}_n$ used in its derivation ($\mathrm{N} \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_{n-1} \Rightarrow P$). The *size* of a program $|P|$ is the length of its trace. We assign semantics $[\![P]\!] : \mathrm{Val}^* \to \mathrm{Val}$ to each program P, where Val is the set of run-time values.

Inductive Syntax-Guided Synthesis. An *inductive* syntax-guided synthesis (SyGuS) problem is defined by a grammar $\mathcal{G}$ and a set of input-output examples $\mathcal{E} = \overrightarrow{\langle i, o \rangle}$, where $i \in \mathrm{Val}^*$, $o \in \mathrm{Val}$[7]. A *solution* to the problem is a program $P \in \mathcal{L}(\mathcal{G})$ such that $\forall \langle i, o \rangle \in \mathcal{E}, [\![P]\!](i) = o$. 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) $\mathcal{G}_p$ is a pair of a CFG $\mathcal{G}$ and a function $p : \mathcal{R} \to [0, 1]$ that maps each production rule $\mathrm{R} \in \mathcal{R}$ to its probability. Probabilities of all the rules for given non-terminal $\mathrm{N} \in \mathcal{N}$ sum up to one: $\forall \mathrm{N}. \sum_{\mathrm{R} \in \mathcal{R}(\mathrm{N})} p(\mathrm{R}) = 1$. 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 $p(P) = \prod_{\mathrm{R}_i \in \mathrm{tr}(P)} p(\mathrm{R}_i)$.

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

4.2 Guided Bottom-up Search Algorithm

Algorithm 1 presents our guided bottom-up search algorithm. The algorithm takes as input a PCFG $\mathcal{G}_p$ and a set of input-output examples $\mathcal{E}$, and enumerates programs in the order of increasing discrete costs according to $\mathcal{G}_p$, until it finds a program P that satisfies the entire specification $\mathcal{E}$ or reaches a certain cost limit Lim. The algorithm maintains a search state that consists of (1) the current cost level Lvl; (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 are equal to Lvl. New programs with a given cost are constructed by the auxiliary procedure New-Programs, which we describe below. In line 5, every new program P is evaluated on the inputs from the semantic specification $\mathcal{E}$; if the program matches the specification exactly, it is returned as the solution. Otherwise, if the evaluation result is already present in E, then P is deemed observationally equivalent to another program in B and discarded. A program with new behavior is added to the bank at cost Lvl and its evaluation result is cached in E; moreover, if the program satisfies some of the examples in $\mathcal{E}$, it is considered a partial solution and added to PSol.

The auxiliary procedure New-Programs takes in the PCFG $\mathcal{G}_p$, the current cost Lvl, and a bank B where all levels below the current one are fully filled. It computes the set of all programs of cost Lvl in $\mathcal{G}_p$. For the sake of efficiency, instead of returning the whole set at once, New-Programs is implemented as an *iterator*: it yields each newly constructed program lazily, and will not construct the whole set if a solution is found at cost Lvl. To construct a program of cost Lvl, the procedure iterates over all production rules $\mathrm{R} \in \mathcal{R}$. Once R is chosen as the top-level

[7] In general, the SyGuS problem allows first-order formulae as a specification, and prior work has shown how to reduce this general formulation to inductive formulation using CEGIS [Alur et al. 2017b; Lee et al. 2018].

Algorithm 1 Guided Bottom-up search algorithm

Input: PCFG $\mathcal{G}_p$, input-output examples $\mathcal{E}$, and optionally, the initial state of the search
Output: A solution P or $\bot$, and the current state of the search

1: **procedure** GUIDED-SEARCH($\mathcal{G}_p, \mathcal{E}, \langle \text{LVL}_0, \text{B}_0, \text{E}_0, \text{PSol}_0 \rangle = \langle 0, \emptyset, \emptyset, \emptyset \rangle$)
2: LVL, B, E, PSol $\leftarrow$ $\text{LVL}_0, \text{B}_0, \text{E}_0, \text{PSol}_0$ ▷ Initialize state of the search
3: **while** LVL $\leq$ LVL_0 + LIM **do**
4: **for** $P \in$ NEW-PROGRAMS($\mathcal{G}_p$, LVL, B) **do** ▷ For all programs of cost LVL
5: EVAL $\leftarrow$ $[\langle i, [\![P]\!](i)\rangle \mid \langle i, o\rangle \in \mathcal{E}]$ ▷ Evaluate on inputs from $\mathcal{E}$
6: **if** (EVAL = $\mathcal{E}$) **then**
7: **return** $(P, \langle \text{LVL}, \text{B}, \text{E}, \text{PSol}\rangle)$ ▷ P fully satisfies $\mathcal{E}$, solution found!
8: **else if** (EVAL $\in$ E) **then**
9: **continue** ▷ P is observationally equivalent to another program in B
10: **else if** (EVAL $\cap$ $\mathcal{E} \neq \emptyset$) **then** ▷ P partially satisfies $\mathcal{E}$
11: PSol $\leftarrow$ PSol $\cup$ P
12: B[LVL] $\leftarrow$ B[LVL] $\cup$ $\{P\}$ ▷ Add to the bank, indexed by cost
13: E $\leftarrow$ E $\cup$ EVAL ▷ Cache evaluation result
14: LVL $\leftarrow$ LVL + 1
15: **return** $(\bot, \langle \text{LVL}, \text{B}, \text{E}, \text{PSol}\rangle)$ ▷ Cost limit reached

