Catherine Wong 1  Kevin Ellis 2  Joshua B. Tenenbaum 1 3  Jacob Andreas 1

Abstract

Inductive program synthesis, or inferring programs from examples of desired behavior, offers a general paradigm for building interpretable, robust, and generalizable machine learning systems. Effective program synthesis depends on two key ingredients: a strong library of functions from which to build programs, and an efficient search strategy for finding programs that solve a given task. We introduce LAPS (Language for Abstraction and Program Search), a technique for using natural language annotations to guide joint learning of libraries and neurally-guided search models for synthesis. When integrated into a state-of-the-art library learning system (DreamCoder), LAPS produces higher-quality libraries and improves search efficiency and generalization on three domains—string editing, image composition, and abstract reasoning about scenes—even when no natural language hints are available at test time.

1. Introduction

Machine learning approaches based on program synthesis—the automatic inference of symbolic programs—can offer robustness, interpretability, verifiability, and strong generalization in few-shot learning settings (Appel et al., 2017; Lake et al., 2017). Many machine learning tasks can be formulated as program synthesis problems, including data manipulation (Delaware et al., 2015; Gulwani et al., 2017), semantic parsing (Artzi & Zettlemoyer, 2013; Liang, 2016), structured visual understanding (Johnson et al., 2017b; Yi et al., 2018), image generation (Ellis et al., 2017; Ganin et al., 2018), and policy learning (Fikes & Nilsson, 1971; Cropper & Muggleton, 2015; Silver et al., 2020).

This paper introduces Language for Abstraction and Program Search (LAPS), a framework for improving the efficiency and generalizability of learned program synthesis models using natural language supervision. In LAPS, language guides learning of both libraries of reusable program abstractions and heuristics for searching in the space of programs. High-quality program libraries and search methods are the main ingredients of effective program synthesis approaches (Gulwani et al., 2017). Recent approaches to program synthesis have attempted to learn search models (Gulwani et al., 2015; Polozov & Gulwani, 2015; Balog et al., 2016; Devlin et al., 2017), program libraries, or both jointly from data (Shin et al., 2019; Dumancic & Cropper; Ellis et al., 2021; 2020; Lázaro-Gredilla et al., 2019), but even the current best learning approaches can be computationally inefficient (often requiring upwards of thousands of CPU hours to bootstrap learning) and do not always discover generalizable libraries or search strategies.

LAPS builds on the intuition that natural language offers a powerful source of information for tackling both learning problems. Language simultaneously provides an efficient channel for communicating the structure of the search space (an instruction like *draw a large hexagon next to a small pentagon* decomposes a complex graphics task into high-level parts) and a lexicon that names important reusable concepts in a given domain (for instance, suggesting that a function to draw variable-sized polygons might be useful for future graphics tasks). In this work we show how inducing jointly compositional generative models over natural language and programs provides a strong scaffold for library and search model learning in a hierarchical program induction model. When integrated into a state-of-the-art learning algorithm, DreamCoder (Ellis et al., 2021; 2018), our approach dramatically improves performance on three different synthesis domains: string editing, structured image generation and scene understanding. Compared to the base synthesis approach, LAPS solves and learns more quickly from synthesis tasks, and produces higher-quality libraries that improve generalization to downstream tasks without natural language hints.

LAPS builds on several recent developments in (non-language-based) program synthesis, so we begin with a review of related work (Sec. 2), then formalize the search and library learning problems (Sec. 3) and base synthesis algorithm (Sec. 4). We then describe how LAPS extends the base algorithm to include language in learning (Sec. 5) and conclude with empirical results (Sec. 6).
2. Related Work

Our work draws on recent program synthesis approaches that learn to synthesize programs from examples using neural models to guide search (Gulwani et al., 2015; Balog et al., 2016; Parisotto et al., 2016; Devlin et al., 2017; Polosukhin & Skidanov, 2018; Abolafia et al., 2018; Nye et al., 2019; Ellis et al., 2019; Si et al., 2019; Ye et al., 2020a); and learn libraries of symbolic abstractions from a collection of related programs or tasks (Dechter et al., 2013; Zhang et al., 2017; Shin et al., 2019; Dumancić & Cropper; Ellis et al., 2018; 2021). Our formulation builds on hierarchical Bayesian formulations of program learning that frame both synthesis and library learning as probabilistic inference (Liang et al., 2010; Lake et al., 2015; Ellis et al., 2021).

Natural language has also been used to scaffold latent representation learning (Frome et al., 2013; Jia & Liang, 2016; Andreas et al., 2017; Ye et al., 2020b; Goyal et al., 2020; Liang et al., 2020; Mu et al., 2019; Luketina et al., 2019), and as a high-level specification for program synthesis tasks (Ye et al., 2020a; Nye et al., 2019; Polosukhin & Skidanov, 2018; Ye et al., 2020b; Desai et al., 2016; Srivastava et al., 2017). Here we present an approach that integrates language annotations in training for learning a more generalizable library and program search model that can be used after training with no additional annotations for new tasks.

