Leveraging Language for Abstraction and Program Search

Catherine Wong [1] **Kevin Ellis** [2] **Jacob Andreas** [1] **Joshua B. Tenenbaum** [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

The program synthesis paradigm – in which models automatically infer symbolic programs – offers many desirable properties for robust machine learning systems, including interpretability, verifiability, and strong generalization in few-shot learning settings (Gilpin et al., 2018; Appel et al., 2017; Lake et al., 2017). A diverse range of machine learning tasks can be formulated as program synthesis problems. These range from classic synthesis domains like data manipulation (Delaware et al., 2015; Gulwani et al., 2017) and semantic parsing (Artzi & Zettlemoyer, 2013; Liang, 2016); to structured reasoning problems in visual understanding (Johnson et al., 2017b; Yi et al., 2018), image generation (Ellis et al., 2017; Ganin et al., 2018; Du et al., 2018), and policy learning (Fikes & Nilsson, 1971; Cropper & Muggleton, 2015; Silver et al., 2020).

[*]Equal contribution [1]MIT [2]Cornell University. Correspondence to: Catherine Wong <catwong@mit.edu>.

Proceedings of the 37^{th} International Conference on Machine Learning, Online, PMLR 119, 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. In LAPS, language guides learning of both *libraries* of reusable functions and *search models* for synthesizing programs. Strong program libraries and search methods are the core ingredients of program synthesis (Gulwani et al., 2017). Recent approaches to program synthesis attempt to learn search models (Gulwani et al., 2015; Polozov & Gulwani, 2015; Balog et al., 2016; Devlin et al., 2017) and program libraries, often jointly with a search model over the library (Shin et al., 2019; Dumancić & Cropper; Ellis et al., 2020; Lázaro-Gredilla et al., 2019). However, even the current best learning approaches can be computationally inefficient (often upwards of thousands of CPU hours) 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 problem into named parts) and a lexicon that names the most important reusable concepts in a given domain (asked to write a graphics *library* for future problems, a sensible programmer would probably consider abstracting out a routine that draws parameterized *polygons* on the basis of the instruction wording alone).

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., 2020; 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 give the formal prob-

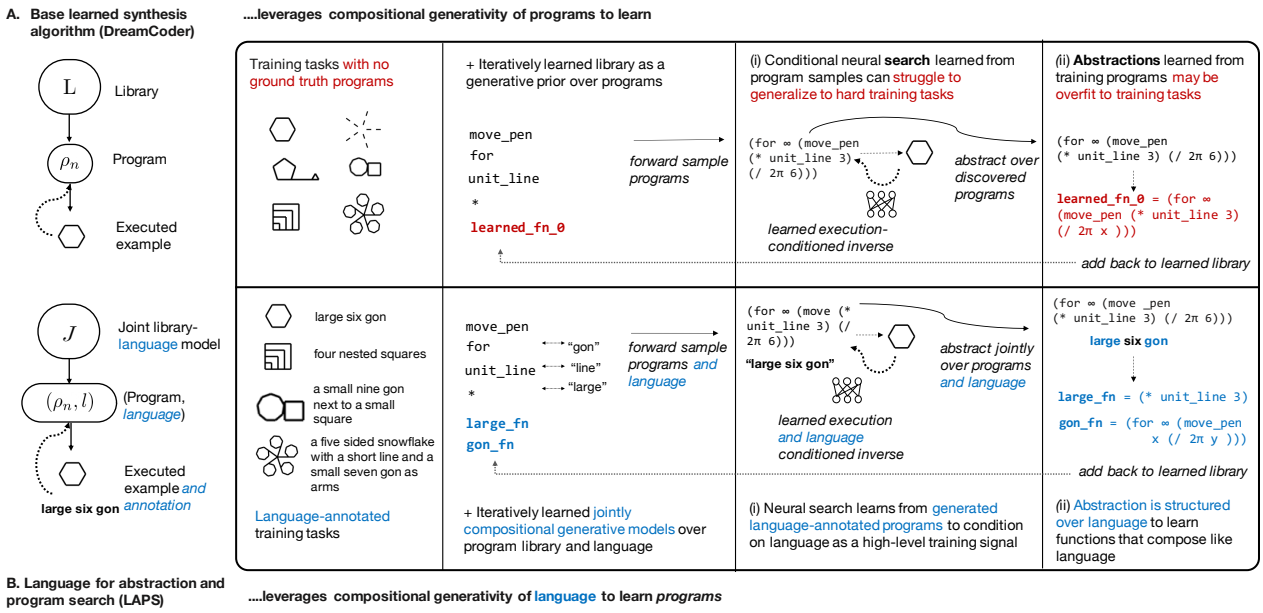

Figure 1. Our model, Language for Abstraction and Program Search (LAPS) integrates natural language into base learned synthesis algorithms formulated as hierarchical Bayesian inference (A, left) for jointly learning a **library** of program abstractions and a **neural search heuristic** for synthesis. 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.

lem formulation (Sec. 3) and baseline program synthesis approach (Sec. 4). We then describe how LAPS extends the base problem formulation to include language in learning (Sec. 5) and conclude with empirical results (Section 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; Polozov & Gulwani, 2015; Balog et al., 2016; Parisotto et al., 2016; Devlin et al., 2017; Shin et al., 2018; Alur et al., 2018; Kalyan et al., 2018; 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; 2020). Our computational formulation builds on *hierarchical Bayesian formulations of program learning* that frame both synthesis and library learning as probabilistic inference (Lin et al., 2014; Liang et al., 2010; Lake et al., 2015; Ellis et al., 2020).

Natural language has also been used to scaffold latent structure learning in both neural and symbolic representation learning algorithms (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 jointly learning a more generalizable library and the search algorithm over it that can be used without language on future 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 ρ solves the task if it produces outputs that are consistent with the specification when executed – that is, if evaluating ρ under an execution model E yields $[\![\rho]\!]_E(x) = y$.

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:

$$\mathcal{L} = \texttt{move\_pen} \,|\, \texttt{unit\_line} \,|\, \texttt{for} \,|\, * \,|\, \pi \,|\, \infty \,|\, 0 \,|\, 1 \,|\, 2 \,|\, ...$$

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:

```
(for ∞(move_pen(* unit_line 3)(/ 2π 6))
```

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

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 Sec. 1 is much more easily accomplished using a library like

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

in which the original hexagon task can be expressed as

```
polygon(6, 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 S is via a *prior* over programs $\mathcal{P}(\rho|\mathcal{L})$ in the library and a *conditional* inference model for inferring $P(\rho|t, \mathcal{L})$, the distribution over programs likely to satisfy 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., 2020; Lake et al., 2015; Dechter et al., 2013)) for jointly learning a library and search model from data (see the graphical model in Fig 1A, left). We can formally define the model's prior over programs as $P[\rho|\mathcal{L}, \theta]$, based on its library $\mathcal{L}$ and parameters θ that specify a distribution over programs that can be written using $\mathcal{L}$. The joint distribution over the observed tasks t and latent programs, library, and parameters, is:

$$\Phi(L, \theta) = \mathrm{P}[L, \theta] \prod_{t \in T} \sum_{\rho} \mathrm{P}[t|\rho]\mathrm{P}[\rho|L, \theta] \tag{1}$$

where $\mathrm{P}[L, \theta]$ is a prior over all possible libraries and parameters, and $\mathrm{P}[t|\rho]$ is the likelihood that the examples in t intended to specify the latent program ρ (for our purposes, $\mathrm{P}[t|\rho] = 1$ if the program produces the desired output examples and 0 otherwise.) Learning in this model involves estimating the optimal library and its parameters

$$L^* = \arg\max_L \int \Phi(L, \theta)\ \mathrm{d}\theta \quad \theta^* = \arg\max_\theta \Phi(L^*, \theta) \tag{2}$$

along with a conditional model $\mathrm{P}[\rho|t, L^*]$ 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 $\mathrm{P}[\rho|t, d, L^*]$ to include the description d. 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. 6 to incorporate information from language. For concreteness, however, our presentation and experiments build on the specific DreamCoder algorithm of (Ellis et al., 2020), which we briefly review here. We choose DreamCoder because it exposes a modular implementation of the library and search learning problems in Eq. 6 and has previously demonstrated state-of-the-art performance across a variety of synthesis domains (Ellis et al., 2020).

DreamCoder is initialized with a base library $\mathcal{L}_0$ of starting primitives and returns a library $\mathcal{L}_f$ containing program abstractions learned from solving training tasks, and a learned neural search model $Q(\rho|t, \mathcal{L})$ that predicts high probability programs, conditioned on the task examples. Learning involves an alternating search over solution programs to the training tasks (given a current library $\mathcal{L}_i$ and search model Q_i) and updates to the library and search model based on new solved tasks. We give details on each component below.

4.1. Program prior

Given a library $\mathcal{L}$ consisting of one or more functions $l \in L$, DreamCoder defines the prior over programs as a probabilistic context free grammar (PFCG) (Johnson, 1998) where programs are productions generated by composing weighted functions $l \in \mathcal{L}$. Formally, DreamCoder assigns a real-valued weight θ_i to each library function $l_i \in \mathcal{L}$ (which can be normalized to give the probability $P[l|\mathcal{L}, \theta]$ for each primitive). The prior probability of any program ρ written using primitives in the library is then given by

$$P[\rho|\mathcal{L}, \theta] = \prod_{l \in \rho} P[l|\mathcal{L}, \theta] \tag{3}$$

the weighted product of probabilities of all of its constituent primitives. As all $P[l|\mathcal{L}, \theta] < 1$, this is equivalent to a *description length* prior over programs written using the library: longer programs (with more constitutent primitives) will have lower prior probability under Eq. 3 since $P[l|\mathcal{L}, \theta]$ 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 $P[\rho|\mathcal{L},\theta]$, 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:

$$Q_i(\rho|t,L) \approx \mathrm{P}[\rho|t,(L_i,\theta_i)] \propto \mathrm{P}[t|\rho]\mathrm{P}[\rho|(L_i,\theta_i)] \quad (4)$$

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 $Q(\rho|t,L)$ 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., 2020) 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 $(\mathcal{L}_i,\theta_i)$ to approximately optimize Eq. 2 (finding L^*,θ^* which maximize the joint distribution over the inferred latent programs, library, and its parameters). (Ellis et al., 2020) 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., 2020) place an additional Dirichlet prior over the library description length:

$$\mathrm{P}\left[\mathcal{L}\right] \propto \exp\left(-\lambda \sum_{\rho\in L} \mathrm{size}(\rho)\right) \quad (5)$$

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 L_{i+1}) and the description length $|L_{i+1}|$ 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., 2020) approximate this by only considering candidate abstractions and program refactorings that can be found based on an efficient algorithm based on lambda-abstraction, which in our simplified notation could refactor the large hexagon program