Input: PCFG $\mathcal{G}_p$, cost level LVL, program bank B filled up to LVL $-$ 1
Output: Iterator over all programs of cost LVL ▷ For all production rules

16: **procedure** NEW-PROGRAMS($\mathcal{G}_p$, LVL, B)
17: **for** $(\text{R} = \text{N} \rightarrow (t\ \text{N}_1\ \text{N}_2\ \ldots\ \text{N}_k) \in \mathcal{R})$ **do**
18: **if** $\text{cost}(\text{R}) = \text{LVL} \wedge k = 0$ **then** ▷ t has arity zero
19: yield t
20: **else if** $\text{cost}(\text{R}) < \text{LVL} \wedge k > 0$ **then** ▷ t has non-zero arity
21: **for** $(c_1, \ldots, c_k) \in \left\{ [1, \text{LVL}]^k \,\middle|\, \sum c_i = \text{LVL} - \text{cost}(\text{R}) \right\}$ **do** ▷ For all subexpression costs
22: **for** $(P_1, \ldots, P_k) \in \{\, \text{B}[c_1] \times \ldots \times \text{B}[c_k] \mid \bigwedge_i \text{N}_i \Rightarrow^* P_i \,\}$ **do** ▷ For all subexpressions
23: yield $(t\ P_1\ \ldots\ P_k)$

production in the derivation of the new program, we have a budget of LVL $-$ $\text{cost}(\text{R})$ 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: $\text{N}_i \Rightarrow^* P_i$ 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 $\mathcal{G}_p = \langle \mathcal{G}, p\rangle$ and $\mathcal{E}$, if the procedure returns $(P, \_)$, 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 $C \leq \text{LVL}_0 + \text{LIM}$, then the algorithm will return $(P, \_)$, 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 LVL $= C$, either $P \in$ B or $\exists P' \in$ B s.t. $\text{cost}(P') \leq \text{cost}(P)$ and $\forall \langle i, o\rangle \in \mathcal{E}$ s.t. $[\![P]\!](i) = [\![P']\!](i)$.

Algorithm 2 The PROBE algorithm

Input: CFG $\mathcal{G}$, set of input-output examples $\mathcal{E}$
Output: A solution P or $\bot$

```
1:  procedure PROBE(G, E)
2:      G_p ← ⟨G, p_u⟩                                  ▷ Initialize PCFG to uniform
3:      Lvl, B, E ← 0, ∅, ∅                             ▷ Initialize search state
4:      while not timeout do
5:          P, ⟨Lvl, B, E, PSol⟩ ← GUIDED-SEARCH (G_p, E, ⟨Lvl, B, E, ∅⟩)   ▷ Search with current PCFG G_p
6:          if P ≠ ⊥ then
7:              return P                                ▷ Solution found
8:          PSol ← SELECT(PSol, E)                      ▷ Select promising partial solutions
9:          if PSol ≠ ∅ then
10:             G_p ← UPDATE(G_p, PSol, E)              ▷ Update the PCFG G_p
11:             Lvl, B, E ← 0, ∅, ∅                     ▷ Restart the search
12:     return ⊥
```

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 LVL), monotonicity of costs (replacing a subexpression with a more expensive one yields a more expensive program) and compositionality 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 $\mathrm{rcost}(P_1) < \mathrm{rcost}(P_2)$. 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 $\mathcal{G}, \mathcal{E}$. It starts by initializing the PCFG with CFG $\mathcal{G}$ and a uniform distribution p_u, which assigns every production rule $\mathrm{R} = \mathrm{N} \to \beta$ the probability $p(\mathrm{R}) = 1/|\mathcal{R}(\mathrm{N})|$. 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, the

ID	Input	Output
e_0	`"+95 310-537-401"`	`"310"`
e_1	`"+72 001-050-856"`	`"001"`
e_2	`"+106 769-858-438"`	`"769"`

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

Cycle	ID	Examples Satisfied	Partial Solutions	Cost
1	P_0	$\{e_0, e_1\}$	`(substr arg 4 3)`	20
2	P_1	$\{e_0, e_1\}$	`(replace (substr arg 4 3) " " arg)`	21
3	P_2	$\{e_1, e_2\}$	`(substr arg (indexof arg (at arg 5) 3) 3)`	37
3	P_3	$\{e_1, e_2\}$	`(substr arg (- 4 (to.int (at arg 4))) 3)`	37

Fig. 12. Partial solutions and the corresponding subset of examples satisfied for the problem in Fig. 11

