Toward Probabilistic Coarse-to-Fine Program Synthesis

Maddy L. Bowers* MIT mlbowers@mit.edu

Alexander K. Lew* MIT alexlew@mit.edu

Vikash K. Mansinghka MIT vkm@mit.edu

Joshua B. Tenenbaum MIT jbt@mit.edu

Armando Solar-Lezama MIT asolar@mit.edu

1 Introduction

Consider the task of example-based program synthesis: given *input-output examples* (x_i, y_i), find a program p such that $[\![p]\!](x_i) = y_i$ for every example. Due to the vastness of the search space (all well-typed programs), we would like to break the synthesis problem into smaller, easier pieces.

This abstract proposes to *begin* by synthesizing *probabilistic* programs, which capture some of the deterministic dependence of y on x, but use randomness as a temporary substitute for as-yet-unknown input-dependent logic. For example, the probabilistic program

$$\lambda l.\text{filter}(\lambda x.\textbf{if } \text{even}(x) \textbf{ then } \text{flip}(0.5) \textbf{ else false})\ l$$

represents the partial or vague hypothesis that the program we are searching for "filters out the odd numbers from the input list... and some of the even ones too." Unlike programs with holes, e.g. $\lambda l.\text{filter}(\lambda x.\textbf{if } \text{even}(x) \textbf{ then } \square \textbf{ else false})\ l$, probabilistic programs have well-defined semantics, so we can evaluate the *likelihood* of each y_i given x_i. This likelihood provides signal as to whether a partial program is on the right track. Promising programs can then be iteratively *refined* by replacing high-entropy expressions with low-entropy ones.

This refinement process can either be used to synthesize *deterministic* programs, by iteratively refining away *all* of the stochastic primitives (Fig. 1); or alternatively, synthesis can be stopped early to yield a probabilistic program. Thus, we also report preliminary results (Fig. 3) of this approach applied to the setting of *Bayesian probabilistic program synthesis* [10], in which the final output is a probabilistic model.

Toward realizing this approach, we contribute:

- An argument that this probabilistic, coarse-to-fine approach to synthesis could unify many techniques from the (deterministic) program synthesis literature (§2).
- A sketch for an algorithm, inspired by bidirectional type inference, to address a key technical challenge in coarse-to-fine synthesis: the evaluation of likelihoods (§3).

LAFI 2024, January 2024, London, UK
2024.

Examples [3 2 7 6 8 4 7] → [4 1 8 5 7 3 8]
[1 8 6 5 7 4] → [2 7 5 6 8 3]

Coarse

$\lambda l.\,\text{map}\ (\lambda x.\,\text{randomDigit})\ l$
$\mathbf{0.1^{13} = 1.0 \times 10^{-13}}$

$\lambda l.\,\text{map}\ (\lambda x.\,\textbf{if } \text{flip}(0.5) \textbf{ then } \text{randomDigit} \textbf{ else } \text{inc}(x))\ l$
$\mathbf{0.55^{6} \cdot 0.05^{7} = 2.2 \times 10^{-11}}$

$\lambda l.\,\text{map}\ (\lambda x.\,\textbf{if } \text{flip}(0.5) \textbf{ then } \text{dec}(x) \textbf{ else } \text{inc}(x))\ l$
$\mathbf{0.5^{13} = 1.2 \times 10^{-4}}$

$\lambda l.\,\text{map}\ (\lambda x.\,\textbf{if } \text{even}(x) \textbf{ then } \text{dec}(x) \textbf{ else } \text{inc}(x))\ l$
$\mathbf{1^{13} = 1.0}$

Fine

Figure 1. We propose to find *coarse* or *vague* probabilistic programs that only partly explain the examples, then iteratively *refine* them by rewriting high-entropy expressions, using likelihoods (shown here in green) to guide the search.

- Preliminary empirical evidence that likelihoods of input-output examples under partial programs can provide useful guidance for top-down synthesis, in two domains: synthesizing recursive list processing functions, and learning symbolic classifiers (§4).