```
(for ∞(move_pen(* unit_line 3)(/ 2π 6))
```

to expose a candidate semantic fragment like

```
λx.(for ∞(move_pen(* unit_line 3)(/ 2π x))
```

while jointly 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

```
λxy.(for ∞(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_t, as in Fig. 1B, such as the natural language description *large six gon* for the large hexagon drawing task. Language offers a more direct source of information for discovering a **library** like the one in our setup,

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

if we leverage the intuitive prior expectation that generalizable abstractions (like a candidate `polygon` function) should correspond systematically to compositional 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 should make it easier to generalize from descriptions like *large six gon* to guide conditional search on complex but compositionally related tasks (like the *small nine gon next to a small square in Fig. 1B)* on the basis of shared words 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 include a joint definition over *natural language* (see graphical model in Fig 1B, left). In particular, we assume the existence of a *jointly* generative model $J(\rho,d)$ over both programs *and their natural language descriptions*, which allows us to extend the original prior over

programs $P[\rho|\mathcal{L},\theta]$ defined on the library $\mathcal{L}$ to the *joint prior* $P[\rho,d|J,\theta]$ where θ is generalized to the parameters of the *joint* model. The extended distribution over the observed task examples t, latent programs ρ, descriptions d, and joint model J with parameters θ becomes

$$\Phi(J,\theta) = \mathrm{P}[J,\theta]\prod_{t\in T}\sum_{\rho}\mathrm{P}[t|\rho]\mathrm{P}[\rho,d|J,\theta] \tag{6}$$

Learning in the joint setting now involves estimating the optimal joint model and its parameters

$$J^* = \arg\max_J \int \Phi(J,\theta)\ \mathrm{d}\theta \qquad \theta^* = \arg\max_\theta \Phi(J^*,\theta) \tag{7}$$

along with a conditional model $\mathrm{P}[\rho|t,d,J^*]$ that can infer programs for new tasks *based on the specification examples and the task descriptions*.

In the remainder of this section we first describe a joint model formulation that can be learned from the language-annotated training tasks, and then show how the joint model can be used to inform learning in the concrete base synthesis algorithm, DreamCoder, as an example of this framework.

5.1. Joint prior over programs and language

We formulate a learnable joint prior $J(\rho,d)$ as

$$J(\rho,d) = P[\rho|\mathcal{J},\theta]P[d|\rho,\mathcal{J},\theta] \tag{8}$$

which decomposes into the product of the original program prior defined on the library $P[\rho|\mathcal{J},\theta] = P[\rho|\mathcal{L},\theta_L]$ (where θ_L are the weights of the program prior) and a *program to descriptions* "translation" model $P[d|\rho,\mathcal{J},\theta] = T(d|J,\rho,\theta_T)$ (where θ_T are the translation model parameters) which describes how natural language descriptions are generated based on 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 on the underlying program library. Learning $T(d|\rho,\mathcal{J})$ formalizes the intuition that there should be a learnable correspondence between programs that solve tasks and the language that describes them.

$T(d|\rho,\mathcal{J})$ can be implemented in many ways (e.g. (Wong & Mooney, 2007; Joshi & Schabes, 1997; Bahdanau et al., 2014; Chen et al., 2018)) and is 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 $T(d|\rho,L)$ into

$$T(d|J,\rho,\theta_T) \propto \prod_{w_i\in d, l_j\in\rho} t(w_i|l_j) \tag{9}$$

a product of token-level translation probabilities $t(w_i|l_j)$ between program fragments l_i in a task's latent program ρ and words w_j in the task description d. (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.

In LAPS, the joint model also can be viewed as a controllable interface for incorporating additional prior knowledge about language in learned program synthesis. Learned translation models $T(d|\rho,\mathcal{J})$ are often parameterized to directly maximize the likelihood of the observed language (here, with respect to inferred latent training programs). However, our formulation also supports $T(d|\rho,\mathcal{J})$ enriched to model additional conditional priors over natural 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 (on unsolved tasks without known latent programs) should correspond to different program components than words already well-modeled in the existing joint model $J(d,\rho)$. In the distant supervision setting, this allows the model to incorporate a prior over words that do not appear in currently solved training tasks (for which there would be otherwise no learning signal to induce a translation model, without inferred latent programs for supervision). Our extended model incorporates this prior by updating Eq. 9 to distinguish between W_{known} (words that appear in solved training tasks with latent programs) and W_{new} (newly encountered words) as

$$T_{ME}(d|J,\rho,\theta_T) \propto \prod_{w_i\in d, l_j\in\rho} (\mathbb{1}[w_i\in W_{known}]t(w_i|l_j)) \\ (\mathbb{1}[w_i\in W_{new}]p(l|\mathcal{L},\theta_L)^{-1}) \tag{10}$$

where new words are modeled as *inversely* related to primitives under the program prior fit to previously encountered tasks – concretely, modeling the expectation that new words should be more likely to relate systematically 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 model $Q(\rho|t, d, J_i)$ that also conditions on language, to predict programs according to the posterior:

$$\begin{aligned} Q(\rho|t, d, J_i) &\approx \mathrm{P}[\rho|t, d, (J, \theta)] \\ &\propto \mathrm{P}[t|\rho]\mathrm{P}[\rho, d|(J, \theta)] \\ &\propto \mathrm{P}[t|\rho]\mathrm{P}[d|\rho, T, \theta_T]\mathrm{P}[\rho|(\mathcal{L}, \theta_L)] \end{aligned} \tag{11}$$

where (T, θ_T) is the translation model and $(\mathcal{L}, \theta_L)$ is the program prior in our joint model formulation. Importantly, we can train the language-conditioned model by using samples from the *joint* generative model, consisting of sampled programs *and corresponding generated language*. As with the original learning setting, the sample-based training allows LAPS to learn a generalizable neural search heuristic that conditions on language in training 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 in guiding learned conditional search: the neural model trained to amortize inference from the joint generative model can also be trained to approximate this mutual exclusivity bias, enabling better exploration and generalization in the presence of new words.

5.3. Abstraction learning as joint model compression

As in the base learning algorithm, 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 T in terms of the optimal program library.

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

$$\mathrm{P}[T|\mathcal{L}] \propto \sum_{l \in \mathcal{L}, w \in W} -H(t(w|l, \mathcal{L}, \theta_T)) \tag{12}$$

where $H(t(w|l, \mathcal{L}, \theta_T)) = -\log(t(w|l, \theta_T))$. This models the intuition that a good library contains program abstractions which correspond well to individual language tokens. This translation model prior also allows our algorithm to inherit the desirable property from the base algorithm in (Ellis et al., 2020): the extended compositional prior can still be efficiently re-approximated with respect to new candidate program abstractions based on their program subcomponents in the existing library, and the component translation distributions $t(w|l, \mathcal{L}, \theta_T)$ in the current translation model. As in the base synthesis algorithm, we finally re-estimate the translation model $T'(d|\mathcal{L}', \rho', \theta'_T)$ for the next iteration of training to refit the task annotations and programs refactored under the updated library.

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 heldout 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* provided by Mechanical Turk workers. We use synthetic language as our primary evaluation benchmark: we are interested in probing a controllable learning setting where words are systematically reused and composed, but refer to concepts at a much higher level of abstraction than the base programming language with which the system is initialized. However, we also use human language to evaluate the practicality of the approach in real-world settings. *Additional information for all domains is in the supplement.*

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

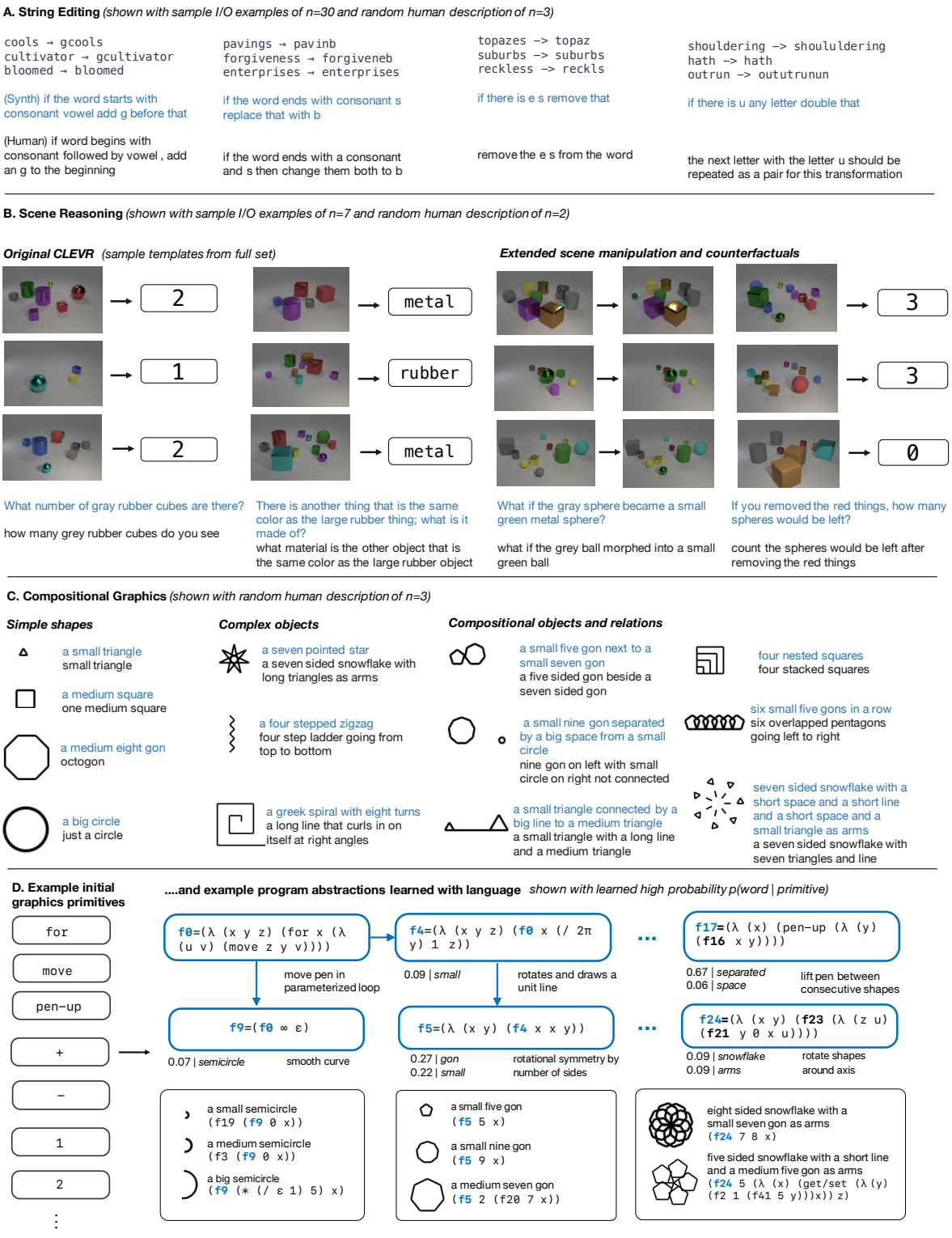

Figure 2. **(A, B, C)** Example tasks from all three synthesis domains shown with synthetic and sample human language annotations. Inductive synthesis domains are shown with a random subset (n=3) of the paired input/output examples. Human language annotations are also randomly sampled (all domains were 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.