second step is to use the promising partial solutions to Update the PCFG (line 10), and the third step is to restart the search (line 11). 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:

```
(substr arg (- (indexof arg "-" 3) 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_2 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_2 promising: indeed, `indexof` turns out to be useful in the final solution. We observe that although P_2 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 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 $\mathcal{E}$. 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 $\{P_0, P_2\}$ 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 $\{P_0, P_2, P_3\}$ 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 $\mathrm{R} \in \mathcal{R}$ using the formula:

$$p(\mathrm{R}) = \frac{p_u(\mathrm{R})^{(1-\mathrm{Fit})}}{Z} \qquad \text{where} \qquad \mathrm{Fit} = \max_{\{P \in \mathrm{PSol} \mid \mathrm{R} \in \mathrm{tr}(P)\}} \frac{|\mathcal{E} \cap \mathrm{E}[P]|}{|\mathcal{E}|}$$

where Z denotes the normalization factor, and 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 $\mathcal{G}$. 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 Lvl 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 `0` with cost 5 and the program `(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 $\mathrm{cost}(\texttt{0}) = 10$ and $\mathrm{cost}(\texttt{(indexof arg "+")}) = 7$. 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 `(indexof arg "+")` is now encountered before the program `0` and selected as the representative of it equivalence class. In other words, the desired the behavior under the new PCFG is that the class $[0, 0, 0]$ has cost 7. 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 `0` 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 `0` with cost 10. This does not solve the problem, however: now class $[0, 0, 0]$ has cost 10 instead the desired cost 7, 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. 2017c,b] 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 a 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 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 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 [eup 2018]. 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 forums. EuPhony uses 82 out of these 205 benchmarks as their testing set based on the criterion 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 formula 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.

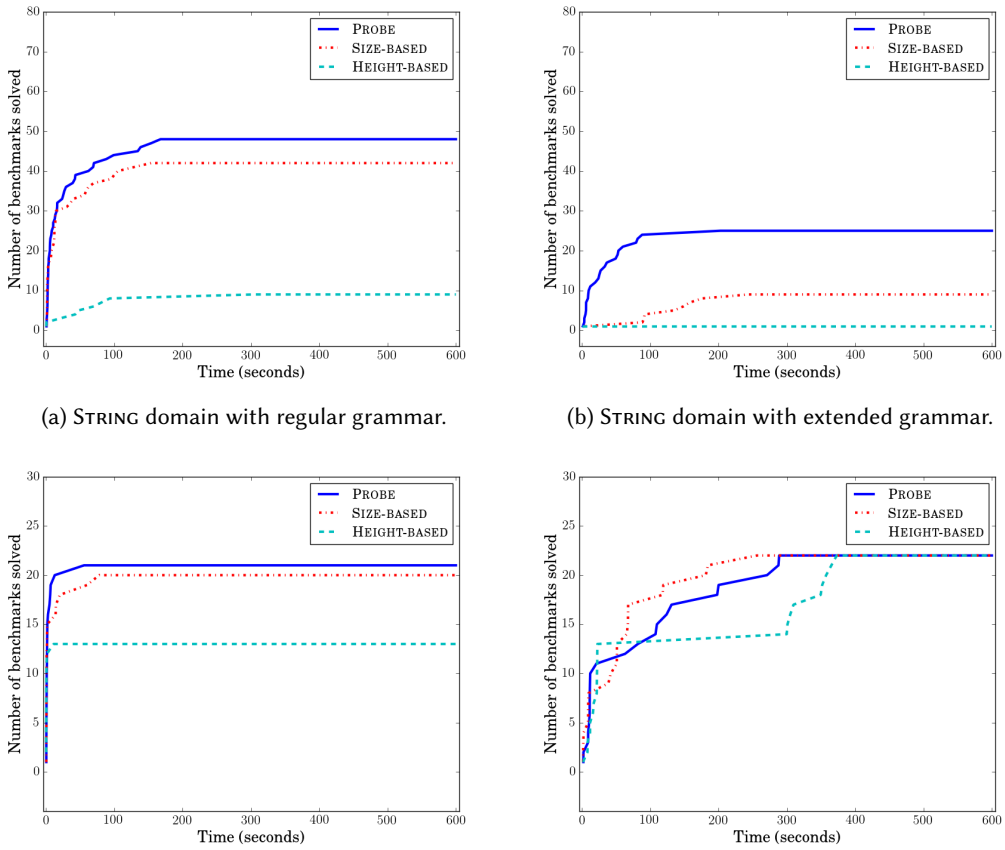
Circuit Benchmarks. The 31 Circuit benchmarks are taken from the EuPhony testing set. These benchmarks involve synthesizing constant-time circuits that are cryptographically resilient to timing attacks. Circuit benchmark grammars have a median of 4 operations, 0 literals, and 6 variables. The semantic 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.



(a) String domain with regular grammar.

(b) String domain with extended grammar.

(c) BitVec domain

(d) Circuit domain

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 enumeration

are able to solve 48, 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 (52% of the original number) 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 solution, 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, Probe, size-based and height-based solve 21, 20 and 13 benchmarks, respectively. In addition to solving one more benchmark, Probe is also considerably faster than size-based enumeration, as we can see from the horizontal distance between the two curves on the graph. Probe significantly outperforms the baseline height-based enumeration technique.

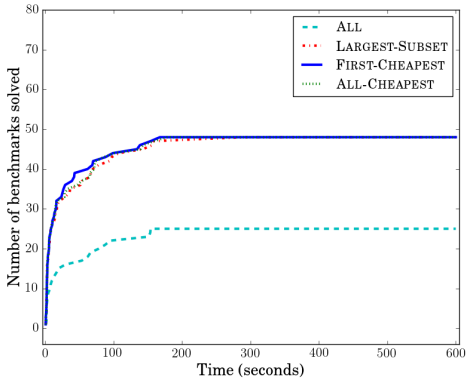
Results for Circuit Domain. The results for the Circuit benchmarks are shown in Fig. 13d. Each of the three techniques solves 22 out of 31 benchmarks, with size-based enumeration outperforming Probe in terms of synthesis times. The reason Probe performs worse in this domain is that the Circuit grammar 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 116, 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.**

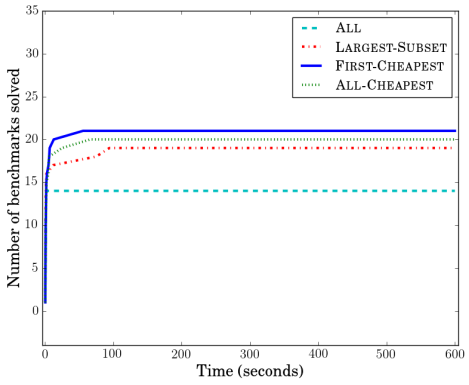
6.3 Q1.2: Selection of Partial Solutions

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.

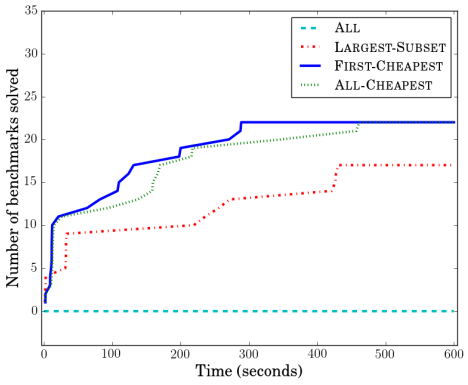
The all baseline scheme performs consistently worse than the other schemes on all three domains (and also worse than unguided size-based enumeration). For the circuit domain (Fig. 14c), the all scheme solves none of the benchmarks. The performance of the remaining schemes is very similar, indicating 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.



