

1 Introduction

An age-old dream within AI is a machine that learns and reasons by writing its own programs. This vision stretches back to the 1960's [35] and is central to much of what it would take to build machines that learn and think like humans. Computational models of cognition often explain the flexibility and richness of human thinking in terms of program learning: from everyday thinking and problem solving (motor program induction as an account of recognition and generation of handwriting and speech [20]; functional programs as a model of natural language semantics [?]) to learning problems that unfold over longer developmental time scales: the child's acquisition of intuitive theories (of kinship, taxonomy, etc.) [39] and natural language grammar [32], to name just a few. An outstanding challenge, however, is to engineer program-learners that display the same level of domain-generalizability as the humans they are meant to model.

Recent program-learning systems developed within the AI and machine learning community are impressive along many dimensions, authoring programs for problem domains like drawing pictures [?, 10], transforming text [13] and numerical sequences [2], and reasoning over common sense knowledge bases [26]. These systems work in different ways, but typically hinge upon a carefully hand-engineered Domain Specific Language (DSL). The DSL restricts the space of programs to contain the kinds of concepts needed for one specific domain. For example, a picture-drawing DSL could include concepts like circles and spirals, and a DSL for numerical sequences could include sorting and reversing lists of numbers. Modern systems also learn how to efficiently deploy the DSL on new problems [7, 2, 17], but – unlike human learners – do not discover the underlying system of concepts needed to navigate the domain.

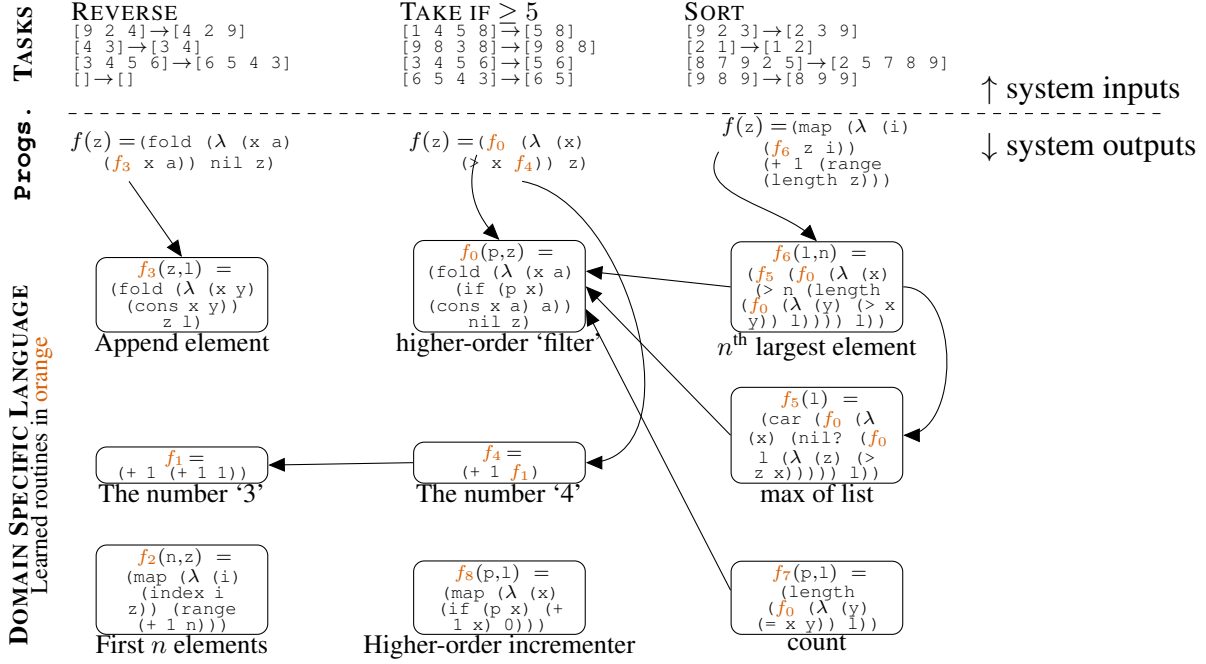
We contribute a program-induction system that learns the domain-specific concepts (DSL) while jointly learning how to use those concepts. This joint learning problem models two complementary notions of domain expertise: (1) domain experts have at their disposal a powerful, yet specialized repertoire of concepts and abstractions (analogous to the DSL) while also (2) having accurate intuitions about when and how to use those concepts to solve new problems. Representative domains, along with DSLs we learn for them, are shown in Figure 1.