2 Patterns of Coarse-to-Fine Reasoning

The signal provided by likelihoods of partial programs can be strong, and may obviate the need for some of the specialized heuristics seen in algorithms for synthesis. For instance:

Condition abduction. Consider the example in Fig. 1, which introduces a probabilistic **if** expression, enabling synthesis to focus in each branch on explaining only some of the input-output examples. The branch condition can later be refined with an input-dependent predicate (in this case, even(x)), to further boost the likelihood. This pattern is known as *condition abduction* in the synthesis literature [1, 2, 6, 9].

Decision-tree learning. Suppose we are entertaining multiple hypotheses of the form **if** *cond* **then** flip(θ_1) **else** flip(θ_2). Hypotheses with more informative predicates *cond*—i.e., predicates that account for more of the variation in outputs—can have lower-entropy branches, with θ_1 and θ_2 closer to 0 or 1. This in turn will lead to higher likelihoods for such hypotheses. This is similar to the logic of top-down decision-tree

$$\begin{aligned} \text{Types } \tau \ &::= \mathbb{B} \mid \mathbb{N} \mid \mathbb{R} \mid \textbf{List } \tau \mid \tau_1 \to \tau_2 \\ \text{Terms } t \ &::= c \mid x \mid \lambda x.t \mid \mu f.\lambda x.t \mid \textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2 \mid t_1 = t_2 \mid \\ &\qquad \textbf{cons } t_1\ t_2 \mid \textbf{nil} \mid \textbf{car } t \mid \textbf{cdr } t \mid \textbf{flip } t \mid \textbf{randomDigit} \end{aligned}$$

$$\frac{t_1 \Downarrow p_1 \leftarrow v_1 \qquad t_2 \Downarrow p_2 \leftarrow v_2}{\textbf{cons } t_1\ t_2 \Downarrow p_1 p_2 \leftarrow \textbf{cons}(v_1, v_2)} \qquad \frac{}{\lambda x.t \to [(\lambda x.t, 1.0)]}$$

$$\frac{t \Downarrow p \leftarrow \textbf{true} \qquad t_1 \Downarrow p_1 \leftarrow v \qquad t_2 \Downarrow p_2 \leftarrow v}{\textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2 \Downarrow pp_1 + (1-p)p_2 \leftarrow v}$$

Figure 2. Example rules for bidirectional likelihood evaluation, for a DSL of recursive list transformations. The symbol c ranges over numeric *constants*. The term $\mu f.\lambda x.t$ builds recursive functions; the name f is bound in the body t for recursive calls. The notation $t \Downarrow p \leftarrow v$ indicates that evaluating t yields likelihood p when constrained to output v, the notation $t \to [(v_i, p_i)]_i$ indicates that unconstrained evaluation of t enumerates values v_i each with probability p_i.

learning, where predicates are selected based on the *information gain* or average reduction in entropy they enable.

Pruning. By pruning *zero*-likelihood partial programs, we recover many *logical* methods for pruning in synthesis. For example, filter $(\lambda x.\text{flip}(0.5))\ l$ assigns positive likelihood only to subsets of l, and map $(\lambda x.\text{randomDigit}())\ l$ only to lists the same length as l. Similar rules were hand-coded into systems like λ^2 [3], but for us, fall out automatically.

3 Technical Challenges

To implement this approach, we require:

- A search algorithm, to find initial vague programs and iteratively refine them. Although the search space we consider is largely discrete, the search algorithm should also have logic for *tuning* continuous parameters (e.g. θ in flip(θ)) to fit the input-output examples.
- A way to evaluate the *marginal likelihoods* that probabilistic programs assign to outputs, to guide the search.

Some probabilistic languages, such as Dice [5] and SPPL [10], automate the exact computation of marginal likelihoods, making them great potential targets for synthesis. However, they cannot express many programs of interest, e.g. the recursive list-processing programs we consider here. Other languages can use approximate inference to *estimate* marginal likelihoods unbiasedly, but generic algorithms would likely take too long to produce *low-variance* estimates to be effective within the inner loop of synthesis.