3. Inductive synthesis and library learning

Consider the problem of writing a graphics program to draw the large hexagon image in the left column of Fig. 1. This is an inductive program synthesis problem: a task $t$ (like draw a large hexagon) is specified with examples of what a program should do, where each example is given as an input $x$ (in this case, the blank image canvas) and the desired output $y$ (the large hexagon image). A program $\rho$ solves the task if it produces outputs that are consistent with the specification when executed — that is, if evaluating $\rho$ under an execution model $E$ yields $\mathbb{E}[\rho|E](x) = y$.

Program synthesis begins with a library $\mathcal{L} = \{l_0, l_n\}$ containing the set of primitives that can be combined to produce solution programs, such as the pseudo-code primitive functions in a simple graphics language:

$$\mathcal{L} = \text{move}_{\text{pen}}|\text{unit}_{\text{line}}|\text{for}[*]|\pi|\infty|0|1|2|...$$

which draw lines on a canvas parameterized by their length and angle. Given a library, there is also the problem of search: effective program synthesis requires a search strategy $S$ that can be given a task specification (such as the image of a hexagon) and automatically discover a solution program like the one shown in Fig. 1:

$$(\text{for } \infty \text{(move}_{\text{pen}}^{*} | \text{unit}_{\text{line}} 3)/(2\pi 6))$$

by searching over programs built from functions in $\mathcal{L}$. 

---

Figure 1. Our model, Language for Abstraction and Program Search (LAPS) integrates natural language into base learned synthesis algorithms formulated as hierarchical Bayesian inference (A, left) for jointly learning a library of program abstractions and a neural search heuristic for synthesis. We give an extended formulation (B, left) defined jointly over the program library and natural language descriptions of synthesis tasks, that can be used to incorporate natural language into both abstraction and search heuristic learning. When incorporated into a concrete learning algorithm, DreamCoder (A, right) we show that LAPS allows the model to leverage language richly during training to improve the generalization of both the learned neural search model and the learned library of program abstractions.
Both of these ingredients – the library $\mathcal{L}$, and the search strategy $S$ – can be made much more efficient if the synthesis engine will be expected to solve multiple related problems. In the graphics domain, for example, synthesis of the various images depicted in Fig. 1 is much more easily accomplished using a library like

$$\mathcal{L} = \text{polygon} | \text{large\_line} | \text{small\_line}...$$

in which the original hexagon task can be expressed as

$$\text{polygon}(6, \text{large\_line})$$

A good library already provides a foundation for efficient search by making solutions easier to express. Even with such a library, search can be further guided by information about the prior structure of programs (for example, the fact that polygon is typically called with a large\_line or small\_line function as a second argument) and by information about the target task itself (for example, the fact that the target image contains six line segments). Thus, one way to describe an effective search strategy is via a prior over programs $P[\rho|\mathcal{L}]$ in the library and a conditional inference model for inferring $P[\rho|t, \mathcal{L}]$, the distribution over programs likely intended by the observed task examples $t$.

The foregoing discussion lays out the basic ingredients of a hierarchical Bayesian formulation of program synthesis (used in learning algorithms like (Ellis et al., 2021; Lake et al., 2015; Dechter et al., 2013); see the graphical model in Fig. 1A, left) for jointly learning a library and conditional search model from a dataset $T$ of synthesis tasks. We denote a prior over programs as $P[\rho|\mathcal{L}, \theta_\mathcal{L}]$, on a library $\mathcal{L}$ with parameters $\theta_\mathcal{L}$. Given the observed tasks, we define the likelihood of the latent library and parameters as:

$$\Phi(\mathcal{L}, \theta_\mathcal{L}) = P[\mathcal{L}, \theta_\mathcal{L}] \prod_{t \in T} P[t|\rho]P[\rho|\mathcal{L}, \theta_\mathcal{L}]$$

(1)

where $P[\mathcal{L}, \theta_\mathcal{L}]$ is a prior over all possible libraries and parameterizations, and $P[t|\rho]$ is the likelihood that each inductive task $t$ is consistent with a program $\rho$ (for our purposes, $P[t|\rho] = 1$ if the program produces the desired output examples and 0 otherwise.) Learning in this model means estimating the optimal library and its parameters

$$\mathcal{L}^* = \arg \max_{\mathcal{L}} \int \Phi(\mathcal{L}, \theta_\mathcal{L}) \; d\theta_\mathcal{L} \quad \theta_\mathcal{L}^* = \arg \max_{\theta_\mathcal{L}} \Phi(\mathcal{L}^*, \theta_\mathcal{L})$$

(2)

along with a conditional model $P[\rho|t, \mathcal{L}^*]$ that can infer programs for new tasks.

This formulation also foreshadows a straightforward way in which linguistic descriptions of tasks (like those in the first column of Fig. 1) could be integrated into learning: we could simply extend the conditional model as $P[\rho|t, d_t, \mathcal{L}^*]$ to include a task’s description $d_t$. We come back to this (and describe a more complete integration) in our approach, but first describe a concrete implementation of Eq. 2 on which we can realize the language-enriched model.