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., 2020)). 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 synthesis problem is specified by an image and solved by synthesizing a program in LOGO Turtle graphics (Abelson & DiSessa, 1986). This domain is inspired by the graphics domain in (Ellis et al., 2020) but intentionally re-designed to be much 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., 2020). The neural search 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 things?*), 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 a dictionary of attributes) from the original CLEVR dataset generator in (Johnson et al., 2017a). We also add additional tasks that require *generating* or *imagining* new latent scenes (*how many metal things would be left if all the blue cylinders were removed?*), which are not solvable in the initial high-level DSL hand-designed in (Johnson et al., 2017b) (and used in synthesis-based CLEVR approaches like (Yi et al., 2018)). We include these to demonstrate a key feature of our approach: the ability to *learn* generalizable libraries by initializing with 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 generated from the ground-truth templates in (Johnson et al., 2017a) (and templates written in the same style for the extended tasks); human annotations are sourced from Mechanical Turk workers shown the same tasks. We initialize synthesizers with similar functional programming primitives to the string-editing domain and 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 translation model**) that trains a neural model directly on solved training tasks (similar to neural synthesis models like DeepCoder model (Devlin et al., 2017) but augmented to condition on language); and ablated variants of our own model (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 and released code contains full training details to replicate all experiments; and additional qualitative results.

Here we discuss evidence that our approach is effective specifically because of how language drives learning in the hierarchical joint model formulation:

(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., 2020), 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 have the prior knowledge for a good 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 description token length (which would be available in the absence of ground truth programs). Table 1 shows that our model is still much 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

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.

Language	Model	Strings (n_{test} = 500)	Graphics (n_{test} = 111)		Scenes (n_{test} = 115)	
		% Solved	% Solved (Best)	% Solved (Mean)	% Solved (Curric.)	% Solved (Mean.)
Synth train/test	DreamCoder (no language)	33.4	49.55	42. 64	67.80	73.9
Synth train/test	Multimodal (no generative translation model)	46.00	26.12	23.20	76.50	49.5
Synth train/test	LAPS in neural search	52.20	92.79	52.93	95.6	88.1
Synth train/test	LAPS + mutual exclusivity	**57.00**	86.49	80.18	**96.5**	82.3
Synth train/test	LAPS + ME + language-program compression	54.60	**98.19**	**81.98**	95.6	**95.9**
Synth train/human test	LAPS + ME + language-program compression	54.60	89.20	–	97.4	–
Human train/human test	LAPS + ME + language-program compression	48.60	58.55	–	95.6	–
No language at test: learned library comparisons						
No language on train/test	Original DSL; Enumerative	0.06	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	–

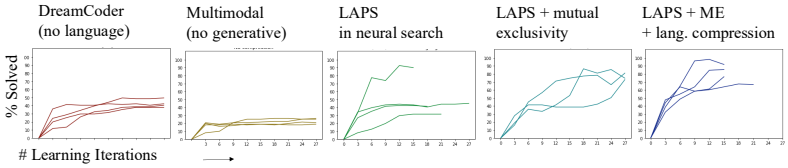

Figure 3. Learning curves comparing baselines and LAPS models in Table 1, showing % heldout tasks solved on the graphics domain over random training task orderings. (*Mean* results in Table 1 shows average test-time performance from the trained model replications.)

highlights a challenge for abstraction learning: not all shared 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 much more effective library overall, which generalizes to the non-language setting. See supplement for example learned abstractions from $\mathcal{L}_f$.

(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*) (similar to the findings in (Marzoev et al., 2020)), suggesting that when even finding human training annotations is too costly, in many cases hand-writing natural language templates to accompany a few ground-truth programs is likely sufficient (and easier than designing a full domain-specific programming language).

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 learned, *jointly generative* models over programs and language into program library and synthesizer learning. Going forwards, an important avenue for scalability will require exploring different concrete implementations of the base algorithm and learned model which relates programs to language. A promising future direction can leverage recent structured, neurally parameterized joint models that can *learn* the compositional units of language and more complex joint distributions, and incorporate pre-trained natural language representations (Joshi & Schabes, 1997; Lee et al., 2016; Wiseman et al., 2018; Kim et al., 2019).

The hierarchical Bayesian framing also draws connections to recent computational cognitive models which model *human conceptual representations and learning* (Goodman et al., 2014; Rule, 2020) as inference over program-like representations; posit program-like semantics for natural language (Portner & Partee, 2008; Fodor, 1975; Margolis et al., 1999); and provide evidence for the role language plays in non-linguistic cognition (Pyers et al., 2010; Spelke, 2017; Lupyan & Zettersten, 2020). Future *human* experiments could explore LAPS as a cognitive model, by combining experimental paradigms for studying language learning with those for studying non-linguistic abstraction and search (e.g. (Smith et al., 2003; Hawkins et al., 2019; Amato & MacDonald, 2010; Lake et al., 2015; 2019; Tian et al., 2020).

References

Abelson, H. and DiSessa, A. A. *Turtle geometry: The computer as a medium for exploring mathematics*. MIT press, 1986.

Abolafia, D. A., Norouzi, M., Shen, J., Zhao, R., and Le, Q. V. Neural program synthesis with priority queue training. *arXiv preprint arXiv:1801.03526*, 2018.

Alur, R., Singh, R., Fisman, D., and Solar-Lezama, A. Search-based program synthesis. *Communications of the ACM*, 61(12):84–93, 2018.

Amato, M. S. and MacDonald, M. C. Sentence processing in an artificial language: Learning and using combinatorial constraints. *Cognition*, 116(1):143–148, 2010.

Andreas, J., Klein, D., and Levine, S. Learning with latent language. *arXiv preprint arXiv:1711.00482*, 2017.

Appel, A. W., Beringer, L., Chlipala, A., Pierce, B. C., Shao, Z., Weirich, S., and Zdancewic, S. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20160331, 2017.

Artzi, Y. and Zettlemoyer, L. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62, 2013.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Brown, P. F., Della Pietra, S. A., Della Pietra, V. J., and Mercer, R. L. The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics*, 19(2):263–311, 1993.

Chen, X., Liu, C., and Song, D. Tree-to-tree neural networks for program translation. *arXiv preprint arXiv:1802.03691*, 2018.

Cropper, A. and Muggleton, S. H. Learning efficient logical robot strategies involving composable objects. AAAI Press/International Joint Conferences on Artificial Intelligence, 2015.

Dechter, E., Malmaud, J., Adams, R. P., and Tenenbaum, J. B. Bootstrap learning via modular concept discovery. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

Delaware, B., Pit-Claudel, C., Gross, J., and Chlipala, A. Fiat: Deductive synthesis of abstract data types in a proof assistant. *Acm Sigplan Notices*, 50(1):689–700, 2015.

Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., and Roy, S. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pp. 345–356, 2016.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 990–998. JMLR. org, 2017.

Du, T., Inala, J. P., Pu, Y., Spielberg, A., Schulz, A., Rus, D., Solar-Lezama, A., and Matusik, W. Inversecsg: Automatic conversion of 3d models to csg trees. *ACM Transactions on Graphics (TOG)*, 37(6):1–16, 2018.

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

Ellis, K., Ritchie, D., Solar-Lezama, A., and Tenenbaum, J. B. Learning to infer graphics programs from hand-drawn images. *arXiv preprint arXiv:1707.09627*, 2017.

Ellis, K., Morales, L., Sablé-Meyer, M., Solar-Lezama, A., and Tenenbaum, J. Learning libraries of subroutines for neurally–guided bayesian program induction. In *Advances in Neural Information Processing Systems*, pp. 7805–7815, 2018.

Ellis, K., Nye, M., Pu, Y., Sosa, F., Tenenbaum, J., and Solar-Lezama, A. Write, execute, assess: Program synthesis with a repl. *arXiv preprint arXiv:1906.04604*, 2019.

Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Cary, L., Morales, L., Hewitt, L., Solar-Lezama, A., and Tenenbaum, J. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *ArXiv preprint*, 2020.

Fikes, R. E. and Nilsson, N. J. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

Fodor, J. A. *The language of thought*, volume 5. Harvard university press, 1975.

Frank, M. C., Goodman, N. D., and Tenenbaum, J. B. Using speakers' referential intentions to model early cross-situational word learning. *Psychological science*, 20(5): 578–585, 2009.

Frome, A., Corrado, G. S., Shlens, J., Bengio, S., Dean, J., Ranzato, M., and Mikolov, T. Devise: A deep visual-semantic embedding model. In *Advances in neural information processing systems*, pp. 2121–2129, 2013.

Gal, Y. and Blunsom, P. A systematic bayesian treatment of the ibm alignment models. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 969–977, 2013.

Gandhi, K. and Lake, B. M. Mutual exclusivity as a challenge for deep neural networks. *arXiv preprint arXiv:1906.10197*, 2019.

Ganin, Y., Kulkarni, T., Babuschkin, I., Eslami, S. A., and Vinyals, O. Synthesizing programs for images using reinforced adversarial learning. In *International Conference on Machine Learning*, pp. 1666–1675. PMLR, 2018.

Gilpin, L. H., Bau, D., Yuan, B. Z., Bajwa, A., Specter, M., and Kagal, L. Explaining explanations: An overview of interpretability of machine learning. In *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*, pp. 80–89. IEEE, 2018.

Goodman, N. D. and Frank, M. C. Pragmatic language interpretation as probabilistic inference. *Trends in cognitive sciences*, 20(11):818–829, 2016.

Goodman, N. D., Tenenbaum, J. B., and Gerstenberg, T. Concepts in a probabilistic language of thought. Technical report, Center for Brains, Minds and Machines (CBMM), 2014.

Goyal, P., Niekum, S., and Mooney, R. J. Pixl2r: Guiding reinforcement learning using natural language by mapping pixels to rewards. *arXiv preprint arXiv:2007.15543*, 2020.

Grice, P. *Studies in the Way of Words*. Harvard University Press, 1989.

Gulwani, S., Hernández-Orallo, J., Kitzelmann, E., Muggleton, S. H., Schmid, U., and Zorn, B. Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99, 2015.

Gulwani, S., Polozov, O., Singh, R., et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4 (1-2):1–119, 2017.