Our prototype uses a scheme, inspired by bidirectional type inference [8], that we believe could generalize to handle marginal density evaluation or estimation in a broader class of programs. The idea is to evaluate expressions in two *modes* (as in the symbolic disintegration algorithm of Shan and Ramsey [11]): *constrained* by a target output value (with the goal of computing its likelihood), and *unconstrained* (with the goal of enumerating possible values the expression may take

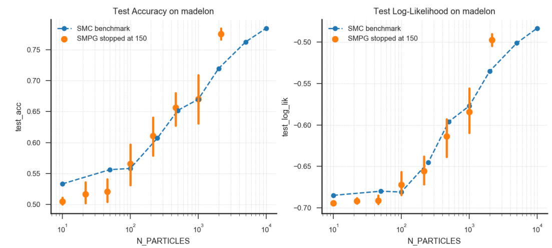

Figure 3. In a simple first-order language with **if**, predicates, and categoricals, coarse-to-fine synthesis should resemble decision-tree learning (§2). Our sequential Monte Carlo implementation in Gen.jl (orange) performs similarly to a sequential Bayesian decision-tree-learning algorithm [7].

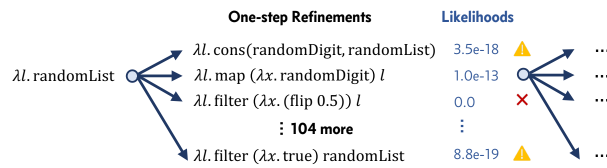

Figure 4. At each step of coarse-to-fine synthesis a probabilistic primitive in a partial program (left) is refined according to production rules in a grammar. The likelihoods of the refined programs are evaluated with respect to the input-output examples. Branches with likelihood zero are pruned, while high likelihood branches are prioritized.

on). The two modes are mutually recursive, as in bidirectional type inference; constraining the output of an application $(e_1\ e_2)$, for example, enumerates possible values $(\lambda x.t)$ of e_1 and for each, constrains $t[e_2/x]$ to produce the desired output. Fig. 2 presents example rules from both modes.

4 Preliminary Results

Recursive list processing programs. We implement a prototype version of the bidirectional likelihood evaluator and coarse-to-fine synthesizer in Julia for a recursive list domain. In Table 1 we analyze the sequence of refinement steps made along the path to synthesizing the solution program in Figure 1 (see also Figure 4 for a depiction of the first refinement step). We find that frequently a very large fraction of refinements can be pruned away. We also find that when ranking candidate refinements by the evaluated likelihood, the desired candidate is ranked highly (and frequently first).

Symbolic classifiers. We also consider a simpler language, with **if** expressions, a small grammar of deterministic predicates, and categorical distributions over possible output labels. The task is to synthesize a program in this language to assign labels to 500-dimensional input vectors from the MADELON dataset [4], based on a dataset of roughly 2000 examples—in this setting, we essentially recover decision tree learning. We implemented a coarse-to-fine sequential Monte

Step	Total	Rank	Pruned
#1	108	1	102
#2	1898	34	1654
#3	5548	1	0
#4	3650	1	1873
#5	1752	1	321

Table 1. Analysis of the refinements made when synthesizing the solution from Fig. 1. *Total* is number of refinements considered, *Pruned* is the number that are zero-likelihood, and *Rank* is how high the refinement that ultimately led to the solution was ranked when ordering by likelihood.

Carlo algorithm in Gen.jl and found that it performed comparably with previous approaches to Bayesian decision-tree learning [7].

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

[2] Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9207)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 163–179. https://doi.org/10.1007/978-3-319-21668-3_10

[3] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 229–239. https://doi.org/10.1145/2737924.2737977

[4] Isabelle Guyon. 2008. Madelon. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5602H.

[5] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling exact inference for discrete probabilistic programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–31.

[6] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 407–426. https://doi.org/10.1145/2509136.2509555

[7] Balaji Lakshminarayanan, Daniel Roy, and Yee Whye Teh. 2013. Top-down particle filtering for Bayesian decision trees. In *International Conference on Machine Learning*. 280–288.

[8] Benjamin C. Pierce and David N. Turner. 1998. Local Type Inference. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, David B. MacQueen and Luca Cardelli (Eds.). ACM, 252–265. https://doi.org/10.1145/268946.268967

[9] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538.

[10] Feras A Saad, Marco F Cusumano-Towner, Ulrich Schaechtle, Martin C Rinard, and Vikash K Mansinghka. 2019. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32.

[11] Chung-chieh Shan and Norman Ramsey. 2017. Exact Bayesian inference by symbolic disintegration. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 130–144.