(a) STRING domain



(b) BITVEC domain



(c) CIRCUIT domain

Fig. 14. Number of benchmarks solved by PROBE with schemes for selecting promising partial solutions. Schemes are described in Sec. 5.2; ALL represents no selection (all partial solutions are used to update the PCFG). Timeout is 10 min, graph scale is linear.

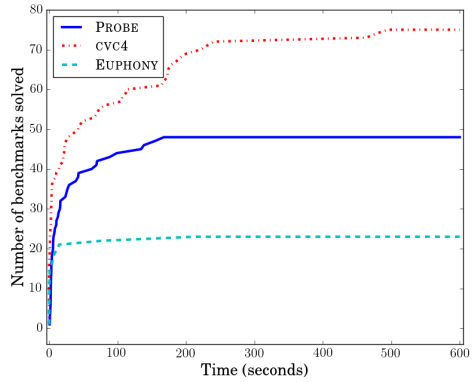
6.4 Q2: Is PROBE Faster than the State-of-the-art?

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.

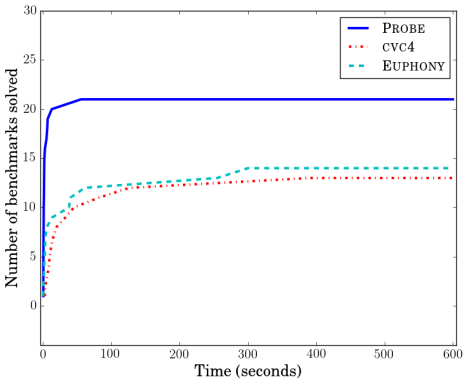
STRING Domain. Results for the STRING domain are shown in Fig. 15a. Of the 82 benchmarks in the STRING suite, PROBE solves 48 benchmarks, with an average time of 29s and a median time of 8.3s. EUPHONY solves 23 benchmarks, with average of 15.4s and a median of 0.7s. CVC4 solves 75 benchmarks, with an average of 61.8s 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.

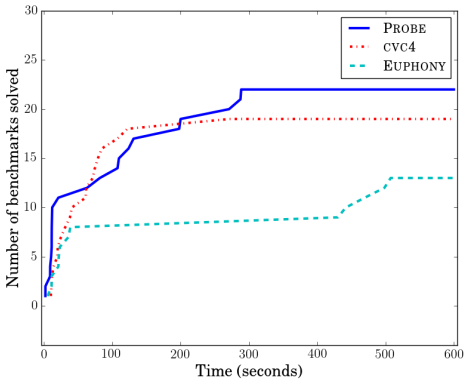
When only examining time to solution, CVC4 outperforms PROBE: not only does it solve more benchmarks faster, but it still solves new benchmarks long after PROBE and EUPHONY have plateaued. However, these solutions are not necessarily usable, as we show in Sec. 6.5.



(a) String domain



(b) BitVec domain



(c) Circuit domain

Fig. 15. Number of benchmarks solved by Probe, EuPhony and CVC4 for String, BitVec and Circuit domains. Timeout is 10 min, graph scale is linear.

BitVec Domain. Out of the 27 BitVec benchmarks, Probe solves 21 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 5s and median time of 1.5s. EuPhony's average time is 52s and median is 4.6s while CVC4 takes an average of 58s and a median of 15s. 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 22 benchmarks with an average time of 90s and median time of 42s (see Fig. 15c). EuPhony solves 13 benchmarks with average and

median times of 193.6s and 36s. CVC4 solves 19 benchmarks with average and median times of 60s and 41s. Probe outperforms both CVC4 and EuPhony in terms of the number of benchmarks solved. Moreover CVC4 generates much larger solutions than Probe, as discussed in Sec. 6.5.

Summary of Results. Of the total 140 benchmarks, Probe solves 91 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 the String domain; 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 solution 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 number of case splits 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 to size-based enumeration (which always returns the smallest solution by definition), as well as EuPhony and CVC4. Each comparison is limited to the benchmarks both tools can solve.

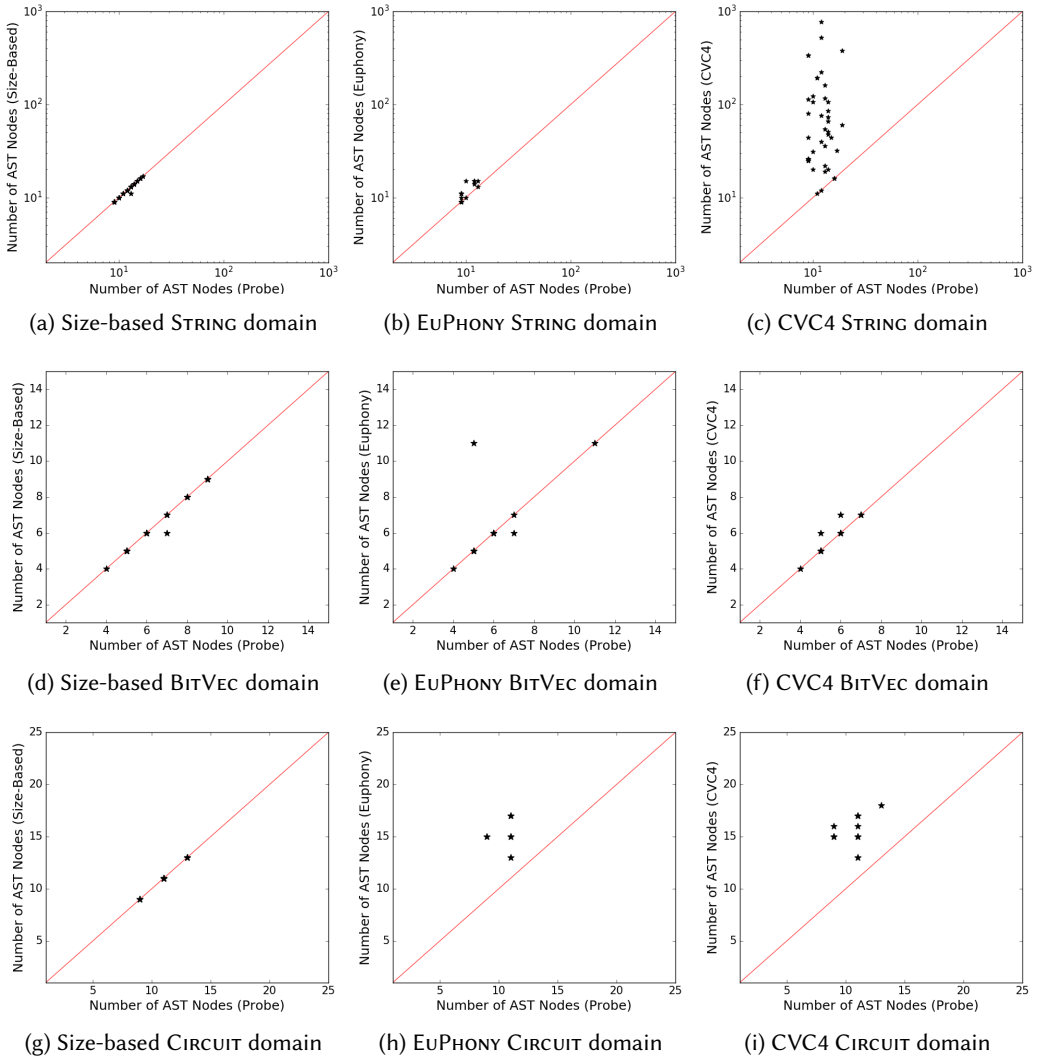
String Domain. First, we notice in Fig. 16a that Probe sometimes finds larger solutions than size-based enumeration, but the difference is small. Likewise, Fig. 16b shows that EuPhony and Probe return similar-sized solutions. Probe returns the smaller solutions for 10 benchmarks, but the difference is not large. On the other hand, CVC4 solutions (Fig. 16c) are larger than Probe's on 41 out of 45 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 finds the minimal solution in all cases except one. Solutions by EuPhony (Fig. 16e) and CVC4 (Fig. 16f) are slightly larger[10] in one (resp. two) cases, but the difference is small. For the Circuit benchmarks, Probe always finds minimal solutions, as shown in Fig. 16g. Both EuPhony (Fig. 16h) and CVC4 (Fig. 16i) generate larger solutions for *all* of the commonly solved benchmarks. Hence, on the Circuit domain, Probe outperforms its competitors with respect to *both* synthesis time and solution size.

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

[9] https://sygus.org/comp/2019/results-slides.pdf, slide 13

[10] Note that we use linear scale for BitVec and Circuit as opposed to logarithmic scale for String.



(a) Size-based String domain (b) EuPhony String domain (c) CVC4 String domain

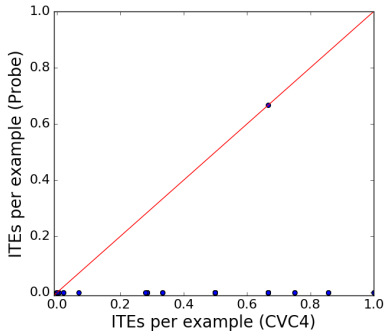
(d) Size-based BitVec domain (e) EuPhony BitVec domain (f) CVC4 BitVec domain

(g) Size-based Circuit domain (h) EuPhony Circuit domain (i) CVC4 Circuit domain

Fig. 16. Comparison between sizes of programs generated by different algorithms. Fig. 16a, Fig. 16b and Fig. 16c compare Probe *vs.* size-based enumeration, EuPhony and CVC4, respectively, on the String domain; graphs are log scale. Fig. 16d, Fig. 16e and Fig. 16f compare the same pairs of tools on the BitVec domain and Fig. 16g, Fig. 16h and Fig. 16i on the Circuit domain; graphs are linear scale.