We call our system 'DreamCoder' because it acquires these two kinds of expertise through a novel kind of wake/sleep or 'dream' learning [14], iterating through a wake cycle – where it solves problems by writing programs – and a pair of sleep cycles, both of which are loosely biologically inspired by actual human sleep. The first sleep cycle, which we refer to as **consolidation**, grows the DSL by replying experiences from waking and consolidating them into new code abstractions. This cycle is inspired by the formation of abstractions during sleep memory consolidation [8]. The second sleep cycle, which we refer to as **dreaming**, improves the agents knowledge of how to write programs by training a neural network to help search for programs. The neural net is trained on replayed experiences as well as 'fantasies', or samples, from the DSL. These two kinds of dreams are inspired by the distinct episodic replay and hallucination components of dream sleep [12].

This dream-learning architecture brings together two lines of prior work, both of which have been separately influential within artificial intelligence. One line of work considers the problem of learning new concepts, abstractions, or 'options' from experience [6, 22, 36, 16, 37], while the other line of work considers the problem of learning how to deploy those concepts efficiently [7, 2, 17]. Our goal with DreamCoder is to show that the combination of these ideas is uniquely powerful, and pushes us toward program-writing systems that, like human learners, autonomously acquire the expertise needed to navigate a new domain of problems.

Because any learning problem can in principle be cast as program induction, it is important to delimit our focus. In contrast to computer assisted programming [34] or genetic programming [18], our goal is not to automate software engineering, or to synthesize large bodies of code starting from scratch. Ours is a basic AI goal: capturing the human ability to learn to think flexibly and efficiently in new domains — to learn what you need to know about a domain so you don't have to solve new problems starting from scratch. We are focused on problems that people solve relatively quickly, once they acquire the relevant domain expertise. These correspond to tasks solved by short programs — if you have an expressive DSL.