4. Base learning algorithm: DreamCoder

The LAPS framework we describe in this paper is a general one for extending Bayesian models of program learning like the one in Eq. 2 to incorporate information from language. For concreteness, however, our presentation and experiments build on the specific DreamCoder algorithm of Ellis et al. (2021), which we briefly review here. We choose DreamCoder because it exposes a modular implementation of the library and search learning problems in Eq. 2 and has previously demonstrated state-of-the-art performance across a variety of synthesis domains (Ellis et al., 2021; 2020).

DreamCoder is initialized with a base library $\mathcal{L}_0$ of starting primitives and a dataset of training tasks $T$. It returns a learned final library $\mathcal{L}_f$ augmented with program abstractions and a learned neural search model $Q(\rho|t, \mathcal{L})$ that predicts high probability programs conditioned on the task examples. Learning is iterative: DreamCoder alternately searches for solution programs to the training tasks (given a current library $\mathcal{L}_t$ and search model $Q_t$) and updates the library and search model based on new solved tasks. We give details on each component below.

4.1. Program prior

DreamCoder defines the prior over programs as a probabilistic context free grammar (PCFG; Johnson 1998) for programs generated as productions from a library $\mathcal{L}$ of functions $l \in \mathcal{L}$. Formally, DreamCoder assigns a real-valued weight $\theta_{l, \mathcal{L}}$ to each library function, which when normalized yields a production probability $P[l|\mathcal{L}, \theta_\mathcal{L}]$. The prior probability of a program $\rho$ is given by

$$P[\rho|\mathcal{L}, \theta_\mathcal{L}] = \prod_{l \in \rho} P[l|\mathcal{L}, \theta_\mathcal{L}]$$

(3)

the weighted product of probabilities of all of its constituent library functions. As all $P[l|\mathcal{L}, \theta_\mathcal{L}] < 1$, this is equivalent to a description length prior over programs: longer programs (with more constituent elements) will have lower prior probability under Eq. 3 since $P[l|\mathcal{L}, \theta_\mathcal{L}]$ monotonically decreases as $|\rho| = |\{l \in \rho\}|$ increases.

4.2. Amortized conditional inference

To identify programs that solve tasks $t$ while obtaining high probability under $P[\rho|\mathcal{L}, \theta_\mathcal{L}]$, DreamCoder trains a neural...
search heuristic \(Q_i(\rho|t, \mathcal{L}_i)\) at each iteration \(i\) to approximate the inverse conditional model. The heuristic uses a neural model trained to predict programs written in the current library \(\mathcal{L}_i\) according to the posterior:

\[
Q_i(\rho|t, \mathcal{L}_i) \approx P[\rho|t, (\mathcal{L}_i, \theta_{\mathcal{L}_i})] \propto P[t|\rho]P[\rho|(\mathcal{L}_i, \theta_{\mathcal{L}_i})]
\]

conditioned on an encoding of the training examples (e.g. an embedding of the image in the task specification). This model is trained in the distant supervision setting (which begins with no supervised program data) by leveraging the forward generative model: sampling programs from the prior, executing them to produce observed tasks, and then minimizing \(Q(\rho|t, \mathcal{L})\) in Eq. 4 on the sampled programs, conditioned on their executions. This generative training procedure is generally applicable to any neural implementation of \(Q(\rho|t, \mathcal{L})\). (But see Ellis et al. (2021) and our supplementary material for additional details on the model architecture, which we reimplement in our experiments).

4.3. Abstraction learning as program compression (maximizing the likelihood of programs)

The DreamCoder algorithm also iteratively updates the library \((\mathcal{L}_i, \theta_{\mathcal{L}_i})\) to approximately optimize Eq. 2 (finding \(\mathcal{L}^*, \theta^*_\mathcal{L}\) which maximize the likelihood of the inferred latent programs). Ellis et al. (2021) leverage equivalence to a compression problem defined over programs and the library. As discussed in 4.1, the PCFG program prior is equivalent to a description length prior over programs. Ellis et al. (2021) place an additional Dirichlet prior over the library description length:

\[
P[\mathcal{L}] \propto \exp\left(-\lambda \sum_{\rho \in \mathcal{L}} \text{size}(\rho)\right)
\]