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 averages 0.42 `ites` per example. When also considering benchmarks Probe cannot solve, some CVC4 programs 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



(a) Number of `ite` operations per examples

Test Benchmark	Training Examples	Testing Examples	Probe Accuracy	CVC4 Accuracy
initials-long	4	54	100%	100%
phone-5-long	7	100	100%	100%
phone-6-long	7	100	100%	100%
phone-7-long	7	100	100%	7%
phone-10-long	7	100	100%	57%
phone-9-long	7	100	N/A	7%
univ_4-long	8	20	N/A	73.6%
univ_5-long	8	20	N/A	68.4%
univ_6-long	8	20	N/A	100%
Avg Accuracy			100%	68.1%

(b) Generalization accuracy on unseen inputs

Fig. 17. Fig. 17a displays the number of `ite` operations per example for the 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.

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 a subset of the 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 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.

Benchmark	Solution generated	Time (s)
stackoverflow1.sl	`(substr arg 0 (+ (indexof arg "Inc" 1) -1))`	2.2s
stackoverflow3.sl	`(substr arg (- (to.int (concat "1" "9")) 2) (len arg))`	2.1s
stackoverflow8.sl	`(substr arg (- (len arg) (+ (+ 2 4) 4)) (len arg))`	6.5s
stackoverflow10.sl	`(substr arg (indexof (replace arg " " (to.str (len arg))) " " 1) 4)`	27.6s
exceljet1.sl	`(substr arg1 (+ (indexof arg1 "_" 1) 1) (len arg1))`	1.5s
exceljet2.sl	`(replace (substr arg (- (len arg) (indexof arg "." 1)) (len arg)) "." "")`	16.5s
initials.sl	`(concat (concat (at name 0) ".") (concat (at name (+ (indexof name " " 0) 1)) "."))`	134.5s
phone-6-long.sl	`(substr name (- (indexof name "-" 4) 3) 3)`	3.4s
43606446.sl	`(substr arg (- (len arg) (+ (+ 1 1) (+ 1 1))) (+ (+ 1 1) 1))`	10.8s
11604909.sl	`(substr (concat " " arg) (indexof arg "." 1) (+ (+ 1 1) 1))`	15.9s

Fig. 18. Probe solutions for 10 randomly selected benchmarks out of the 48 benchmarks Probe solves from the [eup 2018] String testing set, Time indicates the synthesis time in seconds.

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.

Benchmark	Solution generated	Time (s)
hd-11.sl	`(bvult y (bvand x (bvnot y)))`	2.4s
hd-09.sl	`(bvsub x (bvshl (bvand (bvashr x #x000000000000001f) x) #x0000000000000001))`	6.3s
hd-15.sl	`(bvsub (bvor x y) (bvlshr (bvxor x y) #x0000000000000001))`	13s
hd-18.sl	`(bvult (bvxor x (bvneg x)) (bvneg x))`	1.3s
hd-13.sl	`(bvor (bvashr x #x000000000000001f) (bvlshr (bvneg x) #x000000000000001f))`	1.6s

Fig. 19. Probe solutions for 5 randomly selected benchmarks out of the 21 benchmarks Probe solves from the Hacker's Delight BitVec set, Time indicates the synthesis time in seconds.

Benchmark	Solution generated	Time (s)
CrCy_10-sbox2-D5-sIn14.sl	`(xor LN200 (xor LN61 (and (xor LN16 LN17) LN4)))`	9.4s
CrCy_10-sbox2-D5-sIn88.sl	`(xor LN73 (and (xor (and LN70 (xor (xor LN236 LN252) LN253)) LN71) LN74))`	287.1s
CrCy_10-sbox2-D5-sIn78.sl	`(and (xor (and LN70 (xor (xor LN236 LN252) LN253)) LN73) LN77)`	11.8s
CrCy_10-sbox2-D5-sIn80.sl	`(xor LN73 (and LN70 (xor (xor LN236 LN252) LN253)))`	2.2s
CrCy_8-P12-D5-sIn1.sl	`(xor (xor (xor LN3 LN7) (xor (xor LN75 LN78) LN81)) k4)`	9.1s

Fig. 20. Probe solutions for 5 randomly selected benchmarks out of the 22 benchmarks Probe solves from the [eup 2018] Circuit set, Time indicates the synthesis time in seconds.

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 the String domain, Fig. 19 for the 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 number of case-splits, 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. 2013] 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. 2013; Peleg and Polikarpova 2020; Udupa et al. 2013]. These two strategies have complementary strengths and weaknesses, similar to backward chaining and forward chaining in proof search.