DOMAIN: LIST PROCESSING



DOMAIN: GRAPHICS PROGRAMMING

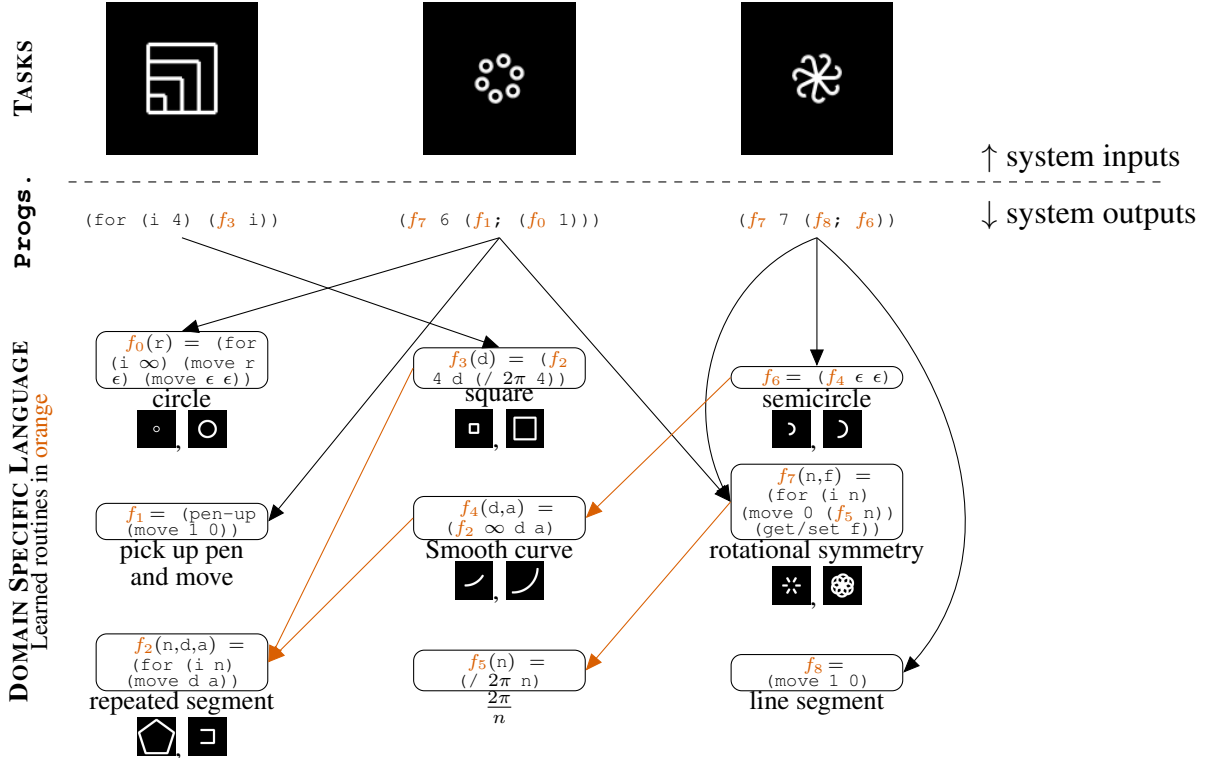


Figure 1: Two of the six domains we apply our system to. Agent observes tasks (top rows) which it solves by writing programs (middle rows) while jointly growing a library (DSL; bottom rows). Learned DSLs rediscover multiple higher-order functions (*filter* for list functions and rotational symmetry for generative graphics). Learned DSL components call each other (arrows).

2 Wake/Sleep Program Induction

DREAMCODER takes as its goal to acquire domain-specific expertise, which it learns by solving a collection of programming **tasks**. It alternately finds programs that solve tasks (Wake – Figure 2 top); improves its DSL by discovering and reusing domain-specific subroutines (Consolidation – Figure 2 left); and trains a neural network that efficiently guides search for programs in the DSL (Dreaming – Figure 2 right). The learned DSL acts as a prior on programs likely to solve tasks in the domain, while the neural net looks at a specific task and produces a “posterior” for programs likely to solve that specific task (Figure 2 middle). The neural network thus functions as a **recognition model** supporting a form of approximate Bayesian program induction, jointly trained with a **generative model** for programs encoded in the DSL, in the spirit of the Helmholtz machine [14]. The recognition model ensures that searching for programs remains tractable even as the DSL (and hence the search space for programs) expands. The generative model, or DSL, distills out common abstractions across programs discovered during waking, growing a network of increasingly deep and specialized domain-specific concepts (Figure 1, bottom rows).

Viewed as an inference problem, our goal is to start with a collection of tasks, written X , and infer both a program for each task, as well as a distribution over programs encoded by a DSL, written \mathcal{D} . We equip \mathcal{D} with a learned weight vector θ , and together (\mathcal{D}, θ) define a generative model over programs (see Appendix 3). We perform maximum a posteriori (MAP) inference of (\mathcal{D}, θ) given X . Writing J for the joint probability of (\mathcal{D}, θ) and X , we want the \mathcal{D}^* and θ^* solving:

$$J(\mathcal{D}, \theta) \triangleq \mathbb{P}[\mathcal{D}, \theta] \prod_{x \in X} \sum_p \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta]$$

$$\mathcal{D}^* = \arg \max_{\mathcal{D}} \int J(\mathcal{D}, \theta) d\theta \quad \theta^* = \arg \max_{\theta} J(\mathcal{D}^*, \theta) \quad (1)$$

where $\mathbb{P}[x|p]$ scores the likelihood of a task $x \in X$ given a program p .¹ The above equations summarize the problem from the point of view of an ideal Bayesian learner. However, Eq. 1 is wildly intractable because evaluating $J(\mathcal{D}, \theta)$ involves summing over the infinite set of all programs. In practice we will only ever be able to sum over a finite set of programs. So, for each task, we define a finite set of programs, called a *frontier*, and only marginalize over the frontiers:

Definition 1. A *frontier of task* x , written \mathcal{F}_x , is a finite set of programs s.t. $\mathbb{P}[x|p] > 0$ for all $p \in \mathcal{F}_x$.