Estimating the optimal library then becomes the problem of inferring new library abstractions which can jointly compress the latent training programs (rewritten under the new library \(\mathcal{L}_{i+1}\)) and the description length \(|\mathcal{L}_{i+1}|\) of the updated library (to optimize for shared abstractions across programs). This objective would still require inference over all possible ways of refactoring the latent programs under the updated library. Ellis et al. (2021) approximate this by only considering candidate abstractions and program refactorings that can be found via an efficient lambda-abstraction algorithm. As an example, this could refactor the large hexagon program

\[
\text{(for } \infty \text{move_pen(* unit_line 3)*/ 2π 6)}
\]

to expose a candidate abstraction like

\[
\lambda x. (\text{for } \infty \text{move_pen(* unit_line 3)*/ 2π x})
\]

while also rewriting the original program using this abstraction. Notably, this fragment – which draws polygons with lines of length 3 for sides – is not the most intuitively generalizable for the graphics domain. A programmer with more domain-specific prior knowledge would probably prefer an abstraction like

\[
\lambda x. (\text{for } \infty \text{move_pen(* unit_line y)*/ 2π x})
\]

which additionally parameterizes the polygon by the length of its sides, and is semantically equivalent to the high-level polygon_fn described in the problem setup in Sec. 3. However, learning abstractions by compressing the library and current solved training tasks may actually disfavor this more intuitively generalizable (but less compressive) candidate. Our second key goal in introducing language will be to leverage it as an additional source of prior knowledge to improve abstraction generalization.

5. Our Approach: Language for Abstraction and Program Search

Our work considers how the general learning problem – jointly learning the library \(\mathcal{L}\) which defines the prior over programs and the conditional search strategy \(S\) which inverts from tasks to programs – can be enriched in the language-annotated setting. Here, at least a subset of the training tasks are additionally annotated with a natural language description \(d_i\) (such as the natural language description large six gon for the large hexagon drawing task in Fig. 1B). Language offers a more direct source of information for discovering a library like the one in our setup,

\[
\mathcal{L} = \text{polygon|large_line|small_line...}
\]

if we leverage the expectation that generalizable abstractions (like a candidate polygon function) should correspond systematically to named fragments in natural language (like the token gon).

Language can also be leveraged by the conditional search model: learning systematic correspondences between language and programs from descriptions like large six gon should inform search on new tasks (like the one described as a small nine gon next to a small square in Fig. 1B)) on the basis of shared language (like gon).

Our approach, LAPS (Language for Abstraction and Program Search) formalizes these intuitions by extending the hierarchical Bayesian problem formulation over programs given in Sec. 3 to additionally generate natural language task descriptions (see graphical model in Fig 1B, left). In particular, we assume the existence of a jointly generative model \(J(\rho, d_t)\) over latent programs that solve tasks, and corresponding natural language descriptions. We rewrite the original prior over programs \(P[\rho|\mathcal{L}, \theta_{\mathcal{L}}]\) defined on a library \(\mathcal{L}\) to a joint prior \(P[\rho, d_t|J, \theta_J]\), and extend the distribution in Eq. 1 over the latent joint model \(J\) with parameters \(\theta_J\),
written as
\[
\Phi(J, \theta_j) = P[J, \theta_j] \prod_{t \in T} \sum_{\rho} P[t|\rho]P[\rho, d_t|J, \theta_j]
\]  
(6)

Learning in the language-augmented setting now involves estimating the optimal joint model and its parameters
\[
J^* = \arg \max \int \Phi(J, \theta_j) d\theta_j \quad \theta^*_j = \arg \max \Phi(J^*, \theta_j)
\]
along with a language-conditioned model \(P[\rho|t, d, J^*]\) that can infer programs for new tasks based on both specification examples and task descriptions.

In the remainder of this section we first describe a general joint model formulation that can be learned from language-annotated training tasks. We then show how the joint framework allows natural language to inform learning at both the abstraction and search level in a concrete example, using DreamCoder as the base hierarchical algorithm.

5.1. Joint prior over programs and language

**Base prior** We formulate our joint prior over language and programs as
\[
P[\rho, d_t] = P[\rho, \mathcal{L}, \theta_L]P[d_t|\rho, \mathcal{L}]
\]
(8) 

decomposed as the product of the original program prior defined on a program library \(P[\rho|\mathcal{L}, \theta_L]\), and a learned program-to-natural-language “translation” model \(T(d_t|\rho, \mathcal{L}) \approx P[d_t|\rho, \mathcal{L}]\) which describes how natural language descriptions are generated for latent programs (in our running example, this model would describe how the large six gon description was generated conditioned on the program solution for that task.) This decomposition builds modularly on the original program prior defined only on the library \(\mathcal{L}\). Learning \(T(d_t|\rho, \mathcal{L})\) formalizes the intuition that there should be a learnable relationship between language that describes tasks and latent programs that solve them.

\(T(d_t|\rho, \mathcal{L})\) can be implemented in many ways (e.g. (Wong & Mooney, 2007; Joshi & Schabes, 1997; Bahdanau et al., 2014; Chen et al., 2018)), compatible with the vast literature on structured translation between languages, including natural languages and programming languages. Our experiments use the translation model popularly known as IBM Model 4 (Brown et al., 1993), one of a class of well-studied Bayesian machine translation models (Gal & Blunsom, 2013) which decompose \(T(d_t|\rho, \mathcal{L})\) into
\[
T(d_t|\rho, \mathcal{L}) \propto \prod_{w \in d_t, l \in \rho} P_T[w|l]
\]
(9)
a product of learned token-level translation probabilities \(P_T[w|l]\) between individual functions \(l\) in a task’s latent program \(\rho\) and words \(w\) in the task description \(d_t\). (See supplementary materials for model implementation and training details.) This token-level decomposition more directly captures the intuition in our setup: that abstractions in a programming library generally correspond systematically to individual names in natural language descriptions, and that the inverse conditional search can be guided based on a generally compositional relationship between program primitives and words. This formulation also allows these compositional relationships to be inferred from fewer observed examples than would be possible with other translation models with weaker inductive biases. However, Eq. 8 should extend to include any similar translation model and need not include this stronger decomposition.