One important advantage of bottom-up enumeration for inductive synthesis 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 space based on other syntactic and semantic notions of program equivalence: for example, [Frankle et al. 2016; Gvero et al. 2013; Osera and Zdancewic 2015] only produce programs in a normal form; [Feser et al. 2015; Kneuss et al. 2013; Smith and Albarghouthi 2019] perform symmetry reduction based on equational theories (either built-in or user-provided); 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 general idea is to prioritize the exploration of grammar productions based on scores assigned by a probabilistic model; the specific technique, however, varies depending on (1) the context taken into consideration by the model when assigning scores, and (2) how the scores are taken into account during search. Like Probe, [Balog et al. 2016; Koukoutos et al. 2017; Menon et al. 2013] use a PCFG, which assigns scores to productions *independently of their context* within the synthesized program; unlike Probe, however, these techniques select the PCFG once, at the beginning of the synthesis process, based on a learned mapping from semantic specifications to scores. On the opposite end of the spectrum, Metal [Si et al. 2019] and Concord [Chen et al. 2020] use graph-based and sequence-based models, respectively, to condition the scores on the *entire partial program* that is being extended. In between these extremes, EuPhony uses a learned context in the form of a *probabilistic higher-order grammar* [Bielik et al. 2016], while NGDS [Kalyan et al. 2018] conditions the scores on the *local specification* propagated top-down by the deductive synthesizer. The more context a model takes into account, the more precise the guidance it provides, but also the harder it is to learn. Another consideration is that neural models, used in [Chen et al. 2020; Kalyan et al. 2018; Si et al. 2019] incur a larger overhead than simple grammar-based models, used in Probe and [Balog et al. 2016; Koukoutos et al. 2017; Lee et al. 2018; Menon et al. 2013], since they have to invoke a neural network at each branching point during search.

As for using the scores to guide search, most existing techniques are specific to *top-down enumeration*. They include prioritized depth-first search [Balog et al. 2016], branch and bound search [Kalyan et al. 2018], and variants of best-first search [Koukoutos et al. 2017; Lee et al. 2018; Menon et al. 2013]. In contrast to these approaches, Probe uses the scores to guide *bottom-up enumeration* with observational equivalence reduction. Probe's enumeration is essentially a bottom-up version of best-first search, and it empirically performs better than the top-down best-first search in EuPhony; one limitation, however, is that our algorithm is specific to PCFGs and extending it to models that require more context is not straightforward.

DeepCoder [Balog et al. 2016] also proposes a scheme they call *sort and add*, which is not specific to top-down enumeration and can be used in conjunction with any synthesis algorithm: this scheme runs synthesis with a reduced grammar, containing only productions with highest scores, and

iteratively adds less likely productions if no solution is found. Although very general, this scheme is less efficient than best-first search: it can waste resources searching with an insufficient grammar, and has to revisit the same programs again once the search is restarted with a larger grammar.

Finally, Metal and Concord, which are based on reinforcement learning (RL), do not perform traditional backtracking search at all. Instead, at each branching point, they simply choose a single production that has the highest score according to the current RL policy; a sequence of such decisions is called a *policy rollout*. If a rollout does not lead to a solution, the policy is updated according to a reward function explained below and a new rollout is performed from scratch.

Learning Probabilistic Models. Approaches to *learning* probabilistic models of programs can be classified into two categories: pre-training and learning on the fly. In the first category, [Menon et al. 2013], EuPhony, and NGDS are trained using a large corpus of human-designed synthesis problems and their gold standard solutions (the latter can be provided by a human or synthesized using size-based enumeration). Such datasets are costly to obtain: because these models are domain-specific, a new training corpus has to be designed for each domain. In contrast, DeepCoder learns from randomly sampled programs and inputs; it is, however, unclear how effective this technique is for domains beyond the highly restricted DSL in the paper. Unlike all these approaches, Probe requires no pre-training, and hence can be used on a new domain without any up-front cost; if a pre-trained PCFG for the domain is available, however, Probe can also be initialized with this model (although we have not explored this avenue in the present work).

DreamCoder, Metal, and Concord are related to the just-in-time approach of Probe in the sense that they update their probabilistic model on the fly. DreamCoder 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.