Using the frontiers we define the following intuitive lower bound on the joint probability, called \mathcal{L} :

$$J \geq \mathcal{L} \triangleq \mathbb{P}[\mathcal{D}, \theta] \prod_{x \in X} \sum_{p \in \mathcal{F}_x} \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta] \quad (2)$$

Our wake and sleep cycles correspond to alternate maximization of \mathcal{L} w.r.t. $\{\mathcal{F}_x\}_{x \in X}$ (**Wake**) and (\mathcal{D}, θ) (**Consolidation**):

Wake: Maxing \mathcal{L} w.r.t. the frontiers. Here (\mathcal{D}, θ) is fixed and we want to find new programs to add to the frontiers so that \mathcal{L} increases the most. \mathcal{L} most increases by finding programs where $\mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta] \propto \mathbb{P}[p|x, \mathcal{D}, \theta]$ is large (i.e., programs with high posterior probability).

Sleep (Consolidation): Maxing \mathcal{L} w.r.t. the DSL. Here $\{\mathcal{F}_x\}_{x \in X}$ is held fixed, and so we can evaluate \mathcal{L} . Now the problem is that of searching the discrete space of DSLs and finding one maximizing $\int \mathcal{L} d\theta$, and then updating θ to $\arg \max_{\theta} \mathcal{L}(\mathcal{D}, \theta, \{\mathcal{F}_x\})$.

Searching for programs is hard because of the large combinatorial search space. We ease this difficulty by training a neural recognition model, $Q(\cdot|\cdot)$, during the **Dreaming** phase: Q is trained to approximate the posterior over programs, $Q(p|x) \approx \mathbb{P}[p|x, \mathcal{D}] \propto \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}]$. Thus training the neural network amortizes the cost of finding programs with high posterior probability.

Sleep (Dreaming): tractably maxing \mathcal{L} w.r.t. the frontiers. Here we train $Q(p|x)$ to assign high probability to programs p where $\mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta]$ is large, because incorporating those programs into the frontiers will most increase \mathcal{L} .

Intuitively, this 3-phase inference procedure can work in practice because each of the 3 phases bootstraps off of the others. As the DSL grows and as the recognition model becomes more accurate, waking becomes more effective, allowing the agent to solve more tasks; when we solve more tasks during waking, the Consolidation cycle has more

¹For example, for list processing, the likelihood is 1 if the program predicts the observed outputs on the observed inputs, and 0 otherwise; when learning a generative model or probabilistic program, the likelihood is the probability of the program sampling the observation.

data from which to learn the DSL; and, because the recognition model is trained on both samples from the DSL and programs found during waking, the recognition model gets both more data, and higher-quality data, whenever the DSL improves and whenever we discover more successful programs.

2.1 Wake: Solving tasks

During waking we enumerate programs from the DSL in decreasing order of their probability according to the recognition model, and then check if a program p assigns positive probability to a task ($\mathbb{P}[x|p] > 0$); if so, we incorporate p into the frontier \mathcal{F}_x . We represent programs as polymorphically-typed λ -calculus expressions, a representation closely resembling Lisp and functional languages like Haskell and OCaml, including variables, conditionals, higher-order recursive functions, and the ability to create new functions.

Why enumerate, when the program synthesis community has invented many sophisticated algorithms that search for programs? [34, 31, 11, 27, 29]. We have two reasons: (1) A key point of our work is that learning the DSL, along with a neural recognition model, can make program induction tractable, even if the search algorithm is very simple. (2) Enumeration is a general approach that can be applied to any program induction problem. Many of these more sophisticated approaches require special conditions on the space of programs.

2.2 Consolidation-Sleep: Growing a Domain Specific Language

The DSL offers a set of abstractions that allow an agent to concisely express solutions to the tasks at hand. We automatically discover these new abstractions by combining two ideas. First, we build on techniques from the programming languages community to develop a new algorithm for automatically refractoring programs, where this refractoring exposes common reused subexpressions found across the programs found during waking. Second, we use this automatic refractoring process to search for DSLs that maximally compress these programs by incorporating reused subexpressions into the DSL.