**Adding richer priors** In LAPS, the joint model can also provide a controllable interface for incorporating additional prior knowledge about language into learning. Learned translation models are often fit to only maximize the likelihood of the observed language (here, with respect to inferred latent training programs). However, our formulation also supports \(T(d_t|\rho, \mathcal{L})\) enriched to include additional priors over language (such as speaker-specific language usage, or pragmatics models that capture a speakers’ other communicative goals (Grice, 1989; Goodman & Frank, 2016).)

In our experiments (Sec. 6.1) we showcase this with results from an extended model incorporating an additional mutual exclusivity prior. Mutual exclusivity models the expectation that newly encountered words should correspond to different meanings than known ones. This prior has been shown to play an important role in language learning in cognitive science (Frank et al., 2009; Markman & Wachtel, 1988), and in machine learning models (Gandhi & Lake, 2019).

In the synthesis setting, mutual exclusivity can capture the expectation that “new” words (which appear in descriptions of currently unsolved tasks) are more likely to correspond to different program components than those used in solved training tasks (and for which there would otherwise be no signal to learn a translation model in the distant setting). Our extended model incorporates this prior by updating Eq. 9 to distinguish between \(W_{\text{known}}\) (words that appear in solved training tasks with latent programs) and \(W_{\text{new}}\) (newly encountered words) as
\[
T_{ME}(d_t|\rho, \mathcal{L}) \propto \prod_{w \in d_t, l \in \rho} (1[w \in W_{\text{known}}]P_T[w|l])
\]
\[
(1[w \in W_{\text{new}}]P[l|\mathcal{L}, \theta_L]^{-1}))
\]
(10)
where new words are modeled as inversely related to primitives under the program prior (fit to previously solved tasks) – modeling the expectation that new words more likely relate to less-used program components than those used so far.
5.2. Integrating the joint model into amortized conditional search

The joint model allows LAPS to incorporate natural language into the learned conditional search model over programs. In place of the original neural amortized model in the base algorithm (Sec. 4.2), we train an extended, language-conditioned model \( Q_i(t|\theta_l, J_i) \) at each iteration to predict programs according to:

\[
Q(t|\theta_l, J_i) \approx P(t|\theta_l, J_i, J, \theta_J) \\
\propto P(t|\theta_l)P(\theta_l, J, \theta_J) \\
\propto P(t|\theta_l)P(\theta_l|J, \theta_J)P(t|J, \theta_J) \\
\approx P(t|\theta_l)\mathcal{T}(\rho, J, \theta_J)P(\rho, J, \theta_J)
\]

which amortizes program inference under our joint model formulation. Importantly, we can train this neural model using samples from the joint generative model, consisting of sampled programs and corresponding generated language. As with the original learning setting, this sample-based training allows LAPS to learn a generalizable, language-conditioned neural search heuristic, capable of leveraging compositional patterns in natural language, from very few examples in the distant supervision setting. We can also now see the benefits of richer language-specific priors (such as mutual exclusivity); the neural model trained to amortize inference from the joint generative model can also approximate the mutual exclusivity bias, enabling better exploration and generalization in the presence of new words.

5.3. Abstraction learning as joint model compression

The extended joint model objective in Eq. 2 and 7 also allows LAPS to incorporate natural language into abstraction learning. Extending the compression-based abstraction objective in the base algorithm – which optimized for libraries that maximally compress the latent training programs and library – requires defining a prior over the language-program translation model \( \mathcal{T} \) in terms of the optimal program library.

We place a prior over \( \mathcal{T} \) defined on a program library \( \mathcal{L} \) and a natural language token vocabulary \( W \) as

\[
P(\mathcal{T} | \mathcal{L}) \propto \sum_{t \in \mathcal{L}, w \in W} -I(P(\mathcal{T}[w|t]))
\]

where \(-I(P_T[w|t]) = - \log(P_T[w|t])\). This models the intuition that a good library contains program abstractions which correspond well to individual language tokens, and reduce entropy in the compositional translation model. Defining the prior compositionally also allows the algorithm to maintain the desirably property from (Ellis et al., 2021), in which the joint likelihood can be efficiently re-approximated with respect to individual candidate program abstractions based on their constituent subcomponents \( l \) and corresponding translation distributions \( P_T[w|l] \) under the current translation model. As in the base synthesis algorithm, we fully re-estimate a new translation model at each iteration \( \mathcal{T}_{i+1}(\rho, \mathcal{L}_{i+1}) \) to fit the updated library and refactored programs. See the supplement for extended details.