The RL-based tools Metal and Concord start with a pre-trained RL policy and then fine-tune it for the specific task during synthesis. Note that off-line training is vital for the performance of these tools, while Probe is effective even without a pre-trained model. The *reward mechanism* in Metal is similar to Probe: it rewards a policy based on the fraction of input-output examples solved by its rollout. Concord instead rewards its policies based on infeasibility information from a deductive reasoning engine: productions that expand to infeasible programs have lower probability in the next rollout. Although the Concord paper reports that its reward mechanism outperforms that of Metal, we conjecture that rewards based on partial solutions are simply not as good a fit for RL as they are for bottom-up enumeration: as we discuss in Sec. 5.2, it is crucial to learn from *shortest* partial solutions to avoid irrelevant syntactic features; policy rollouts do not guarantee that short solutions are generated first. Finally, Concord's reward mechanism requires expensive solver invocations to check infeasibility of partial programs, while Probe's reward mechanism incurs practically no overhead compared to unguided search.

Leveraging Partial Solutions to Guide Synthesis. LaSy [Perelman et al. 2014] and FrAngel [Shi et al. 2019] are component-based synthesis techniques that leverage information from partial solutions to generate new programs. LaSy 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.

Similar to Probe, FrAngel leverages partial solutions that satisfy any subset of the example specification. FrAngel generates new programs by randomly combining fragments from partial solutions. Probe is similar to FrAngel and LaSy 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, Bester [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 Bester: it leverages these accumulated partial solutions 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. Moreover, this optimization only works on partial programs that can be evaluated on some examples. Probe's bottom-up search generates complete programs that can always be evaluated on all examples.

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 the 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 EuPhony, 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 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

2018. Euphony Benchmark Suite. https://github.com/wslee/euphony/tree/master/benchmarks

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *International Conference on Computer Aided Verification*. Springer, 934–950.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

Ethem Alpaydin. 2014. *Introduction to Machine Learning* (3 ed.). MIT Press, Cambridge, MA.

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. 1–8. http://ieeexplore.ieee.org/document/6679385/

Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2016. Sygus-comp 2016: results and analysis. *arXiv preprint arXiv:1611.07627* (2016).

Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017a. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438* (2017).

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017b. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 319–336.

Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-based Program Synthesis. *Commun. ACM* 61, 12 (Nov. 2018), 84–93. https://doi.org/10.1145/3208071

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).

Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-Time Learning for Bottom-up Enumerative Synthesis. (2020). https://shraddhabarke.github.io/publication/probe-oopsla

Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: probabilistic model for code. In *International Conference on Machine Learning*. 2933–2942.

Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. 2020. Program Synthesis Using Deduction-Guided Reinforcement Learning. In *International Conference on Computer Aided Verification*. Springer, 587–610.

Kevin Ellis, Lucas Morales, Mathias Sablé Meyer, Armando Solar-Lezama, and Joshua B Tenenbaum. 2018. Search, compress, compile: Library learning in neurally-guided bayesian program learning. *Advances in neural information processing systems* (2018).

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017a. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 422–436.

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. 2017b. Component-based synthesis for complex APIs. *ACM SIGPLAN Notices* 52, 1 (2017), 599–612.