Mathematically this compression takes the form of finding the DSL maximizing $\int \mathcal{L} d\theta$ (Sec. 2). We replace this marginal with an AIC approximation and minimize the following expression, which can be interpreted as a kind of compression:

$$\underbrace{-\log \mathbb{P}[\mathcal{D}] + \min_{\theta} \left(-\log \mathbb{P}[\theta|\mathcal{D}] + \|\theta\|_0 \right)}_{\text{Description length of } (\mathcal{D}, \theta)} + \sum_{x \in X} \underbrace{-\log \sum_{p \in \mathcal{F}_x} \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta]}_{\text{Description length of programs for task } x} \quad (3)$$

At a high level, our approach is to search locally through the space of DSLs, proposing small changes to \mathcal{D} until Eq. 3 fails to decrease. These small changes consist of introducing new candidate λ -expressions into the DSL.

However, there is a snag with this simple approach: whenever we add a new expression e to the DSL, the programs found during waking ($\{\mathcal{F}_x\}$) are not written in terms of e . Concretely, imagine we wanted to discover a new DSL procedure for doubling numbers, after having found the programs $(\text{cons } (+ \ 9 \ 9) \ \text{nil})$ and $(\lambda \ (x) \ (+ \ (\text{car } x) \ (\text{car } x)))$. As human programmers, we can look at these pieces of code and recognize that, if we define a new procedure called `double`, defined as $(\lambda \ (x) \ (+ \ x \ x))$, then we can rewrite the original programs as $(\text{cons } (\text{double } 9) \ \text{nil})$ and $(\lambda \ (x) \ (\text{double } (\text{car } x)))$. This process is a kind of refractoring where a new subroutine is defined (`double`) and the old programs rewritten in terms of the new subroutine. Figure 3A diagrams this refractoring process for a more complicated setting, where the agent must rediscover the higher-order function `map` starting from the basics of Lisp and the Y-combinator.

We refine our objective to compress *refactorings* of the programs found during waking, minimizing

$$-\log \mathbb{P}[\mathcal{D}] + \min_{\theta} \left(-\log \mathbb{P}[\theta|\mathcal{D}] + \|\theta\|_0 + \sum_{x \in X} -\log \underbrace{\sum_{\substack{p: \\ \exists p' \in \mathcal{F}_x: p \longrightarrow^* p'}}_{\text{Refactors } \mathcal{F}_x}} \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta] \right) \quad (4)$$

where $p \longrightarrow^* p'$ is the standard notation for “expression p evaluates to p' by the rules of λ -calculus” [28]. Equation 4 captures the idea that we want to add new components to the DSL while jointly refractoring our old programs in terms of these new components. But this joint optimization is intractable, because there are infinitely many ways of refractoring