Taken together, Alg. 1 summarizes the concrete algorithm using LAPS to incorporate language into (Ellis et al., 2021).

### 6. Experiments

We demonstrate LAPS on three different domains: string editing, compositional graphics drawing, and scene reasoning, which we choose to represent a diverse range of tasks and accompanying language (Fig. 2). In all three domains, we find that compared to the base synthesizer, LAPS learns and solves holdout synthesis problems faster (Table 1, Sec. 1-2), and produces higher-quality libraries that improve generalization even when natural language hints are not available after training (Table 1, Sec. 3).

Below we summarize each domain. We then discuss results showing that LAPS is effective because of how the hierarchical model incorporates language during learning: we find that (1) LAPS searches more effectively during training, enabling it to solve and learn from more diverse training tasks than the baseline model; (2) LAPS abstracts more effectively during training, adding in more generalizable library routines as it learns; and (3) LAPS can use language during testing if it is available, as an important additional source of high-level information during synthesis.

#### 6.1. Domains

All three domains consist of a dataset of inductive synthesis tasks \( t \) specified as input/output examples; procedurally generated synthetic language annotations; and human language annotations sourced from Mechanical Turk. We use synthetic language as our primary evaluation benchmark; we are interested in a controlled probe of learning when words are systematically reused and composed, but refer to more abstract concepts than in the initial base programming language. However, we also use human language to evaluate the practicability of our approach in real-world settings. Additional information for all domains is in the supplement.
A. String Editing (shown with sample I/O examples of n=30 and random human description of n=3)

- Original CLEVR
  - how many grey rubber cubes do you see
  - What number of gray rubber cubes are there?
- (Human) if word begins with consonant followed by vowel, add an g before the beginning
- (Synth) if the word starts with a consonant and s then change them both to b
- bloomed ➔ cultivator ➔ gcultivator
gcools ➔ cools

B. Scene Reasoning (shown with sample I/O examples of n=7 and random human description of n=2)

Original CLEVR (sample templates from full set)

<table>
<thead>
<tr>
<th>Original CLEVR</th>
<th>Extended scene manipulation and counterfactuals</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Image 1]</td>
<td>![Image 2]</td>
</tr>
<tr>
<td>![Image 3]</td>
<td>![Image 4]</td>
</tr>
<tr>
<td>![Image 5]</td>
<td>![Image 6]</td>
</tr>
<tr>
<td>![Image 7]</td>
<td>![Image 8]</td>
</tr>
<tr>
<td>![Image 9]</td>
<td>![Image 10]</td>
</tr>
<tr>
<td>![Image 11]</td>
<td>![Image 12]</td>
</tr>
</tbody>
</table>

B. Scene Reasoning (shown with sample I/O examples of n=7 and random human description of n=2)

- What number of gray rubber cubes are there?
- If you removed the red things, how many spheres would be left?
- What if the gray sphere became a small green metal sphere?
- If there is u any letter double that

C. Compositional Graphics (shown with random human description of n=3)

- Simple shapes
  - a small triangle
  - a medium square
  - a medium eight gon
  - a big circle
- Complex objects
  - a seven pointed star
  - a four stepped zigzag
  - a greek spiral with eight turns
- Compositional objects and relations
  - a small five gon next to a small seven gon
  - a small nine gon separated by a big space from a small circle
  - a small triangle connected by a big line to a medium triangle

D. Example initial graphics primitives...and example program abstractions learned with language shown with learned high probability p(word | primitive)

- for move pen-up + − 1 2
- ![Image 13] ![Image 14] ...
- f0=λ(x y z) (for x (λ (u v) (move y v))))
- f4=(λ (x y z) (f0 x y z))
- ![Image 15] ...
- f3=(λ (y) (f2 (λ y (f16 y))))
- ![Image 16] ...
- f24=(λ (x y) (f23 (λ (z u) (f21 u w))))
- ![Image 17] ...

Figure 2. (A, B, C) Example tasks from all three synthesis domains shown with synthetic and sample human language annotations. Inductive synthesis domains are shown with a random subset (n=3) of the paired input/output examples. Human language annotations are also randomly sampled (all domains were annotated by multiple people for a broader range of language.) (D) Representative initial program primitives and library abstractions learned with LAPS for the graphics domain. Shown with example tasks solved with synthesized programs containing the learned abstractions and high probability natural language learned from the joint model.
String editing: structured string transformation problems taken from (Andreas et al., 2017) (n=1000 train; n=500 test). Tasks consist of input dictionary strings transformed using randomly sampled regular expression transducer (30 I/O examples per task). We choose this domain to demonstrate LAPS on an important classic synthesis domain (Lau & Weld, 1998). The dataset of Andreas et al. (2017) contains human annotations; synthetic language annotations are generated over the ground-truth regexes using templates based on the original human annotations. We initialize synthesizers with functional programming primitives (map, fold, cons, car, cdr, length, index) and character constants (following the simpler text editing domain in the baseline paper (Ellis et al., 2021)). The neural search model encodes the I/O task examples as character arrays with a bidirectional GRU.

