ProCon: Continuous Enumeration for Just-In-Time Bottom-Up Synthesis

KYLE THOMPSON, ILANA SHAPIRO, and ANIRUDH CANUMALLA, UC San Diego, USA

In order to scale synthesis algorithms to the vast size of the search space without coupling the algorithm with existing training datasets, recent work has combined probabilistic models with just-in-time learning. With just-in-time probabilistically-guided weighted bottom-up-enumerative search, researchers have created an off-the-shelf approach that learns from partial solutions during the synthesis process itself. However, current approaches are based on discrete size-based enumeration, which does not leverage the full power of the probabilistic model since it involves rounding the probabilities to discrete weights. To overcome this limitation, we present *continuous rule-based enumeration*, in which programs, are enumerated in order of continuous, nonrounded weights as determined by a given weighting function. We implement our approach in a tool called ProCon, which operates on the syntax-guided synthesis (SyGuS) format and evaluate it on a variety of SyGus benchmarks. We demonstrate that ProCon's more precise enumeration can solve SyGus benchmarks while enumerating fewer programs.

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

Additional Key Words and Phrases: Program Synthesis, Probabilistic Models

ACM Reference Format:
Kyle Thompson, Ilana Shapiro, and Anirudh Canumalla. 2024. ProCon: Continuous Enumeration for Just-In-Time Bottom-Up Synthesis. *J. ACM* 37, 4, Article 111 (March 2024), 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Syntax-guided synthesis, or SyGus, is a popular synthesis framework in which the programmer provides a syntactic template for the desired program in addition to the correctness (i.e. semantic) specification. A common instance of SyGuS is a search problem whose input consists of context-free grammar (CFG) that defines the space of possible programs, and the semantic specification comprises of a set of input-output examples. The input-output example paradigm is also called Programming-by-Example, or PBE. From the CFG and input-output examples, the goal of the synthesizer is to find a program generated by the grammar whose behavior is consistent with the semantic specification. The use of inductive semantic specifications such as input-output examples is also known as *inductive program synthesis* [Alur, Bodik, et al. 2013].

One of the primary challenges in synthesis is how to efficiently search for the desired program in the monumentally vast space of possible programs. As the synthesizer considers longer programs, synthesis quickly becomes intractable due to the size of the search space. To improve the scalability of synthesis models, several approaches optimizing synthesis search algorithms have been studied, including EUSolver [Alur, Radhakrishna, et al. 2017], Euphony [Lee et al. 2018] (which guide

Authors' address: Kyle Thompson, r7thompson@ucsd.edu; Ilana Shapiro, ilshapiro@ucsd.edu; Anirudh Canumalla, acanumalla@ucsd.edu, UC San Diego, 3235 Voigt Dr, La Jolla, CA, USA, 92093.

0004-5411/2024/3-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

top-down search), and TF-Coder [Shi et al. 2022], Bustle [Odena et al. 2021], Heap Search [Fijalkow et al. 2022], Probe [Barke et al. 2020], and Bee Search [Ameen and Lelis 2023] (which guide bottom-up search).

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. A key limitation of most guided bottom-up search methods is how it enumerates programs during the search. Probe, for instance, improves its performance by modeling its weight system after *size-based enumeration*, where the program bank is indexed by AST size, rather than height, as this has been observed empirically to be more efficient [Barke et al. 2020]. However, in weighted enumerative search, both height-based and sized-based approaches require discrete values to index the bank, meaning that transforming these approaches to rank programs by continuous, real-valued weights is difficult. Thus, Probe, as well as the similar models in Bustle and TF-Coder, rank programs by rounding real-valued probabilities to discrete costs, i.e. only enumerating programs *approximately* in the order of decreasing likelihood. This results in a loss of the full power of the probabilistic model used to guide the search.

To overcome this limitation, we propose an alternative approach we call *continuous bottom-up enumeration*. Rather than ranking programs by any discrete metric, we rank *nonterminals* from the grammar in order of their real-valued continuous weights, allowing us to leverage the full power of the probabilistic ranking. Specifically, we index the program bank by nonterminals rather than costs, and ensure that the bank is ordered by continuous weights. We implement our approach in a tool called ProCon that we build on top of Probe. Our approach meshes well with Probe's just-in-time-learning paradigm, in which the probabilistic model is learned on the fly. We evaluate ProCon on 77 string benchmarks and demonstrate that we are able to find the solution with fewer examples than Probe.