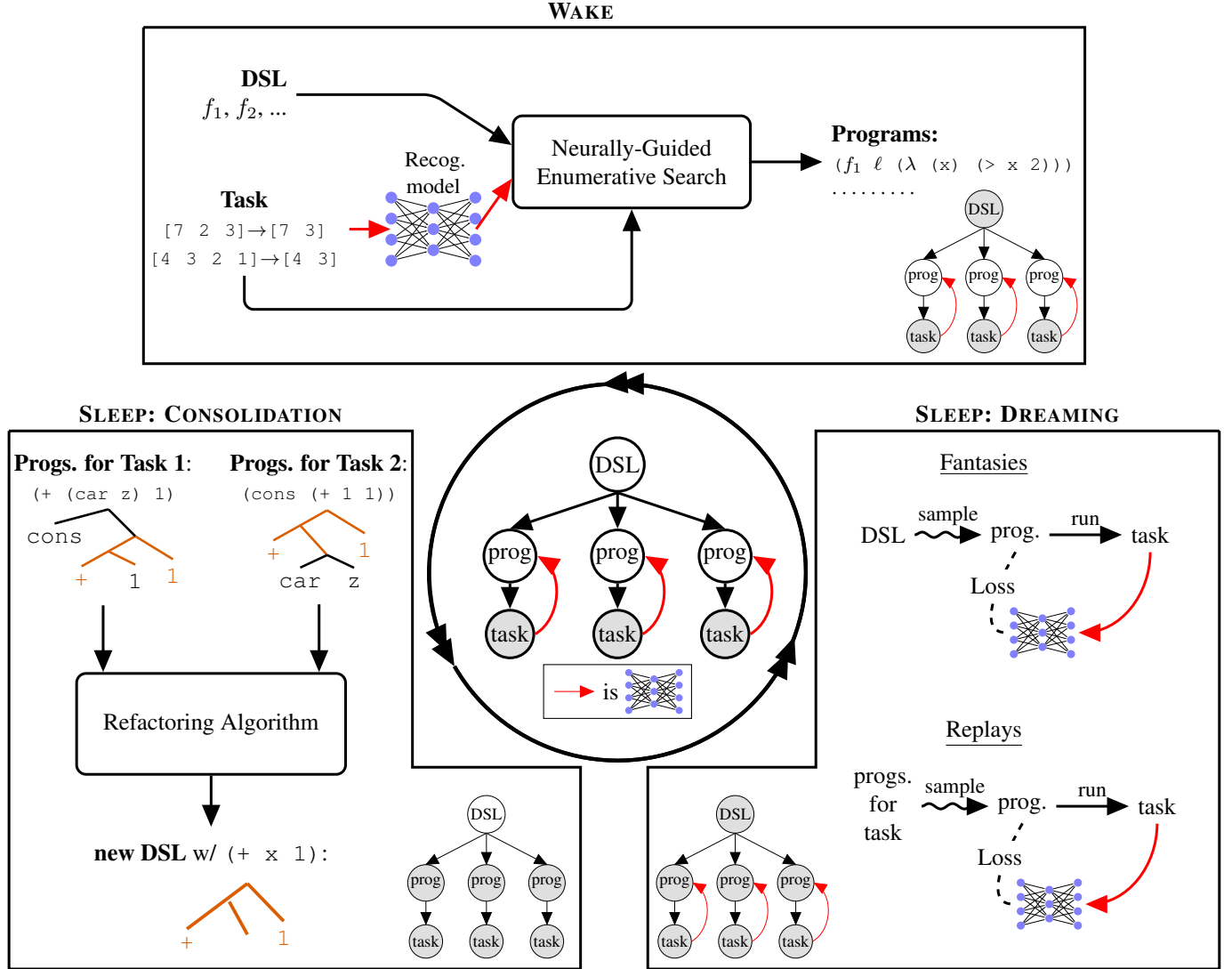


Figure 2: **Middle:** DREAMCODER as a graphical model. Agent observes programming tasks (e.g., input/outputs for list processing or images for graphics programs), which it explains with latent programs, while jointly inferring a latent Domain Specific Language (DSL) capturing cross-program regularities. A neural network, called the *recognition model* (red arrows) is trained to quickly infer programs with high posterior probability. **Top:** Wake phase infers programs while holding the DSL and recognition model fixed. **Left:** Sleep (Consolidation) phase infers DSL while holding the programs fixed by refactoring programs found during waking and extracting common components. **Right:** Sleep (Dreaming) phase trains recognition model to predict approximate posterior over programs conditioned on task. Trained on ‘Fantasies’ (programs sampled from DSL) & ‘Replays’ (programs found during waking).

a program. To make refactoring tractable we first limit the degree to which a piece of code can be refactored: rather than consider every refactoring, we bound the number of λ -calculus evaluation steps separating a refactoring from its original program. Now the number of refactorings this is finite but astronomically large: for the example in Figure 3 there are approximately 10^{14} possible refactorings – a quantity that grows exponentially both as a function of program size and a function of the bound on evaluation steps. How can we tame this combinatorial explosion?

A structure called a **version space algebra** [21, 25, 29] neatly resolves this exponential growth. A version space is a tree-shaped data structure that compactly represents a large set of programs and supports efficient set operations like union, intersection, and membership checking. In Appendix A.3, we give a dynamic program that takes as input a program p and then outputs a version space containing the refactorings of p . Our technique is astronomically more efficient than explicitly representing the space of possible refactorings: for the example in Figure 3A, we represent the space of refactorings using a version space with 10^6 nodes, which encodes 10^{14} refactorings.