Hawkins, R. X., Goodman, N. D., and Goldstone, R. L. The emergence of social norms and conventions. *Trends in cognitive sciences*, 23(2):158–169, 2019.

Jia, R. and Liang, P. Data recombination for neural semantic parsing. *arXiv preprint arXiv:1606.03622*, 2016.

Johnson, J., Hariharan, B., Van Der Maaten, L., Fei-Fei, L., Lawrence Zitnick, C., and Girshick, R. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2901–2910, 2017a.

Johnson, J., Hariharan, B., Van Der Maaten, L., Hoffman, J., Fei-Fei, L., Lawrence Zitnick, C., and Girshick, R. Inferring and executing programs for visual reasoning. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2989–2998, 2017b.

Johnson, M. Pcfg models of linguistic tree representations. *Computational Linguistics*, 24(4):613–632, 1998.

Joshi, A. K. and Schabes, Y. Tree-adjoining grammars. In *Handbook of formal languages*, pp. 69–123. Springer, 1997.

Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P., and Gulwani, S. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186*, 2018.

Kim, Y., Dyer, C., and Rush, A. M. Compound probabilistic context-free grammars for grammar induction. *arXiv preprint arXiv:1906.10225*, 2019.

Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Lake, B. M., Ullman, T. D., Tenenbaum, J. B., and Gershman, S. J. Building machines that learn and think like people. *Behavioral and brain sciences*, 40, 2017.

Lake, B. M., Linzen, T., and Baroni, M. Human few-shot learning of compositional instructions. *arXiv preprint arXiv:1901.04587*, 2019.

Lau, T. A. and Weld, D. S. Programming by demonstration: An inductive learning formulation. In *Proceedings of the 4th international conference on Intelligent user interfaces*, pp. 145–152, 1998.

Lázaro-Gredilla, M., Lin, D., Guntupalli, J. S., and George, D. Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs. *Science Robotics*, 4(26), 2019.

Lee, K., Lewis, M., and Zettlemoyer, L. Global neural ccg parsing with optimality guarantees. *arXiv preprint arXiv:1607.01432*, 2016.

Liang, P. Learning executable semantic parsers for natural language understanding. *Communications of the ACM*, 59(9):68–76, 2016.

Liang, P., Jordan, M. I., and Klein, D. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 639–646, 2010.

Liang, W., Zou, J., and Yu, Z. Alice: Active learning with contrastive natural language explanations. *arXiv preprint arXiv:2009.10259*, 2020.

Lin, D., Dechter, E., Ellis, K., Tenenbaum, J. B., and Muggleton, S. H. Bias reformulation for one-shot function induction. 2014.

Luketina, J., Nardelli, N., Farquhar, G., Foerster, J., Andreas, J., Grefenstette, E., Whiteson, S., and Rocktäschel, T. A survey of reinforcement learning informed by natural language. *arXiv preprint arXiv:1906.03926*, 2019.

Lupyan, G. and Zettersten, M. Does vocabulary help structure the mind? 2020.

Mao, J., Gan, C., Kohli, P., Tenenbaum, J. B., and Wu, J. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. *arXiv preprint arXiv:1904.12584*, 2019.

Margolis, E., Laurence, S., et al. *Concepts: core readings*. Mit Press, 1999.

Markman, E. M. and Wachtel, G. F. Children's use of mutual exclusivity to constrain the meanings of words. *Cognitive psychology*, 20(2):121–157, 1988.

Marzoev, A., Madden, S., Kaashoek, M. F., Cafarella, M., and Andreas, J. Unnatural language processing: Bridging the gap between synthetic and natural language data. *arXiv preprint arXiv:2004.13645*, 2020.

Mu, J., Liang, P., and Goodman, N. Shaping visual representations with language for few-shot classification. *arXiv preprint arXiv:1911.02683*, 2019.

Nye, M., Hewitt, L., Tenenbaum, J., and Solar-Lezama, A. Learning to infer program sketches. *arXiv preprint arXiv:1902.06349*, 2019.

Parisotto, E., Mohamed, A.-r., Singh, R., Li, L., Zhou, D., and Kohli, P. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

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

Polozov, O. and Gulwani, S. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 107–126, 2015.

Portner, P. H. and Partee, B. H. *Formal semantics: The essential readings*, volume 7. John Wiley & Sons, 2008.

Pyers, J. E., Shusterman, A., Senghas, A., Spelke, E. S., and Emmorey, K. Evidence from an emerging sign language reveals that language supports spatial cognition. *Proceedings of the National Academy of Sciences*, 107 (27):12116–12120, 2010.

Rule, J. S. *The child as hacker: building more human-like models of learning*. PhD thesis, Massachusetts Institute of Technology, 2020.

Shin, E. C., Allamanis, M., Brockschmidt, M., and Polozov, A. Program synthesis and semantic parsing with learned code idioms. In *Advances in Neural Information Processing Systems*, pp. 10824–10834, 2019.

Shin, R., Polosukhin, I., and Song, D. Improving neural program synthesis with inferred execution traces. In *NeurIPS*, pp. 8931–8940, 2018.

Si, X., Yang, Y., Dai, H., Naik, M., and Song, L. Learning a meta-solver for syntax-guided program synthesis. In *International Conference on Learning Representations*, 2019.

Silver, T., Allen, K. R., Lew, A. K., Kaelbling, L. P., and Tenenbaum, J. Few-shot bayesian imitation learning with logical program policies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 10251–10258, 2020.

Smith, K., Brighton, H., and Kirby, S. Complex systems in language evolution: the cultural emergence of compositional structure. *Advances in Complex Systems*, 6(04): 537–558, 2003.

Spelke, E. S. Core knowledge, language, and number. *Language Learning and Development*, 13(2):147–170, 2017.

Srivastava, S., Labutov, I., and Mitchell, T. Joint concept learning and semantic parsing from natural language explanations. In *Proceedings of the 2017 conference on empirical methods in natural language processing*, pp. 1527–1536, 2017.

Tian, L. Y., Ellis, K., Kryven, M., and Tenenbaum, J. B. Learning abstract structure for drawing by efficient motor program induction. *arXiv preprint arXiv:2008.03519*, 2020.

Wiseman, S., Shieber, S. M., and Rush, A. M. Learning neural templates for text generation. *arXiv preprint arXiv:1808.10122*, 2018.

Wong, Y. W. and Mooney, R. Learning synchronous grammars for semantic parsing with lambda calculus. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pp. 960–967, 2007.

Ye, X., Chen, Q., Dillig, I., and Durrett, G. Benchmarking multimodal regex synthesis with complex structures. *arXiv preprint arXiv:2005.00663*, 2020a.

Ye, X., Chen, Q., Dillig, I., and Durrett, G. Optimal neural program synthesis from multimodal specifications. *arXiv preprint arXiv:2010.01678*, 2020b.

Yi, K., Wu, J., Gan, C., Torralba, A., Kohli, P., and Tenenbaum, J. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. In *Advances in Neural Information Processing Systems*, pp. 1031–1042, 2018.

Zhang, Y., Pasupat, P., and Liang, P. Macro grammars and holistic triggering for efficient semantic parsing. *arXiv preprint arXiv:1707.07806*, 2017.

Supplemental: Leveraging Language for Abstraction and Program Search

This contains the supplemental appendix to the 2021 submission. It is organized sequentially in reference to the main text; S{N} refers back to section N in the main text.
A complete release of code for our implementation, including command line scripts to replicate the experiments in the paper and links to the datasets, can be found at: `https://bit.ly/2LMTu1G`. (This is an anonymized GitHub repo, and will be replaced with a deanonymized repository after the review period.)

S4. Base learning algorithm: DreamCoder

The LAPS framework described in the main paper is a general one (Sec. 4) for extending Bayesian models of program learning to incorporate information from natural language (see (Liang et al., 2010; Lake et al., 2015; Dechter et al., 2013; Lake et al., 2013)). Our concrete implementation and experiments use the DreamCoder approach of (Ellis et al., 2020; 2018) as the base synthesis algorithm, which implements the hierarchical Bayesian formulation of program learning. It defines a modular interface with two primary learning components: a learned *conditional inference* model for search (as a neural search heuristic); and a learned *abstraction* algorithm for updating the program prior (based on program refactoring and compression) (Ellis et al., 2020). Each of these learning components has been additionally implemented in other work (such as (Devlin et al., 2017; Polosukhin & Skidanov, 2018; Nye et al., 2019; Parisotto et al., 2016; Balog et al., 2016) for neurally guided synthesis, and (Dechter et al., 2013; Zhang et al., 2017; Shin et al., 2019; Artzi et al., 2014; Dumancić & Cropper) for program abstraction learning).

This supplementary section provides theoretical and implementation details on the DreamCoder algorithm we use in our experiments. We match our implementation as closely as possible to the original work for comparison with published baselines. We provide key details relevant to the language-guided extension, but strongly recommend the original works which introduce the DreamCoder algorithm (Ellis et al., 2020; 2018) for further reference.

S4.1 Program prior and MDL equivalence

Hierarchical Bayesian program learning formulations require a prior over expressible programs. DreamCoder is learned iteratively: it is initialized with a base library $\mathcal{L}_0$ and returns a library $\mathcal{L}_f$ containing program abstractions learned from solving training tasks. Therefore, DreamCoder defines its program prior with respect to the current library $\mathcal{L}_i$ maintained at each iteartion. This is parameterized as a simple PCFG $\mathrm{P}[\rho|\mathcal{L},\theta]$ whose productions are of the form $l_i \to l_j \in \mathcal{L}$, each with a real-valued weight θ_l, where the probability of a program ρ is given by $P[\rho|\mathcal{L},\theta] = \prod_{l\in\rho} P[l|\mathcal{L},\theta]$ (Sec. 4.1).

Minor complexity arises in order to support typing (Pierce, 2002): following (Ellis et al., 2018), the library $\mathcal{L}_i$ is implemented as a set of polymorphically typed λ-calculus expressions. The only change this produces to the original prior definition is to restrict the set of possible productions under the PCFG: that is, permissible productions are of the form $l_i \to l_j \in \{\mathcal{L}|l_i \to l_j \text{is well typed}\}$. The prior probabilities of programs are therefore calculated with respect to the set of well-typed productions.

As discussed in the main paper, this prior definition is *equivalent to a minimum description-length prior over programs* under $(\mathcal{L},\theta)$ when all $\theta < 1.0$, as the product of additional productions in an expression will strictly decrease as the number of productions in an expression increases.

S4.2 Amortized conditional inference

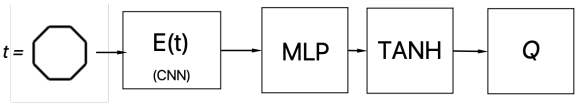

Figure 1. Architecture of the neural model $Q_i(\rho|t,\mathcal{L}_i)$. The model takes as input task examples t. These are encoded using a domain-specific encoder $E(t)$. Task encodings feed to an MLP and activation layer and output a tensor Q. This parameterizes a distribution over program bigrams in the final DSL, which defines a conditional distribution from which to enumerate programs during search.