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

(1) *Continuous bottom-up enumeration*: a new approach to probabilistically-guided bottom-up enumerative search that enumerates programs in order of real-valued weights (Sec. 3).
(2) *ProCon*: a prototype implementation of our continuous ranking system with just-in-time-learning and its evaluation on 77 string benchmarks (Sec. 4).

2 RELATED WORK

EUSolver extends traditional top-down enumerative search approach with a divide-and-conquer algorithm, enumerating smaller expressions that are correct on subsets of inputs, as well as the predicates that distinguish these subsets. Then, it combines these expressions and predicates using a multi-label decision tree learning algorithm to form a conditional expression using Boolean combinations of the enumerated predicates [Alur, Radhakrishna, et al. 2017]. Euphony builds upon EUSolver by extending the grammar with a probabilistic model that determines the likelihood of each program. Euphony achieves superior performance by combining a probabilistic model with weighted top-down enumerative search (specifically, A*) to efficiently enumerate programs in the order of their likelihood. EUSolver learns a probabilistic higher order grammar (PHOG) from known solutions of synthesis problems solved by existing techniques in order to incorporate a richer context [Lee et al. 2018].

More recently, Probe has drawn from Euphony to (1) develop *guided bottom-up-search*, which uses a PCFG to rank enumerated programs by likelihood via bottom-up, rather than top-down, enumerative search and (2) use the partial, yet complete, programs encountered during bottom-up search to enable just-in-time learning of the model. In other words, Probe learns a PCFG "just-in-time," i.e. during synthesis, rather than ahead of time, which eliminates the dependency on large and often difficult to obtain training datasets that tools like Euphony and EUSolver require.

Programs are enumerated by discrete cost levels determined by the PCFG. The weighted top-down A* search in EUSolver is incompatible with just-in-time-learning, since top-down enumeration generates incomplete programs that cannot (yet) be evaluated [Barke et al. 2020].

TF-Coder, Bustle, and Heap Search take a similar approach: all use a cost function to guide bottom-up search. The function these systems employ favors programs that are more likely to lead to a solution. TF-Coder's cost function requires a manually crafted set of weights for each operation of the language. During the search, TF-Coder prioritizes combining programs with lower weights than programs with larger weights, thus biasing the search, where the weight of a program is defined as the sum of the weights of the production rules used to generate the program [Shi et al. 2022]. Bustle, on the other hand, uses a neural network as a cost function to compute the probability that a program is part of a solution. The network is a binary classification model that receives the input-output pairs of the task and the output of a program to each of the input values, and returns the probability that the program is a subprogram of a solution to the task [Odena et al. 2021]. Finally, unlike TF-Coder, Bustle, and Probe, Heap Search uses a data structure based on heaps to efficiently enumerate all programs in non-increasing order of the real-valued PCFG probabilities. However, it does so with respect to the *evaluation* of the programs, meaning that each program's AST must be created to determine its cost. This means that Heap Search can generate a very large number of programs that are more expensive than the solution program. Heap Search also sacrifices observational equivalence to achieve its best-first ordering.

To our knowledge, Bee Search represents the current state-of-the-art in guided bottom-up enumerative search. Bee Search performs search in a best-first ordering according to novel cost functions from both the pre-generation (i.e. before AST generation, like Probe) and post-generation (i.e. after AST generation, like Heap Sort) types of cost functions. Bee Search performs best-first search with respect to the is with respect to the *generation* of programs – i.e. Bee Search does not even create in memory programs that are more expensive than the solution program, thus circumventing the Heap Sort problem. Bee Search's generation-time best-first search is achieved by searching in an abstract cost-tuple space. One cost-tuple space is defined for each non-terminal rule, and each cost-tuple state represents a set of programs that are to be generated, such that the best-first ordering of programs is attained. Bee Search does not sacrifice observational equivalence checks in its approach. It performs no worse, and usually outperforms, Probe, TF-Coder, Bustle, and Heap Search on all its benchmarks, particularly for larger DSLs [Ameen and Lelis 2023].