Compositional graphics: inverse graphics problems (n=200 train; n=111 test) where each task is specified by an image and solved by synthesizing a program in LOGO Turtle graphics (Abelson & DiSessa, 1986). This is inspired by the graphics domain in (Ellis et al., 2021) but re-designed to be more challenging (ground-truth programs are much longer on average in the base programming language) and explicitly compositional. Synthetic language annotations are generated with high-level templates over the objects and relations in each task; human annotations are sourced as image descriptions from MTurk. We initialize synthesizers with the graphics primitives in (Ellis et al., 2021). The neural model encodes image examples with a CNN.

Structured scene reasoning: inductive scene reasoning tasks (n=212 train; n=115 test) where each synthesis problem is specified by a structured input scene, and outputs can be a number (how many red rubber things are there?), a boolean value (are there more blue things than green?), or another scene (what if all of the red things turned blue?). This domain is modeled on CLEVR (Johnson et al., 2017a) but designed to support inductive synthesis tasks specified over the symbolic scene representations (an array of objects represented as dictionaries of attributes) from the original CLEVR task generator in Johnson et al. (2017a). We also add new tasks that require generating or imagining latent scenes (how many metal things would be left if all the blue cylinders were removed?), which are not solvable in the original high-level DSL hand-designed for Johnson et al. (2017b) (and used in synthesis-based approaches like Yi et al. (2018)). We include these to demonstrate a key feature of our approach: the ability to learn generalizable libraries from a basic but expressive set of primitives, rather than restricting the program space pre-emptively with a hand-designed language. We use synthetic language annotations from the original templates in (Johnson et al., 2017a) (and templates written in the same style for the extended tasks); human annotations are sourced from annotators shown the same tasks. We initialize synthesizers with functional programming primitives similar to the string-editing domain, with domain-specific query functions and constants (get_color(x); get_shape(x); blue; cube). The neural model encodes the task examples as flattened arrays of object attributes using a bidirectional GRU.

6.2. Results

On all three domains, we compare our model against the baseline synthesizer (Table 1, DreamCoder, no language); a multimodal baseline (Table 1, multimodal, no generative model) that trains a neural model directly on solved training tasks (similar to neural synthesis models like DeepCoder (Devlin et al., 2017) but augmented to condition on language); and ablated LAPS variants (Table 1; LAPS rows) to evaluate the additive contributions of the individual learning components. We compare all models using a matched search budget per task and number of training iterations overall, determined using a hyperparameter search with the baseline. The supplement contains full details (and code) to replicate all experiments; and additional qualitative results.

We find that:

1. LAPS searches more effectively during training, enabling it to solve and learn from more training tasks than the baseline synthesizer. Under the hierarchical model formulation, search and abstraction are closely related: successfully solving tasks is the basis for abstraction learning.

Comparing the model learning trajectories (Fig. 3) on training tasks shows that the LAPS models consistently search more effectively during training: at each iteration they solve more tasks within a given time budget. Fig. 3 also highlights that LAPS models improve training robustness in the distant learning setting: as in the baseline paper (Ellis et al., 2021), we find the baseline model learning to be highly variable without a training curriculum (compare training curves from Fig. 3 with different random seed replications; and the best vs. mean performance, Table 1.) Comparing the LAPS ablations also suggests that linguistic priors (like mutual exclusivity) can indeed be practically useful here during learning (Table 1, compare LAPS with ME and without).

What if we do use a curriculum? In the scene reasoning domain (where previous approaches (e.g. Mao et al. 2019) have argued for a curriculum), we also test a simple curriculum by ordering tasks according to their natural language token length (which can be evaluated without ground truth programs). Table 1 shows that our model is still more effective, and that non-curriculum performance is in fact comparable to curriculum performance.

(2) LAPS abstracts more effectively during training, adding in more generalizable library routines as it learns. The variability across training replications in the baselines also highlights a challenge for abstraction learning: not all shared
Leveraging Language to Learn Program Abstractions and Search Heuristics

Table 1. % held-out test-tasks solved. To compare robustness, we run random seed replications in the graphics domain for the synthetic language dataset. Best reports the best model across replications; Mean averages across replications.