John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 229–239.

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed Synthesis: A Type-theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 802–815. https://doi.org/10.1145/2837614.2837629

Jianhang Gao, Qing Zhao, Wei Ren, Ananthram Swami, Ram Ramanathan, and Amotz Bar-Noy. 2012. Dynamic Shortest Path Algorithms for Hypergraphs. *CoRR* abs/1202.0082 (2012). arXiv:1202.0082 http://arxiv.org/abs/1202.0082

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

Sumit Gulwani. 2016. Programming by Examples (and its applications in Data Wrangling). In *Verification and Synthesis of Correct and Secure Systems*, Javier Esparza, Orna Grumberg, and Salomon Sickert (Eds.). IOS Press.

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. *ACM SIGPLAN Notices* 46, 6 (2011), 62–73.

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 27–38.

Jeevana Priya Inala and Rishabh Singh. 2018. WebRelate: integrating web data with spreadsheets using examples. *PACMPL* 2, POPL (2018), 2:1–2:28. https://dl.acm.org/doi/10.1145/3158090

Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 215–224.

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186* (2018).

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis Modulo Recursive Functions. *SIGPLAN Not.* 48, 10 (Oct. 2013), 407–426.

Manos Koukoutos, Etienne Kneuss, and Viktor Kuncak. 2016. An Update on Deductive Synthesis and Repair in the Leon Tool. In *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016*. 100–111.

Manos Koukoutos, Mukund Raghothaman, Etienne Kneuss, and Viktor Kuncak. 2017. On repair with probabilistic attribute grammars. *arXiv preprint arXiv:1707.04148* (2017).

Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples. In *Proceedings of the 35th Conference on Programming Language Design and Implementation*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 55. https://doi.org/10.1145/2594291.2594333

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices* 53, 4 (2018), 436–449.

Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A machine learning framework for programming by example. In *International Conference on Machine Learning*. 187–195.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 619–630.

Hila Peleg and Nadia Polikarpova. 2020. Perfect is the Enemy of Good: Best-Effort Program Synthesis. In *34th European Conference on Object-Oriented Programming, ECOOP*.

Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven synthesis. *ACM Sigplan Notices* 49, 6 (2014), 408–418.

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling up Superoptimization. *SIGARCH Comput. Archit. News* 44, 2 (March 2016), 297–310. https://doi.org/10.1145/2980024.2872387

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 419–428.

Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. 2019. cvc 4 sy: smart and fast term enumeration for syntax-guided synthesis. In *International Conference on Computer Aided Verification*. Springer, 74–83.

Rohin Shah, Sumith Kulal, and Rastislav Bodik. 2018. Scalable Synthesis with Symbolic Syntax Graphs. (2018).

Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: component-based synthesis with control structures. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29. https://dl.acm.org/doi/10.1145/3290386

Xujie Si, Yuan Yang, Hanjun Dai, Mayur Naik, and Le Song. 2019. Learning a Meta-Solver for Syntax-Guided Program Synthesis. https://openreview.net/forum?id=Syl8Sn0cK7

Calvin Smith and Aws Albarghouthi. 2019. Program Synthesis with Equivalence Reduction. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*. 24–47. https://doi.org/10.1007/978-3-030-11245-5_2

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review* 40, 5 (2006), 404–415.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. *ACM SIGPLAN Notices* 48, 6 (2013), 287–296.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017a. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 452–466.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017c. Program Synthesis Using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL, Article 63, 30 pages. https://doi.org/10.1145/3158151

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017b. Synthesis of Data Completion Scripts Using Finite Tree Automata. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 62, 26 pages. https://doi.org/10.1145/3133886

Henry S Warren. 2013. *Hacker's delight*. Pearson Education.