3 IMPLEMENTATION

3.1 Problem Description

We formulate our problem identically to [Barke et al. 2020]. Our goal is to find programs that satisfy a given set of input-output examples, $\mathcal{E}$. Our programs must adhere to a specified grammar, $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{S}, \mathcal{R})$, where $\mathcal{N}$ is a set of nonterminal symbols, Σ is a set of terminal symbols, $\mathcal{S}$ is the starting nonterminal, and $\mathcal{R}$ is a set of production rules. Any given production rule $R \in \mathcal{R}$ is of the form $N \longrightarrow (t\ N_1\ N_2\ ...\ N_k)$. We use $N_{\text{rhs}}(R)$ to denote the tuple of nonterminals on the right hand side of the rule. We use $N_{\text{lhs}}(R)$ to denote the nonterminal on the left hand side of the rule.

3.2 Background on Probe architecture

Since ProCon builds on Probe's architecture, we first provide an overview of Probe's system that we go on to modify. At a high level, Probe takes in as input an inductive synthesis problem in SyGuS format; specifically, a context-free grammar of the desired DSL and a set of input-output examples. It then alternates between a synthesis phase and a learning phase until a solution is found or timeout is reached, as seen in Figure 1. This feedback loop enables Probe's just-in-time-learning.

In the synthesis phase, PROBE searches over the space of programs in order of increasing discrete costs. During the learning phase, PROBE updates the PCFG using the partial solutions found in the synthesis phase [Barke et al. 2020].

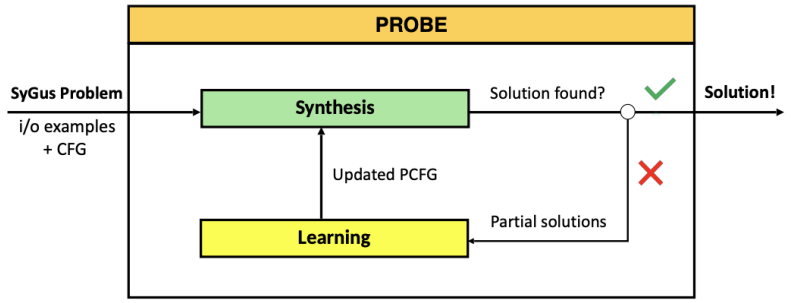

Fig. 1. Overview of PROBE [Barke et al. 2020]

For the synthesis phase, PROBE's algorithm takes a set of input-output examples and enumerates programs in the order of increasing discrete costs according to the PCFG until it finds a program that satisfies the entire specification or reaches a certain cost limit. The algorithm maintains a search state that consists of (1) the current discrete cost level; (2) the program bank storing all enumerated programs indexed by their cost; (3) the evaluation cache storing evaluation results of all programs in the bank in order to check for observational equivalence; and (4) the set of all enumerated partial solutions.

For the learning phase, PROBE uses a simple closed formula to compute new probabilities for each production rule based on the highest proportion of input-output examples that any partial solution derived using this rule satisfies. Since size-based bottom-up enumeration is not amenable to real-valued costs, PROBE enumerates programs *approximately* in the order of decreasing likelihood by converting CFG rule probabilities into discrete costs (the rounded negative logs), which determine the current cost level of that program. The program bank is then indexed by cost, and many programs will end up at the same cost level [Barke et al. 2020].

Every time the PCFG is updated during a learning phase (which happens after the cost level changes), PROBE restarts the bottom-up enumeration from scratch, i.e. it empties the program bank and evaluation cache, and resets the current cost level to zero. 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 [Barke et al. 2020].

PROCON only alters the synthesis phase of PROBE's algorithm. It eliminates the need to round the CFG rule probabilities for each program to a discrete cost level, and indeed eliminates the need for cost levels that comprise multiple programs entirely by ranking non-terminals, rather than programs, by their real-valued probabilities.