<table>
<thead>
<tr>
<th>Language</th>
<th>Model</th>
<th>Strings (n_{rep} = 500)</th>
<th>Graphics (n_{rep} = 111)</th>
<th>Scenes (n_{rep} = 115)</th>
<th>% Solved</th>
<th>% Solved (Best)</th>
<th>% Solved (Mean)</th>
<th>% Solved (Curric.)</th>
<th>% Solved (Mean.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>synth train/test</td>
<td>DreamCoder (no language)</td>
<td>33.4</td>
<td>49.55</td>
<td>42.64</td>
<td>67.80</td>
<td>73.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>synth train/test</td>
<td>Multimodal (no generative translation model)</td>
<td>46.00</td>
<td>28.12</td>
<td>23.20</td>
<td>76.50</td>
<td>49.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>synth train/test</td>
<td>LAPS in neural search</td>
<td>52.20</td>
<td>92.79</td>
<td>52.93</td>
<td>95.6</td>
<td>88.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>synth train/test</td>
<td>LAPS + mutual exclusivity</td>
<td>57.00</td>
<td>86.49</td>
<td>80.18</td>
<td>96.5</td>
<td>82.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>synth train/test</td>
<td>LAPS + ME + language-program compression</td>
<td>54.60</td>
<td>98.19</td>
<td>81.98</td>
<td>95.6</td>
<td>95.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>synth train/human test</td>
<td>LAPS + ME + language-program compression</td>
<td>54.60</td>
<td>89.20</td>
<td>–</td>
<td>97.4</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>human train/human test</td>
<td>LAPS + ME + language-program compression</td>
<td>48.60</td>
<td>58.55</td>
<td>–</td>
<td>95.6</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No language at test

| No language on train/test | Original DSL: Enumerative | 0.00                     | 0.00                      | –                       | 27.8     | –               |                 |                  |                 |
| No language on train/test | DreamCoder (best library): Enumerative | 27.2                     | 41.44                     | –                       | 53.6     | –               |                 |                  |                 |
| No lang at test           | LAPS (best library): Enumerative | 33.2                     | 62.16                     | –                       | 93.04    | –               |                 |                  |                 |
| No lang at test           | LAPS (best library): example-only neural synthesis | 52.4                     | 91.0                      | –                       | 95.6     | –               |                 |                  |                 |

Figure 3. Learning curves comparing baselines and LAPS models in Table 1, showing % heldout tasks solved on the graphics domain over random training task orderings. (Mean results in Table 1 shows average test-time performance from the trained model replications.)

subroutines encountered in training generalize well to new tasks. Adding poor abstractions can actually be detrimental: they increase the combinatorial search space. We find that our approach produces higher-quality libraries after training: Table 1 (no language at test time section) shows that we consistently improve performance in a head-to-head comparison using enumerative search from the library priors alone – in some domains, enumerative search with our model’s library outperforms neurally guided search from the baseline model. We also find the learned library is effective for neurally-guided synthesis when no language hints are available after training (Table 1, no language at test, example-guided synthesis), showing that LAPS incorporates language to learn a more effective library overall, which generalizes to the non-language setting. See supplement for example learned abstractions from $\tilde{E}_f$.

(3) **LAPS can use language during testing if it is available, though it doesn’t need to for competitive performance.** Clearly, language can provide a useful source of high-level information if it is available for new tasks. Our approach produces a neural synthesizer pre-trained to condition on language where available. Results on all three domains show that the model can use it to achieve additional performance gains (Table 1, see language at test rows). We also find that the models trained on synthetic annotations generalize effectively to natural human language at test (Table 1, synth train, human test), suggesting that even if human annotation is too costly, in many cases hand-writing natural language templates to accompany a few ground-truth programs is likely sufficient (and easier than hand designing a full DSL).

7. Conclusion

We presented **Language for Abstraction and Program Search** (LAPS). LAPS builds on hierarchical Bayesian models of program learning: we offer a general framework for introducing jointly generative models over programs and language into learned synthesis. Going forwards, an important avenue for future work will be exploring different concrete implementations of the base algorithm and translation model which relates programs to language. A promising future direction could leverage recent structured, neural joint language learning with those for studying non-linguistic abstraction and search (e.g. Smith et al. 2003; Hawkins et al. 2019; Lake et al. 2015; 2019; Tian et al. 2020).

The hierarchical Bayesian framing also draws connections to computational cognitive models which model human conceptual representations and learning (Goodman et al., 2014; Fodor, 1975; Rule, 2020) as inference over program-like representations. Future human experiments could explore LAPS as a cognitive model, combining paradigms for studying language learning with those for studying non-linguistic abstraction and search (e.g. Smith et al. 2003; Hawkins et al. 2019; Lake et al. 2015; 2019; Tian et al. 2020).

Acknowledgements: Many thanks to M. Nye, J. Mu, A. Marzoev, J. Fan, R. Hawkins, R. Levy, L. Schulz and our anonymous reviewers for invaluable feedback. Supported by grants from the Air Force Office of Scientific Research, the NSF under Grant No. 1918839 and NSF-funded Center for Brains, Minds, and Machines, the MIT-IBM Watson AI Lab, Google, Microsoft and Amazon.
References


Dumancić, S. and Cropper, A. Inventing abstractions by refactoring knowledge.


Leveraging Language to Learn Program Abstractions and Search Heuristics


Polosukhin, I. and Skidanov, A. Neural program search: Solving data processing tasks from description and examples. 2018.