To identify programs that solve tasks t while obtaining high probability under $P[\rho|\mathcal{L},\theta]$, DreamCoder trains a neural search heuristic $Q_i(\rho|t,\mathcal{L}_i)$ at each iteration i to approximate the inverse model.

The training procedure in (Ellis et al., 2020) (summarized in Sec. 4.2) is a key contribution of the original work for learning in the distant supervision setting. The model is trained on samples from the generative prior (providing an endless training stream of random synthesis tasks); and this proce-

dure should generalize immediately to any neural model for predicting programs conditioned on the task specification (e.g. (Devlin et al., 2017; Polosukhin & Skidanov, 2018; Nye et al., 2019; Parisotto et al., 2016; Balog et al., 2016)). The model is also supervised on any original training task examples and their program solutions discovered during learning.

In our experiments we use the baseline neural model architecture in (Ellis et al., 2020). This is parameterized by two modular components:

1. *A domain-specific task encoder* $E(t)$. This encodes the task examples (e.g. *images* in the graphics program domain, or input-output strings in the text editing domain) that are input to the neural model. This task encoder architecture is defined domain-specifically based on the form of the task examples (e.g. a CNN for the graphics domain). It outputs a fixed dimensional embedding for any given task as *input* to the model. In our experiments this is a 64-dimensional embedding across all domains (See S6.1 for domain-specific architectures; and released code.)

2. *A conditional model over programs* $Q(\rho|E(t))$. This component receives the task encoding as input and outputs a distribution over programs. Following (Ellis et al., 2020), this is a 2-layer fully-connected MLP (with 64 hidden units and a final tanh activation layer) that outputs a fixed-dimensional real-valued tensor encoding a distribution over programs in the library $\mathcal{L}$ as output. The real-valued tensor corresponds to weights over program primitives conditioned on their local context in the syntax tree of the program, consisting of the parent node in the syntax tree and which argument is being generated. This functions as a 'bigram transition model' over trees that encodes the likelihood of transitions from one primitive to the next. Q returns this as a $(|\mathcal{L}|+1)\times(|\mathcal{L}|+2)\times A$-dimensional tensor, where A is the maximum arity of any primitive in the library.

This parameterization supports fast sampling of programs during conditional synthesis: the neural model runs once per task (to encode the task examples and produce the bigram transition model) and the resulting parameterization can then be used to sample programs during synthesis (e.g. by enumerating programs by expanding trees (as 'bigrams' over parent and children primitives) ranked in order of their likelihood starting from the program root.)

Following (Ellis et al., 2020), the neural model is trained to optimize the following MAP inference objective on the training tasks and the sampled tasks from the prior:

$$\mathcal{L}_{\text{MAP}=\mathrm{E}_{x\sim(\mathcal{L},\theta)}}\left[\log Q\left(\arg\max_{\rho}\mathrm{P}[\rho|x,\mathcal{L},\theta]\;\middle|\;x\right)\right] \quad (1)$$

S4.3 Abstraction learning as program compression

DreamCoder learns new abstractions to approximately optimize for Eq. 2 (main paper), which infers an optimal library and parameters with respect to the observed programs on the training tasks.

The DreamCoder abstraction algorithm is a primary contribution of the original work in (Ellis et al., 2020), and is discussed extensively in (Ellis et al., 2020). We therefore provide additional technical details here that are relevant to its integration with LAPS in our experiments, but strongly encourage referencing (Ellis et al., 2020) for the full implementation.

As discussed in (Ellis et al., 2020) and our main work, DreamCoder approaches abstraction using an equivalence between Eq. 2 and the *minimum description length* of the *prior* (as the description length of the library) and the *programs* produced from the prior (under the PCFG definition of the prior). Therefore, in practice, inferring the optimal library is equivalent to inferring the library which maximally compresses the description length of the library and the description length of programs which explain the training tasks. In particular, DreamCoder optimizes the following compression objective with respect to the training tasks T and the finite *beam* B_t of program solutions discovered for each training task during learning:

$$\log \mathrm{P}[\mathcal{L}] + \arg\max_{\theta}\sum_{t\in T}\log\sum_{\rho\in\mathcal{B}_t}\mathrm{P}[t|\rho]\max_{\rho'\rho}\mathrm{P}[\rho'|\mathcal{L},\theta] + \log\mathrm{P}[\theta|\mathcal{L}] - |\theta|_0 \quad (2)$$

The key aspect of this algorithm is that it considers abstractions which compress not only the programs as they are *currently written*, but any semantically equivalent *refactorings* of these programs. Specifically, as programs are written in a λ-calculus, *refactoring* refers to any program which is equivalent up to β-reduction (i.e., function application/variable substitution (Pierce, 2002)). A primary contribution of the original work in (Ellis et al., 2020) is an efficient algorithm for computing these refactorings that is unchanged when we integrate language; we refer to the original text for details.

In our work, the primary important aspect of this aspect is that refactorings are defined compositionally over the existing program primitives. Specifically, refactorings can be efficiently calculated according to semantic equivalences in the the λ-calculus (namely, that function application and variable substitution guarantee that the resulting refactored programs are equivalent. *Abstractions* created by variable substitution will always be composed of subcomponents from the initial library.) We take advantage of this compositionality when defining our joint abstraction algorithm over natural language. Defining an initial *compositional* translation model between language and the program components

ensures that we can approximate compression in the joint model after the programs are refactored, without needing to induce an entirely new translation model over language and the refactored programs.

S5. Our Approach: Language for Abstraction and Program Search

This section now describes technical details for the concrete LAPS implementation in our reported experiments, which is defined over the DreamCoder implementation. We structure this section according to the parallel implementations in the base algorithm for clarity. However, except for the specifics of the joint-abstraction algorithm, the technical implementation of each component should extend directly to most other similar learned synthesis algorithms (e.g. the joint model implementation should be reusable in *any* synthesis algorithm that uses an explicit symbolic library of primitives.)

S5.1 Joint prior over programs and language

LAPS extends the prior P[ρ] over programs under the library to a *joint* prior $J(\rho, d)$ over programs for a given task and their natural language descriptions d (Sec. 5.1). We formulate this prior as

$$J(\rho, d) = P[\rho|\mathcal{L}, \theta]T(d|\rho, \mathcal{L})$$

the product of the original prior over programs $P[\rho|\mathcal{L}, \theta]$ defined on the program library, and a *program to descriptions* "translation" model $T(d|\rho, \mathcal{L})$ that describes how descriptions are generated for programs written in the library.

The concrete implementation described in the main paper uses a translation model that additionally decomposes compositionally over language and programs–in particular, on the basis of token-token translation distributions $t(d|l)$ between words $d \in D$ and $l \in \mathcal{L}$. Many available translation and semantic parsing models (such as synchronous grammars over natural language and programs) preserve this further compositional requirement (e.g. (Artzi et al., 2014; Wong & Mooney, 2006)).

See Figure 3 for example samples from the generative model on the graphics domain at earlier and later stages of training.

Our implementation uses a classical statistical machine translation model (the Model 4 version of the IBM Statistical Machine Translation models (Gal & Blunsom, 2013)) whose parameters can be tractably estimated from very few paired programs and descriptions (in the distant supervision setting used in the original work, there may be no more than a couple of hundred training tasks in the full dataset, and fewer than 10 solved tasks on which to train the translation model at any given time.) In addition to inference in small data settings, this translation model has a fully compositional generative definition (Gal & Blunsom, 2013) that allows it to be easily used to train the neural amortized inference model which conditions on language.

Despite this, however, this translation model (and the further inductive biases used to specifically relate program trees to sentences) make strong compositonality assumptions about the relationship between program primitives and words as a joint generative model of programs and language; we find that these inductive biases are useful in the small data setting and produce empirically successful results. However, this is likely because of *how* the joint model is used during training, which does not require a perfect generative model of language (or language with respect to programs) for either amortizing inference or abstraction in order to use language as a heuristic during learning.

A full definition of the statistical translation model we use can be found in (Gal & Blunsom, 2013). We re-summarize important details here. The IBM family of translation models estimates the conditional token-token probabilities $t(d|l)$ on the basis of *alignment* variables $a_{l,d}$, which specify a direct correspondence between tokens in parallel texts (e.g. a word in a task description and a program primitive.) These alignments are *many:many* between tokens in programs and natural language sentences – a given word can correspond to multiple primitives, and vice versa. Conditioned on a set of *alignments* from paired programs and descriptions, the conditional probabilities in *both* directions (the probability of generating a program primitive in a program based on the presence of a word in a sentence, and vice versa) are defined by marginalizing over the alignment variables. We provide one direction ($p(d|l)$), as the other is symmetrical:

$$p(d|l) \propto \sum_{a_1} ... \sum_{a_m} p(d, a_1 ... a_m|l) \propto \prod_{i=1}^{m} q(a_i|i, l, m)$$

where a_i are alignment variables inferred over a paired corpus and $q(j|i, l, m)$ can be interpreted as the probability of alignment variable a_i (for the token with index i in a program) taking value j (where j is an index into the corresponding sentence) conditioned on the lengths l and m of the program and natural language sentence (Gal & Blunsom, 2013).

These alignments are inferred by approximately inverting the generative model in (Gal & Blunsom, 2013) to maximize the likelihood of the observed paired sentences and programs. One implementation detail: the alignment algorithm operates over pairs of strings. For convenience we infer alignments between sentences and linearized token sequences in the program tree (which can be done with complete recoverability of the original program tree (Andreas et al., 2013)). This is another inductive assumption that we choose after preliminary experimentation and find that our

implementation yields strong empirical results regardless.

The IBM translation model is a noisy-channel generative model that requires an additional language model $p(d)$ to generate language (Gal & Blunsom, 2013; Heafield, 2011). We use an efficient parallelized implementation for inferring the translation model parameters from (Koehn et al., 2007), which also contains a basic language model inference algorithm inferred over the full corpus of training task sentences (as a trigram model, which we again find simple but effective for our very small data setting). Specific model hyperparameters for all experiments are available in the released code repo (in the experiment runtime commands.)

Mutual exclusivity: Section 5.1 of the main paper also describes how the joint model can be modified to include language-specific priors, such as a simple implementation of the well-known **mutual exclusivity** prior documented in the cognitive language-learning literature (Markman & Wachtel, 1988; Gandhi & Lake, 2019) and given a Bayesian formulation in (Frank et al., 2009). We provide an implementation to demonstrate that the joint model can be easily extended: specifically, a simple mutual exclusivity assumption can be added into the joint model by simply updating the compositional translation model to include additional distributions $t_{ME}(d_{new}|l)$ where d_{new} are words that *only* appear in unsolved training tasks and

$$t_{ME}(d_{new}|l) \propto \alpha P(l|L, \theta_L)^{-1}$$

new words are now assumed to correspond to primitives *inversely* proportional to their current usage under the learned program prior. As we show in the next section, incorporating this prior at the level of the joint model can be used to approximate mutual exclusivity assumptions in the learned search heuristic, encouraging exploration in the presence of new words.