3.3 Enumeration by Real-Valued Likelihood

In this section, we describe how to perform bottom-up enumeration by real-valued weights. Our algorithm supports any function of the form $W : \mathcal{L}(\mathcal{G}) \longrightarrow \mathbb{R}^+$, where $\mathcal{L}(\mathcal{G})$ is the language defined by grammar $\mathcal{G}$.

Our algorithm uses $W(P)$ to assign a weight to the root node of the input program P. The total weight of a program P is the sum of the weights of all of the nodes in P, denoted $\mathbf{W}(P)$.

For example, consider the following tiny language:

```
S := e
e := arg | (substr e n n)
n := 1 | 0 | (+ n n)
```

Suppose this tiny language had a weight function that partially consisted of

$W(\texttt{0}) = 0.1$
$W(\texttt{1}) = 0.8$
$W(\texttt{arg}) = 0.5$
$W(\texttt{(substr arg 0 1)}) = 0.3$

In this case, $\mathbf{W}(\texttt{(substr arg 0 1)}) = 1.7$

Our goal is to enumerate programs in order of weights as determined by $\mathbf{W}$. For this language, that would mean enumerating programs in the following order: `0`, `arg`, `1`, `(substr arg 0 1)`. Like previous approaches, we maintain a bank of programs that we have already enumerated. We build new programs by combining programs in the bank. Unlike previous approaches that index the bank by integral program costs, we index the bank only by nonterminal. For each nonterminal in the bank, the list of previously enumerated programs is sorted by real-valued weights. This invariant is cheap to maintain as our algorithm enumerates programs by weight, so we can simply append to the end of the bank at the proper nonterminal when we enumerate a new program.

3.3.1 Enumerating Programs for a Particular Rule. We will first describe the algorithm to enumerate programs for a particular production rule $R \in \mathcal{R}$. We enumerate programs from rule R by enumerating subterms to use in the non-terminals of rule R. We enumerate these tuples of subterms in order of the total weight of the tuple. Since the bank of programs is ordered by $\mathbf{W}$, it is easy to enumerate subterms in order of weight by doing a Dijkstra search over indices into the bank. For sake of example, suppose we want to enumerate subterms to the rule `(substr e n n)`. We can achieve this enumeration by enumerating indices into the sequences bank[e], bank[n], and bank[n]. It is obvious that the tuple of subterms with the lowest total weight is at indices (0, 0, 0). From here, it is clear that the tuple of subterms with the next lowest weight is at indices (1, 0, 0), (0, 1, 0), or (0, 0, 1). Thus our Djikstra search forms a tree where nodes are tuples of indices, and each child of a node contains the same tuple of indices with one index incremented. Note that this naive implementation of determining the children of a node could lead to duplicates. This is visualized more clearly in Figure 2. One could take care of duplicates by either maintaining an

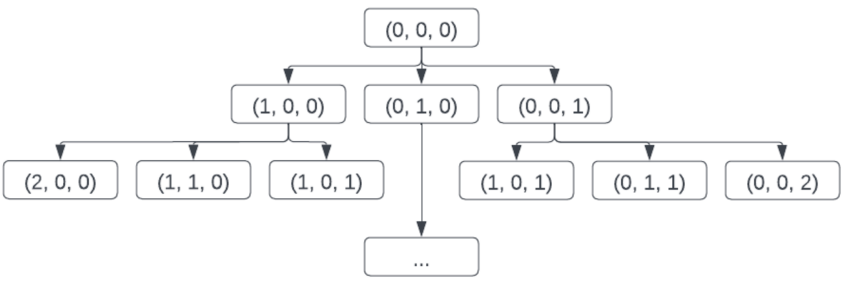

Fig. 2. Naive Enumeration with Duplicates

explored set, or by maintaining a bitset of permissions at each node to determine which indices

can be incremented. We opt to maintain a bit set at each node since we hypothesize that it is more efficient. This is visualized in Figure 3, where the red indices are “fixed" and do not have permission to get incremented.

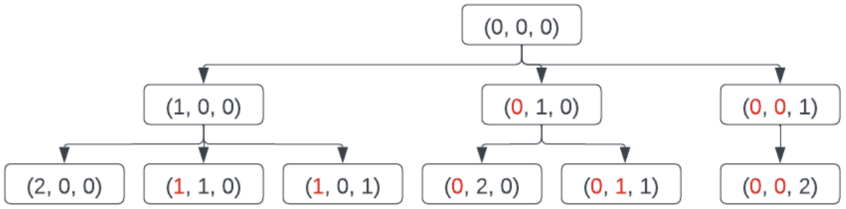

Fig. 3. Enumeration Without Duplicates

Note that this Djikstra search must happen alongside the growing of the program bank. If we just threw tuples of indices into the frontier, they might be out of bounds of the current program bank. Therefore, we maintain a set of indices, called “abstract candidates” that do not point to anything yet, but will point to something once the bank grows larger. Once an abstract candidate points to concrete subprograms, it can be added to the frontier. The full algorithm for enumerating subprograms to a rule R is given in Appendix A.

3.3.2 Enumerating All Programs. At any particular iteration, to make sure we find the program with the next lowest weight, we must consider the next candidates from each production rule. Specifically, we can keep track of the lowest-weight program so far and enumerate new programs from a production rule while the weight of the **children** of the new programs is smaller than the best candidate. Note that we do the comparison with the children because the enumerator for the production rule is in order of child weight, not program weight. Furthermore, we know that the program weight is greater than or equal to the child weight since $W(\cdot)$ is positive. Once we find the next best program, we add it to the program bank. The full details for enumerating programs in order of likelihood is in Appendix B.

3.3.3 Full ProCon Solver. We combine our enumeration by weight with Probe’s on-the-fly weight updates win Appendix C. Note that these on the fly weight updates assume that W is in the form of a PCFG. We use a program to update the PCFG if the program satisfies a unique subset of input output examples. We found that it is very important to do these updates in *batches*. One could imagine updating the PCFG after a single program satisfies a unique set of solutions. However, this gives too much power to individual programs and can lead to a noisy search. This is roughly analogous changing the cost level in Probe after just one program is added to Probe’s cost-indexed version of the bank, which would lead to Probe’s PCFG update. We thus set a threshold (`updateWeightLevel` in the psuedocode) leading to a variable sized “batch" of programs to accept based on their weights, before updating the PCFG with this entire batch. Importantly, the threshold is real-valued, since the weights are real-valued.

4 EVALUATION

We evaluated ProCon against 77 string benchmarks with a timeout of 60 seconds. We notice that ProCon solves more benchmarks per program enumerated (Figure 4), but it enumerates programs slower than Probe (Figure 5). We hypothesize that ProCon is slower at enumerating programs due primarily to the enqueueing and dequeing in Appendix A. These operations can become expensive since the queue of candidates can grow to the tens of thousands.