Practically, we calculate the mutual exclusivity prior in our concrete implementation by leveraging the *alignments* upon which our token-token translation probabilities are defined. Specifically, we add *pseudoalignments* between each d_{new} and each $l \propto \alpha P(l|L, \theta_L)^{-1}$; when the token-token translation probabilities marginalize over the latent alignments and these pseudo alignments, the resulting translation probabilities encode the mutual exclusivity prior.

S5.2 Integrating the joint model into amortized conditional search

The amortized conditional inference model $Q(\rho|t)$ (Sec. 4.2) extends straightforwardly in LAPS to condition on language $Q(\rho|d, t)$ (Sec. 5.2). Importantly, the training procedure in Sec. 4.2 (training the neural model on samples from the prior) also extends to the language-enriched condition (training the neural model on samples from the joint prior, which include generated language annotations.)

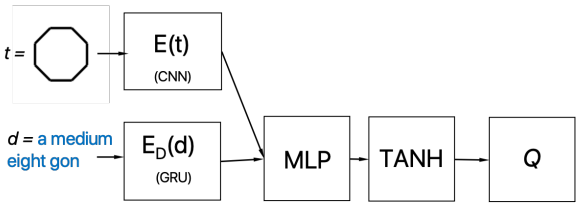

Figure 2. Architecture of the language-conditioned neural model $Q(\rho|d, t)$. The model takes as input task examples t. These are encoded using a domain-specific encoder $E(t)$. The model additionally takes in task descriptions d, encoded using a languag encoder $E_D(t)$ (implemented as a GRU). Task encodings are concatendated and feed to an MLP and activation layer and output a tensor Q. This parameterizes a distribution over program bigrams in the final DSL, which defines a conditional distribution from which to enumerate programs during search.

In our experiments we implement the concrete neural model $Q(\rho|d, t)$ in our experiments by extending modularly on the original model in (Ellis et al., 2020) (and in the supplemental S4.2) for direct comparison. Our full architecture therefore has *three* modular components to additionally condition on language:

1. A *natural language task descriptions encoder* $E_D(d)$. This receives the task description d as input. We implement this as an RNN model using a bidirectional GRU (Cho et al., 2014) with 64 hidden units; we embed natural language symbols as 64-dimensional vectors, and randomly initialize and backpropagate through the embedding during training. We tokenize the sentences in u on whitespace and concatenate each sentence, delimited by special start and end of sentence tokens. At test time, we replace any OOV tokens with a special UNK token.
2. A domain-specific task encoder $E(t)$, following S4.2.
3. A bigram transition model over program primitives, following S4.2. To condition jointly on $E_D(d)$ and $E(t)$ we simply concatenate these two embeddings and update the first layer of the MLP to take the 128-dimensional concatenated embeddings as input.

5.3 Abstraction learning as joint model compression

Finally, the *abstraction learning* model in (Ellis et al., 2020) can also be generalized to condition on language, by extending the optimal library inference algorithm with respect to the program prior to an optimal library inference algorithm with respect to the joint model over language and programs (Eq. 6 and 7, main text.)

In our concrete implementation with respect to the DreamCoder algorithm, this means extending the description-

length compression objective – originally defined over the program library and training task programs – to include the translation model definition. The main paper defines a description-length prior over the compositional translation model (Eq. 10). Optimizing this tractably requires redefining the abstraction algorithm in (Ellis et al., 2020) – which refactors λ-calculus programs via $lambda$-abstraction (see S4.3 for a summary) – to also jointly re-estimate the description length of the translation model $|T(D|\mathcal{L}')|$ using the refactored programs under the new candidate library $\mathcal{L}'$.

We implement an efficient approximation that can be calculated with respect to the classical statistical translation model described in S4.1 (Gal & Blunsom, 2013). In particular, we leverage the *alignment*-based definition (which uses latent correspondences inferred between program tokens and sentence tokens in paired programs and descriptions) to approximate $-log(t(d|l))$, the entropy of the token-token translation probabilities.

Specifically, as the IBM model defines the conditional token-token probabilities

$$t(d|l) \propto \sum_{a_1} ... \sum_{a_m} p(d, a_1...a_m|l)$$

marginalized over alignments, where (slightly abusing notation) in any given paired program and sentence description we will have estimated a set of alignments $a_{d_j,l_k...l_n}$ between the j-th token in the program corresponding to one *or more* tokens $l_k...l_n$ in the paired program. We therefore define the *description*-length of each token-token translation as the sum of the description lengths of the alignments which express it under a library $\mathcal{L}$:

$$\sum_{a_i} ... \sum_{a_m} p(d, a_1...a_m|l, \mathcal{L}) \propto \sum_{a_1} ... \sum_{a_m} |a_i|_{\mathcal{L}}$$

and the description lengths under the *refactored* library $\mathcal{L}'$ containing new abstractions compresses according to

$$|a'_{d_j,l'_k...l'_n}|_{\mathcal{L}'} < |a'_{d_j,l_k...l_n}|_{\mathcal{L}} \iff \{l'_i \text{contains only } l_k...l_n \text{ as subcomponents} | l'_k...l'_n\} \quad (3)$$

and we say that a primitive $l \in \mathcal{L}$ is a *subcomponent* of a refactored abstraction $l \in \mathcal{L}$ if the abstraction can be β-reduced such that l appears in it. That is, a refactored alignment $a' : w_i \to \{l'...l_n\}$ is compressed only when a new abstraction l' encapsulates over a strict subset of the constituent program primitives already aligned to the word in the original alignment. This allows us to re-approximate the description length of the new translation model with respect to a semantically-equivalent program refactoring without inducing $t(d|l)$ from scratch (which would require retraining the full translation model over the sentences and refactored programs.)

S6. Experiments

This section describes additional details on each of the domains – *string editing, compositional graphics,* and *scene understanding* – in Section 6 of the main paper (see **Figure 3, main text** for examples from all three domains, shown along with the synthetic and human language annotations). We also provide additional details on the model and baseline hyperparameters available for each domain. All datasets generated for these experiments (including human language annotations) are released and links to static repositories are provided in the code release. We also release a complete set of commands to exactly replicate all model experiments.

All experiments for were conducted on a high-powered computing cluster using a fixed training budget of wall-clock search time per task for all models and baselines in a given domain (determined via hyperparameter search using the baseline model per domain, and reported on a per-domain basis below). The experiments on the string editing and graphics domains used models trained using 48 CPUs for search (using the original parallel enumerative search implemented in the released code for the DreamCoder model in (Ellis et al., 2020)); and the experiments trained on the scene reasoning task used 24 CPUs (as preliminary experiments revealed that these experiments required shorter search time for our main model, and we wished to reduce the carbon footprint of the remaining experiments after our first two domains.)

For all experiments we train the neural models for 1×10^4 gradient steps. For experiments with language-guided compression, we use an upper bound of 5 new abstractions introduced per iteration. For mutual exclusivity experiments, we set $\alpha_{ME} = 0.1$. For all experiments, during program-only compression (see (Ellis et al., 2020) for a discussion of program-only compression hyperparameters) we use the hyperparameters from (Ellis et al., 2020) for parsimony with earlier work: a structure penalty of 1.5 and pseudocounts = 30.

S6.1 Domains

(See **Figure 2**, main text for examples from all three domains, shown along with the synthetic and human language annotations.) As discussed in the main paper, each domain consists of a dataset of *tasks*; a set of procedurally generated *synthetic language annotations*; and a set of *human language annotations* provided by Mechanical Turk workers; we also described the *base primitives* $\mathcal{L}_0$ with which all models (including baselines and ablations) were initialized for each domain.

S6.1.1 String Editing

Tasks: structured string transformation problems taken from a publicly released dataset in (Andreas et al., 2017) (n=1000 train; n=500 test). Tasks consist of input dictionary strings transformed using randomly sampled regular expression transducer (n=30 examples per task). Transducers were sampled according to abstract templates defined in (Andreas et al., 2017) and required identifying matched sequences of characters and *adding* letters before them; *removing* sequences; *replacing* them with new sequences, or *doubling* the sequence each time they appeared (See **Figure 2A, main text**).

Language data: The human language dataset for this domain was previously collected by (Andreas et al., 2017). We defined a synthetic grammar of high-level templates over the ground truth regular expression transducers (corresponding to the original templates used to generate the tasks.) The synthetic templates were defined based on language from the original human annotations, and in most cases closely matched the true human provided annotations (which were generally quite structured), though with significantly less variation (the original language contained multiple human descriptions per task. We generate a single synthetic for each one. The synthetic dataset has a vocabulary size of n=44 for both train and test. We use the human annotations in the original dataset when evaluating on human data, which have a vocabulary of n=727 (train) and n=622 (test).) We generate a synthetic dataset on this domain partly because of inaccuracies noted in (Andreas et al., 2017). The released code contains the complete generation procedure for these synthetic annotations. See Figure 2A for representative tasks with examples, synthetic language, and human descriptions.

Initial program primitives: We initialize all models with a set $\mathcal{L}_0$ of LISP-like primitives that operate over substring sequences to both construct regular expression match sequences and manipulate strings, augmented with three text manipulation-specific primitives intended for executing constructed regular expression sequences; `t` is a polymorphic type variable using standard Hindley-Milner polymorphism typing (Pierce, 2002). The execution engine does include a regex-matching model; however, the synthesis model is naive to this execution engine and simply searches for manipulations over the input strings and the regexes as data arrays.

$\mathcal{L}_0$ contains 14 substring manipulation primitives, given below with type information. We also give a semantic gloss for primitives that are not standard LISP primitives.

- `if (bool → t → t → t)`
- `cons (t → list(t) → list(t))`
- `car (list(t) → t)`
- `cdr list(t) → list(t`
- `map` $((t_0 \rightarrow t_1) \rightarrow list(t_0) \rightarrow list(t_1))$
- `tail (list(t) → t)`
- `append (t → list(t) → list(t))`
  Appends element to end of list.
- `revcdr (list(t) → list(t))`
  Takes all except the last element of the list.
- `match (substr → substr → bool)`
  Returns true if the first argument, when executed as a regular expression, matches the second argument.
- `regexsplit (substr → fullstr → list(substr))`
  Attempts to execute the first argument as a regular expression, and splits the second argument into a list of substrings, using the regular expression match as a delimiter (and includes the matched sequences in the returned list.)
- `flatten (list(substr) → fullstr)`
  Flattens a list of substrings back into a string.
- `rconcat (substr → substr → substr)`
  Concatenates two substrings.
- `rnot (substr → substr)`
  Takes a substring argument `s` and returns the substring literal [^ s]
- `ror (substr → substr → substr)`
  Takes substring literals a and b and returns the substring literal ((a)—(b))

We also include 26 character constants of type `substr` and constants `dot` (regular expression wildcard character) and `empty` (empty string).