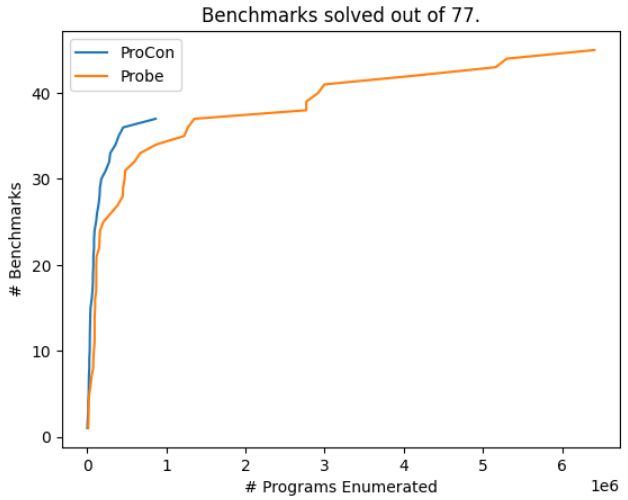

Fig. 4. Evaluation against Probe by number of programs enumerated

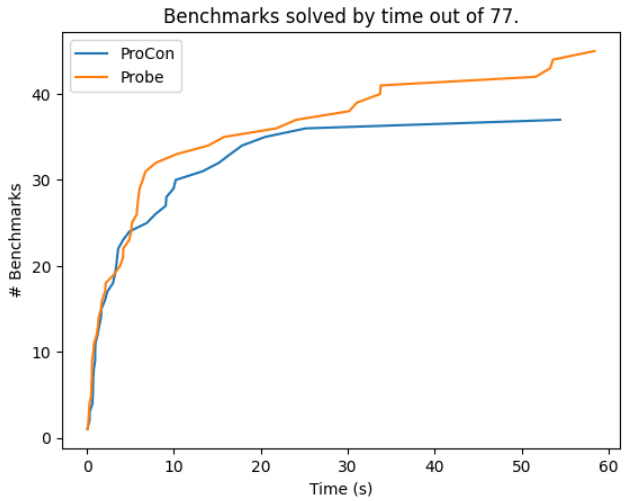

Fig. 5. Evaluation against Probe by wall clock time

5 CONCLUSION AND FUTURE WORK

We have presented an improvement to bottom-up enumerative synthesis with just-in-time learning that leverages the full power of the associated probabilistic model by enabling continuous, real-valued enumeration. We have implemented our algorithm in a tool called ProCon that works with the popular SyGuS input format and uses input-output examples as semantic constraints. We evaluated ProCon on 77 string benchmarks, and found that it was able to arrive at the solution with fewer enumerated programs than Probe, although there appears to be a computational bottleneck somewhere in our implementation since Probe is still faster.

In future work, we would also like to improve the learning phase of ProCon's algorithm by exploring other probabilistic models. Euphony demonstrated superiority of PHOGs over PCFGs in their approach [Lee et al. 2018], and analogously it seems that ProCon could also be improved by replacing its simple PCFG with a context-aware model. Note that such a model would get context from *subterms* instead of *incomplete programs*. Of course learning a more sophisticated probabilistic model would likely require training data which is currently not required ProCon. One interesting direction could be incorporating a trained probablistic model with Probe's learning on the fly since Probe's weights incorporate information from the specification.

We would also like to concretely understand why more-precise enumeration might be better. The learning phase of this algorithm makes it difficult to debug exactly why ProCon comes to a solution before Probe in a particular example. We would like to run an experiment where we fix the weights of the search to be the weights that lead to the solution. Then, we can run a more direct comparison between continuous and discrete enumeration without worrying about changing weights. Finally, it is important include Bee Search as a baseline in our future evaluations, since they currently represent the state of the art in guided bottom up synthesis.

REFERENCES

Rajeev Alur, Rastislav Bodik, et al.. Oct. 2013. "Syntax-guided synthesis." *2013 Formal Methods in Computer-Aided Design*, (Oct. 2013). doi: 10.1109/fmcad.2013.6679385.

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. "Scaling Enumerative Program Synthesis via Divide and Conquer." *Tools and Algorithms for the Construction and Analysis of Systems*, 319–336. doi: 10.1007/978-3-662-54577-5_18.

Saqib Ameen and Levi H. S. Lelis. 2023. "Program Synthesis with Best-First Bottom-Up Search." *J. Artif. Intell. Res.*, 77, 1275–1310. doi: 10.1613/JAIR.1.14394.

Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Nov. 2020. "Just-in-Time Learning for Bottom-Up Enumerative Synthesis." *Proceedings of the ACM on Programming Languages*, 4, OOPSLA, (Nov. 2020), 1–29. doi: 10.1145/3428295.

Nathanaël Fijalkow, Guillaume Lagarde, Théo Matricon, Kevin Ellis, Pierre Ohlmann, and Akarsh Nayan Potta. 2022. "Scaling Neural Program Synthesis with Distribution-Based Search." In: *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*. AAAI Press, 6623–6630. doi: 10.1609/AAAI.V36I6.20616.

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. June 2018. "Accelerating Search-Based Program Synthesis using Learned Probabilistic Models." *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, (June 2018). doi: 10.1145/3192366.3192410.

Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. 2021. "BUSTLE: Bottom-Up Program Synthesis Through Learning-Guided Exploration." In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=yHeg4PbFHh.

Kensen Shi, David Bieber, and Rishabh Singh. 2022. "TF-Coder: Program Synthesis for Tensor Manipulations." *ACM Trans. Program. Lang. Syst.*, 44, 2, 10:1–10:36. doi: 10.1145/3517034.

A RULE ITERATOR PSUEDOCODE

Algorithm 1 Rule Iterator

Input: A rule from the grammar: $R \in \mathcal{G}$
Output: Iterator over programs from R in order of the sum of the weights of the subprograms composed by R.

procedure SETUP
    $\text{argQueues}_{N \in N_{\text{rhs}}(R)} \leftarrow \text{bank}(N)$
    $\text{abstractCandidates} \leftarrow \{((0, 0, ..., 0), (1, 1, ..., 1))\}$
    $\text{concreteCandidates} \leftarrow []$

procedure UPDATEABSTRACTCANDIDATES
    realizableCandidates $\leftarrow$ {$c \in$ abstractCandidates | The indices are in bounds in the bank.}
    abstractCandidates $\leftarrow$ abstractCandidates \ realizableCandidates
    for $c \in$ realizableCandidates **do**
        $p \leftarrow$ REALIZE(c)
        concreteCandidates.enqueue(p)

procedure NEWABSTRACTCANDIDATES($c \in C$)
    newAbstractCandidates $\leftarrow \emptyset$
    (_, _, indices, permissions) $\leftarrow c$
    for $\{i \in [0..|\text{indices}|) \mid \text{permissions}(i)\}$ **do**
        $\text{newIndices}(j) \leftarrow \text{if } (j = i) \text{ then } (\text{indices}(j) + 1) \text{ else indices}(j)$
        $\text{newPermissions}(j) \leftarrow i \leq j$
        newAbstractCandidates = newAbstractCandidates $\cup$ {(newIndices, newPermissions)}
    return newAbstractCandidates

procedure PEEK
    UPDATEABSTRACTCANDIDATES()
    return concreteCandidates.peek

procedure NEXTPROGRAM
    UPDATEABSTRACTCANDIDATES()
    nextCandidate $\leftarrow$ concreteCandidates.dequeue()
    abstractCandidates $\leftarrow$ abstractCandidates $\cup$ NEWABSTRACTCANDIDATES(nextCandidate)
return nextCandidate

B PROGRAM ITERATOR PSUEDOCODE

Algorithm 2 Program Iterator

Input: A Grammar $\mathcal{G}$
Output: Iterator over Programs from $\mathcal{G}$ in order of likelihood.

```
1:  procedure SETUP
2:      candidateQueue ← []
3:      ruleIterators(R) ← Rule Iterator for R
4:      bank_{N∈G}(N) = []
5:
6:  procedure NEXTPROGRAM
7:      best ← if (|candidateQueue| = 0) then null else candidateQueue.peek
8:      for R ∈ ℛ do
9:          r ← ruleIterators(R)
10:         while hasNext(r) ∧ (best = null ∨ r.peek.childWeight  < best.weight) do
11:             p ← r.nextProgram()
12:             if p is observationally unique then
13:                 candidateQueue.enqueue(p)
14:                 best ← candidateQueue.head
15:     result ← candidateQueue.dequeue())
16:     bank(result.nonterminal).append(result)
17:     return result
```

C PROCON SOLVER PSUEDOCODE

Algorithm 3 ProCon Solver

Input: A Grammar $\mathcal{G}$ a set of I/O examples $\mathcal{E}$, and an update granularity u.
Output: A program from G that satisfies all examples in $\mathcal{E}$.

```
1:  procedure Solve
2:      uniquePartialSolutions ← ∅
3:      while True do
4:          enumerator ← program iterator for G
5:          interestingPrograms ← ∅
6:          updateWeightLevel ← u
7:          repeat
8:              nextProg ← enumerator.nextProgram()
9:              coverage ← examples in E staisfied by nextProg
10:             if coverage = E then
11:                 return nextProg
12:             if coverage ∉ uniquePartialSolutions then
13:                 uniquePartialSolutions ← uniquePartialSolutions ∪ coverage
14:                 interestingPrograms ← interestingPrograms ∪ nextProg
15:             if u ≤ nextProg.weight ∧ interestingPrograms = ∅ then
16:                 updateWeightLevel ← updateWeightLevel + u
17:         until u ≤ nextProg.weight ∧ interestingPrograms ≠ ∅
18:         Call UpdateProbabilities(interestingPrograms)
```

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009