Domain hyperparameters We largely follow prior work (Ellis et al., 2020) to set algorithm training parameters; the earlier (Ellis et al., 2020) uses a 720s enumerative search budget for solving both text editing and general list manipulation tasks. We use the same 720s enumerative budget here.

S6.1.2 Compositional Graphics

Tasks: inverse graphics problems (n=200 train; n=111 test) where each synthesis problem is specified by an image and solved by synthesizing a program in LOGO Turtle graphics (Abelson & DiSessa, 1986). The domain is inspired by the graphics domain in (Ellis et al., 2020) but intentionally

re-designed to be much more challenging (ground-truth programs are much longer on average in the base programming language) and explicitly compositional: the training and testing tasks contain *simple shape tasks* defined by compositional parameters for a set of basic shapes (*a small triangle, a medium square; a small semicircle*); *complex shape tasks* that require inferring more challenging (and longer) parameterized shapes (*a greek spiral with eight turns*); and *compositional tasks* defined by geometric rules and relations over the simple shapes (*a seven sided snowflake with a short line and a small triangle as arms; a small triangle connected by a big space from a small circle*) (See **Figure 2C**).

Simple parameterized shapes are either polygons (*triangle, square, [n] gon*), curves (*semicircle, circle*) or *line*s. Simple shapes are parameterized by one of three sizes (*small* or *short*; *medium*; and *big*). When generating synthetic language descriptions, pluralized objects are tokenized with separate tokens for the noun lemma and a token for the plural suffix (e.g. *square s*).

Complex parameterized shapes require constructing more complex images out of basic lines, and are intended to evaluate performance on tasks that pose a greater search challenge in the initial DSL, and whose structure is not directly cued by compositional relationships over easier components. Further, the complex shapes can be solved using abstractions (e.g. for repeatedly rotating a pen at right angles) that are not directly cued by shared lexical names – we evaluate the algorithm's ability to learn and use abstractions that correspond to useful sublexical structures shared across multiple lexemes. We define four template families for complex shapes: *spiral*s, *staircase*s, *zigzag*s, and *star*s.

Compositional graphics tasks invoke compositional relationships over the simple parameterized shapes. We define templates for generating 6 families of compositional tasks: *nested*, *next to*, *separated by*, *connected by*, *in a row*, and *snowflake*s.

Language data: We gather human language annotations by asking Mechanical Turk workers to write an image description for the rendered graphics images that specify each task. Each worker labeled 20 training and 10 testing images after viewing a disjoint, randomly sampled set of 15 example images paired with their synthetic language captions. (Workers were asked to write a *short, clear description that a person or robot could use to recreate the picture*, and told that the examples were paired with *automatically generated captions as an example of the kinds of descriptions you could write for this picture*.) We control for description quality by requiring workers to complete a reference task on their own descriptions: after writing their initial annotations, workers were required to correctly match each annotation to the target image (from amidst a set of 12 distractors drawn heuristically from similar images on the full task dataset, and other images they themselves had described), and only annotations correctly matched to the target image were retained (workers were given a chance to redescribe pictures they failed to match to their own captions.) We preprocess the human dataset minimally to standardize number terms (e.g. we use the same token type for both *3* and *three*) and to split plurals into a lemma and suffix, as in the synthetic dataset. The final dataset has a vocabulary size of n=562 for both train and test.

As with the string editing domain, we define a synthetic dataset using parameterized templates based on systematic language reused in the human annotations (see Figure 2A for a comparison between human annotations and synthetic language); as with that domain, we choose a synthetic dataset to ensure systematic re-use of high level terms for repeated compositional objects (such as the "n-gon" or "snowflake" terminology.)

We then generate graphics tasks by defining parameterized templates over ground truth programs *in* $\mathcal{L}_0$, and a corresponding generator for synthesizing natural language descriptions based on each ground truth program. It is important to note that the templates are defined at any extremely high level and were written with respect to low-level programs in a simple graphics language (many of which were derived by generalizing compositionally over complex structures in (Ellis et al., 2020), such as the 'snowflake' images).

Initial program primitives: For comparison with prior work, our initial library on this domain (and the base language used to generate the ground truth graphics programs) is an implementation of the LOGO Graphics DSL used in (**?**), which consists of four typed, imperative primitives modeled within the $\lambda-$calculus with a state monad S:

```
move:  distance → angle → S → S
pen-up:  (S → S) → S → S
for:  int → (S → S) → S → S
get/set:  (S → S) → S → S
```

as well as four arithmetic operators (+, -, *. /), integer constants (1-9), unit distances and angles (1 meter and 2π radians), and special values ∞ and ϵ.

Figure 3 (main text) shows examples of the graphics tasks, synthetic descriptions, human descriptions, and sample programs in the ground truth initial DSL.

Domain hyperparameters We largely follow prior work (Ellis et al., 2020) to set algorithm training parameters. Consistent with the graphics program experiments in (Ellis et al., 2020), we train all models, including baselines and ablations, using an enumerative search budget of 1800s per task (both when using pure enumerative search from the DSL prior, and neurally-guided search conditioned on the task examples and language descriptions); the results in Table

1 compare the relative advantage of our model given this fixed search time. We train all models on 48 CPUs during parallel enumerative search, and run the algorithm for a maximum of 27 iterations (see learning curves. As we run multiple random seed replications of models in this domain, we tuned the iteration limit based on performance on the first replication, allowing models models to train while performance continued to increase. To conserve computational resources, we later stopped several of our own model replications before 27 iterations, as they had reached near ceiling performance. As we report the best held-out test score across all 27 iterations for any one model, the early stopping would only serve to give a conservative estimate on performance for these models.) We randomly reorder the training set of tasks once before the first loop, then iterate through batches of n=40 tasks at each iteration; learning curves show results from evaluating on held-out tasks every n=3 iterations.

S6.1.3 Scene Reasoning

Tasks: 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 things?*), or another scene (*what if all of the red things turned blue?*). This domain is modeled on CLEVR (Johnson et al., 2017) but designed to support non-linguistic, inductive synthesis in the programming-by-example paradigm: each task is specified with *n=7* paired input output examples. See **Figure 3B, main text** for example tasks showcasing the original and extended templates, synthetic language annotations, and human language annotations.

The dataset includes questions randomly generated from the following subset of the *original CLEVR question templates* (see (Johnson et al., 2017) for additional details on the task generation process and question templates; we also release our own augmented question generation code and the full dataset):

- **zero_hop**: questions that require counting or answering an attribute query about a subset of objects in the scene. (e.g. *How many small cylinders are there?*; *What material is the purple thing?*).
- **one_hop**: questions similar to the *zero_hop* tasks, but that require reasoning over an additional relational query (e.g *What number of things are right the small gray thing?*).
- **single_or**: questions that additionally introduce a *disjunction* between sets of objects. (e.g. *How many objects are either large metal spheres or large rubber things?*)).
- **(compare_integer**: questions that additionally introduce a $\geq$ or $\leq$ operator between counts of sets of objects. (e.g. *Is the number of large rubber cubes less than the number of large green rubber things?*)
- **same_relate**: questions that additionally require reasoning about other objects with the same attribute as a specified object. (e.g. *How many other things are there of the same size as the cyan thing?*).

We choose these templates as a representative subset of the style of the full CLEVR dataset, that requires the full language of high-level primitives in (Johnson et al., 2017) to solve. We omit some longer questions in the same format (e.g. *two_hop*) as our intention is to compare synthesis baselines, rather than to achieve SOTA performance on CLEVR: this would likely only increase the computing resources needed to compare the various methods and we already found a significant differential between our model and the baselines on the shorter questions.)

We also add *new* question templates generated in the style of the original CLEVR tasks, but designed to model other common AI tasks (such as generating new scenes based on existing ones) and to require new abstractions (that were not expressible in the original restricted symbolic language used to generate scenes in (Johnson et al., 2017)):

- **localization**: questions for object localization. These return an output *scene* consisting of a localized set of objects based on a set of query attributes (e.g. *Find the gray rubber thing.*).
- **remove**: questions that either return an output *scene* with a subset of the objects removed, or that query about latent scenes where a subset of objects has bee removed. (e.g *What if you removed all of the gray metal things?*; *If you removed the green cubes, how many cubes would be left?*).
- **transform**: questions that either return an output *scene* where a subset of the objects has been *transformed* to set new attributes, or that query about latent scenes where a subset of objects has been modified this way. (e.g *What if all the blue metal things became rubber things?*; *If all of the large yellow rubber things became gray spheres, how many gray spheres would there be?*).

We treat these as program synthesis tasks: the input scenes are specified as *symbolic scene graphs consisting of an array of structured, objects defined as a dictionary of their attributes*, and programs are designed to manipulate these structured arrays (this data structure is the original format in which scenes themselves are generated in (Johnson et al.,

2017); the images displayed in Figure 3, main text are rendered using the original image rendering pipeline). Our intention is *not* to build a visual reasoning architecture: rather, we are interested in learning structured manipulations of scenes. We see work in *inverse graphics* (such as (Yi et al., 2018)) which outputs a structured scene graph based on pixel images as the *first* step in a symbolic processing and reasoning pipeline as analogous; we are interested in the structured manipulation of these scene representations.

Language data: Synthetic language annotations are generated based on the original high-level templates in (Johnson et al., 2017), as well as additional templates we define for the extended questions in the same style. We gather human language annotations by asking Mechanical Turk workers to write an instruction or question describing the set of inductive examples. However, due to the difficulty of solving certain tasks in a limited time frame based on the inductive examples alone (such as the questions about disjunctions over scenes), we show Mechanical Turk workers the synthetic descriptions for this domain and ask them to write a semantically similar description that changes more than one word in the original caption, and that would be "more natural for a human to understand". This paraphrasing paradigm is similar to that used in (Wang et al., 2015), though we find that in comparison to other domains it generates less diverse language data.) We remove all punctuation, tokenize on spaces, and use an additional domain heuristic to stem all plurals (e.g. *cubes*).

Initial program primitives: We initialize all models with a set $\mathcal{L}_0$ of LISP-like primitives. These are similar to the initial list manipulation primitives used in the *string editing* domain: as both domains can be treated as manipulating structured arrays, we are interested in learning differentiated, domain-specific abstractions based on a very similar base language. L_0 also includes primitives for querying attributes of objects on the domain (these are typed getters that simply query the object dictionary of attributes) and several domain-specific functions necessary for manipulating these attribute. We deliberately use a much more base level programming language than the high-level, domain-specific language hand-designed in (Johnson et al., 2017); our goal is to *learn* the necessary abstractions.

We give a semantic gloss for primitives that are not standard LISP primitives.

- `if (bool → t → t → t)`
- `cons (object → list(object) → list(object))`
- `car (list(object) → object)`
- `map` $((t_0 \rightarrow t_1) \rightarrow list(t_0) \rightarrow list(t_1))$
- `fold` $((list(t) \rightarrow list(t)) \rightarrow (t \rightarrow list(t) \rightarrow list(t)) \rightarrow list(t))$
- `len (list(t) → int)`
- `>` `(list(t) → bool)`
- `<` `(list(t) → bool)`
- `set_union (list(t) → list(t) → list(t))`
- `set_intersect (list(t) → list(t) → list(t))`
- `set_difference (list(t) → list(t) → list(t))`
- `relate (object → relation → list(t))` Returns an array of objects that satisfy a spatial relation with respect to an input object.

We also include *equality* comparators for each of the attribute types (e.g. `eq_color?`; *getters* for each attribute, and *setters* for each attribute. We also include integer constants 0-9 for counting and constants for the attributes (`blue, red, big, small, rubber, metal`) based on the original object and spatial relation constants (Johnson et al., 2017).

Domain hyperparameters: We run a coarse hyperparameter search based on the baseline model to set the domain hyperparameters. We train all models, including baselines and ablations, using an enumerative search budget of 1000s per task and run the models for a maximum of 5 iterations. we run multiple random seed replications reordering the training set, in the same way as the compositional graphics domain. The results in Table 1 also compare a *curriculum* ordering of the training set based on the number of tokens in the synthetic language captions (split on spaces.)

S 6.2 Results and Additional Qualitative Results

In this section, we discuss additional qualitative results from an in depth exploration of the graphics domain that were omitted from the main paper for space, but provide additional insight on the behavior of the learned model in the hardest learning domain (based on the differential between baseline and LAPS-augmented performance.)

Learned abstractions and synthesized programs. Figure 4 show sample abstractions in the final libraries $\mathcal{L}_f$ for the best performing models in the graphics domain as a concrete exemplar of abstractions that are learned and how they are used, along with sample tasks solved with these abstractions. The figures are shown as dependency graphs to indicate how progressively more complex abstractions *build* on abstractions at prior iterations of learning; we also show selected

probabilities from the translation model (depicted are examples from the top-3 primitive translations for a given word; some primitives are not high probability translations for any word.)

Joint generative model samples. Figure 3 shows samples from the joint generative model on the graphics domain (programs from the library which are executed to produce the task example image, and translated to produce language annotations) at early and later stages of training, indicating that the joint model itself improves as learning improves, which itself allows better training for the conditional inference model and better abstraction guiding based on language.

References

Abelson, H. and DiSessa, A. A. *Turtle geometry: The computer as a medium for exploring mathematics*. MIT press, 1986.

Andreas, J., Vlachos, A., and Clark, S. Semantic parsing as machine translation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 47–52, 2013.

Andreas, J., Klein, D., and Levine, S. Learning with latent language. *arXiv preprint arXiv:1711.00482*, 2017.

Artzi, Y., Das, D., and Petrov, S. Learning compact lexicons for ccg semantic parsing. 2014.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Dechter, E., Malmaud, J., Adams, R. P., and Tenenbaum, J. B. Bootstrap learning via modular concept discovery. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 990–998. JMLR. org, 2017.

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

Ellis, K., Morales, L., Sablé-Meyer, M., Solar-Lezama, A., and Tenenbaum, J. Learning libraries of subroutines for neurally–guided bayesian program induction. In *Advances in Neural Information Processing Systems*, pp. 7805–7815, 2018.

Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Cary, L., Morales, L., Hewitt, L., Solar-Lezama, A., and Tenenbaum, J. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *ArXiv preprint*, 2020.

Frank, M. C., Goodman, N. D., and Tenenbaum, J. B. Using speakers' referential intentions to model early cross-situational word learning. *Psychological science*, 20(5): 578–585, 2009.

Gal, Y. and Blunsom, P. A systematic bayesian treatment of the ibm alignment models. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 969–977, 2013.

Gandhi, K. and Lake, B. M. Mutual exclusivity as a challenge for deep neural networks. *arXiv preprint arXiv:1906.10197*, 2019.

Heafield, K. Kenlm: Faster and smaller language model queries. In *Proceedings of the sixth workshop on statistical machine translation*, pp. 187–197. Association for Computational Linguistics, 2011.

Johnson, J., Hariharan, B., Van Der Maaten, L., Hoffman, J., Fei-Fei, L., Lawrence Zitnick, C., and Girshick, R. Inferring and executing programs for visual reasoning. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2989–2998, 2017.

Koehn, P., Hoang, H., Birch, A., Callison-Burch, C., Federico, M., Bertoldi, N., Cowan, B., Shen, W., Moran, C., Zens, R., et al. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the association for computational linguistics companion volume proceedings of the demo and poster sessions*, pp. 177–180, 2007.

Lake, B. M., Salakhutdinov, R. R., and Tenenbaum, J. One-shot learning by inverting a compositional causal process. In *Advances in neural information processing systems*, pp. 2526–2534, 2013.

Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Liang, P., Jordan, M. I., and Klein, D. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 639–646, 2010.

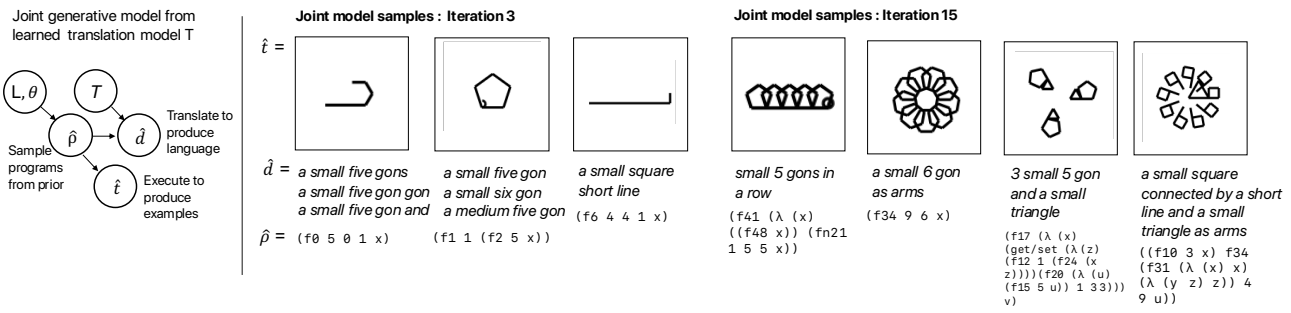

Figure 3. (left) Joint generative model J over programs sampled from the DSL prior and natural language produced by the translation model $T(D|\mathcal{L})$, inferred from solved training tasks. Samples from the model are used to train a neural synthesizer to guide search on more challenging, unsolved tasks. (right) Samples from the J generative model in the graphics domain shows how program complexity increases and generated language improves across iterations, as the system both adds richer abstractions to the DSL and learns better alignments over the solution set, enabling the trained neural model to solve more complex tasks

.

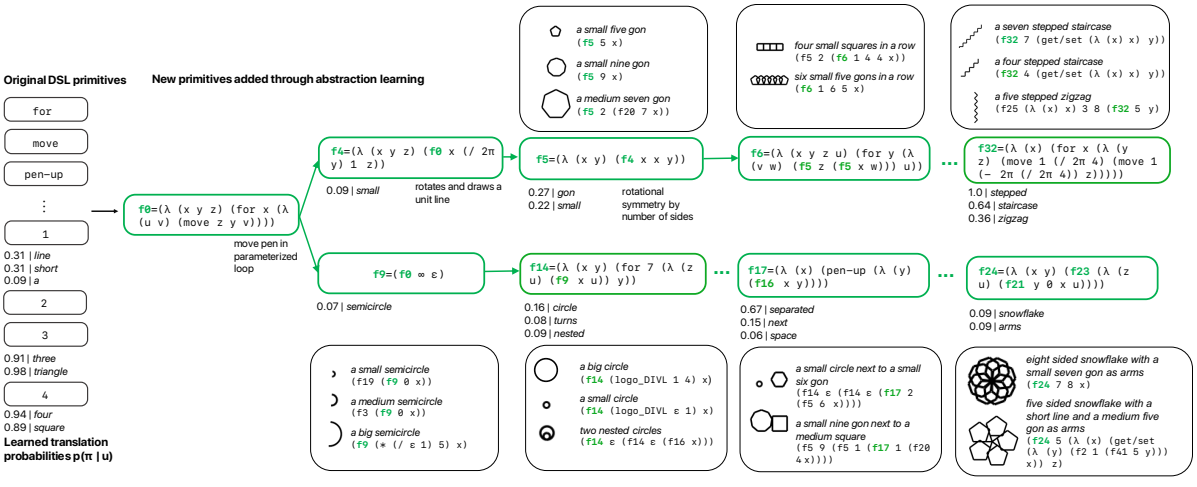

Figure 4. Abstractions and programs learned for the graphics domain. Sample abstractions (right) learned from a minimal starting DSL (left) for solving progressively more complex graphics program synthesis tasks with language annotations. Also shown with translation probabilities. Our iterative algorithm learns alignment-based translation probabilities between natural language words and program primitives to guide program search and abstraction (depicted are examples from the top-3 primitive translations for a given word; some primitives are not high probability translations for any word.

Markman, E. M. and Wachtel, G. F. Children's use of mutual exclusivity to constrain the meanings of words. *Cognitive psychology*, 20(2):121–157, 1988.

Nye, M., Hewitt, L., Tenenbaum, J., and Solar-Lezama, A. Learning to infer program sketches. *arXiv preprint arXiv:1902.06349*, 2019.

Parisotto, E., Mohamed, A.-r., Singh, R., Li, L., Zhou, D., and Kohli, P. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

Pierce, B. C. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

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

Shin, E. C., Allamanis, M., Brockschmidt, M., and Polozov, A. Program synthesis and semantic parsing with learned code idioms. In *Advances in Neural Information Processing Systems*, pp. 10824–10834, 2019.

Wang, Y., Berant, J., and Liang, P. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1332–1342, 2015.

Wong, Y. W. and Mooney, R. J. Learning for semantic parsing with statistical machine translation. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, pp. 439–446. Association for Computational Linguistics, 2006.

Yi, K., Wu, J., Gan, C., Torralba, A., Kohli, P., and Tenenbaum, J. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. In *Advances in Neural Information Processing Systems*, pp. 1031–1042, 2018.

Zhang, Y., Pasupat, P., and Liang, P. Macro grammars and holistic triggering for efficient semantic parsing. *arXiv preprint arXiv:1707.07806*, 2017.