With this machinery in hand, we now have all the pieces needed to learn a DSL (Algorithm 1). The functions $I\beta(\cdot)$ and REFACTOR construct a version space from a program and extract the shortest program from a version space, respectively (Algorithm 1, lines 5-6, 14; see Appendix A.3). To define the prior distribution over (\mathcal{D}, θ) (Algorithm 1, lines 7-8), we penalize the syntactic complexity of the λ -calculus expressions in the DSL, defining $\mathbb{P}[\mathcal{D}] \propto \exp(-\lambda \sum_{p \in \mathcal{D}} \text{size}(p))$ where $\text{size}(p)$ measures the size of the syntax tree of program p , and λ controls how strongly we regularize the size of the DSL. We place a symmetric Dirichlet prior over the weight vector θ .

To appropriately score each proposed \mathcal{D} we must reestimate the weight vector θ (Algorithm 1, line 7). Although this may seem very similar to estimating the parameters of a probabilistic context free grammar, for which we have effective approaches like the Inside/Outside algorithm [19], our DSLs are context-sensitive due to the presence of variables in the programs and also due to the polymorphic typing system. Appendix A.4 derives a tractable MAP estimator for θ .

Algorithm 1 DSL Induction Algorithm

```

1: Input: Set of frontiers  $\{\mathcal{F}_x\}$ 
2: Output: DSL  $\mathcal{D}$ , weight vector  $\theta$ 
3:  $\mathcal{D} \leftarrow$  every primitive in  $\{\mathcal{F}_x\}$ 
4: while true do
5:    $\forall p \in \bigcup_x \mathcal{F}_x : v_p \leftarrow I\beta(p)$  ▷ Construct a version space for each program
6:   Define  $L(\mathcal{D}', \theta) = \prod_x \sum_{p \in \mathcal{F}_x} \mathbb{P}[x|p] \mathbb{P}[\text{REFACTOR}(p|\mathcal{D}')|\mathcal{D}', \theta]$  ▷ Likelihood if  $(\mathcal{D}', \theta)$  were the DSL
7:   Define  $\theta^*(\mathcal{D}') = \arg \max_{\theta} \mathbb{P}[\theta|\mathcal{D}'] L(\mathcal{D}', \theta)$  ▷ MAP estimate of  $\theta$ 
8:   Define  $\text{score}(\mathcal{D}') = \log \mathbb{P}[\mathcal{D}'] + L(\mathcal{D}', \theta^*) - \|\theta\|_0$  ▷ objective function
9:   components  $\leftarrow \{\text{REFACTOR}(v|\mathcal{D}) : \forall x, \forall p \in \mathcal{F}_x, \forall v \in \text{children}(v_p)\}$  ▷ Propose many new DSL components
10:  proposals  $\leftarrow \{\mathcal{D} \cup \{c\} : \forall c \in \text{components}\}$  ▷ Propose many new DSLs
11:   $\mathcal{D}' \leftarrow \arg \max_{\mathcal{D}' \in \text{proposals}} \text{score}(\mathcal{D}')$  ▷ Get highest scoring new DSL
12:  if  $\text{score}(\mathcal{D}') < \text{score}(\mathcal{D})$  return  $\mathcal{D}, \theta^*(\mathcal{D})$  ▷ No changes to DSL led to a better score
13:   $\mathcal{D} \leftarrow \mathcal{D}'$  ▷ Found better DSL. Update DSL.
14:   $\forall x : \mathcal{F}_x \leftarrow \{\text{REFACTOR}(p|\mathcal{D}) : p \in \mathcal{F}_x\}$  ▷ Refactor frontiers in terms of new DSL
15: end while

```

2.3 Dream Sleep: Training a Neural Recognition Model

During “dreaming” the system learns a recognition model that guides program search. It learns from (program, task) pairs drawn from two sources of self-supervised data: *replays* of programs discovered during waking, and *fantasies*, or programs drawn from the DSL. Replays ensure that the recognition model is trained on the tasks solved so far, and does not forget how to solve them. Fantasies ensure that the recognition model has a large and highly varied corpus of (program, task) pairs to learn from.

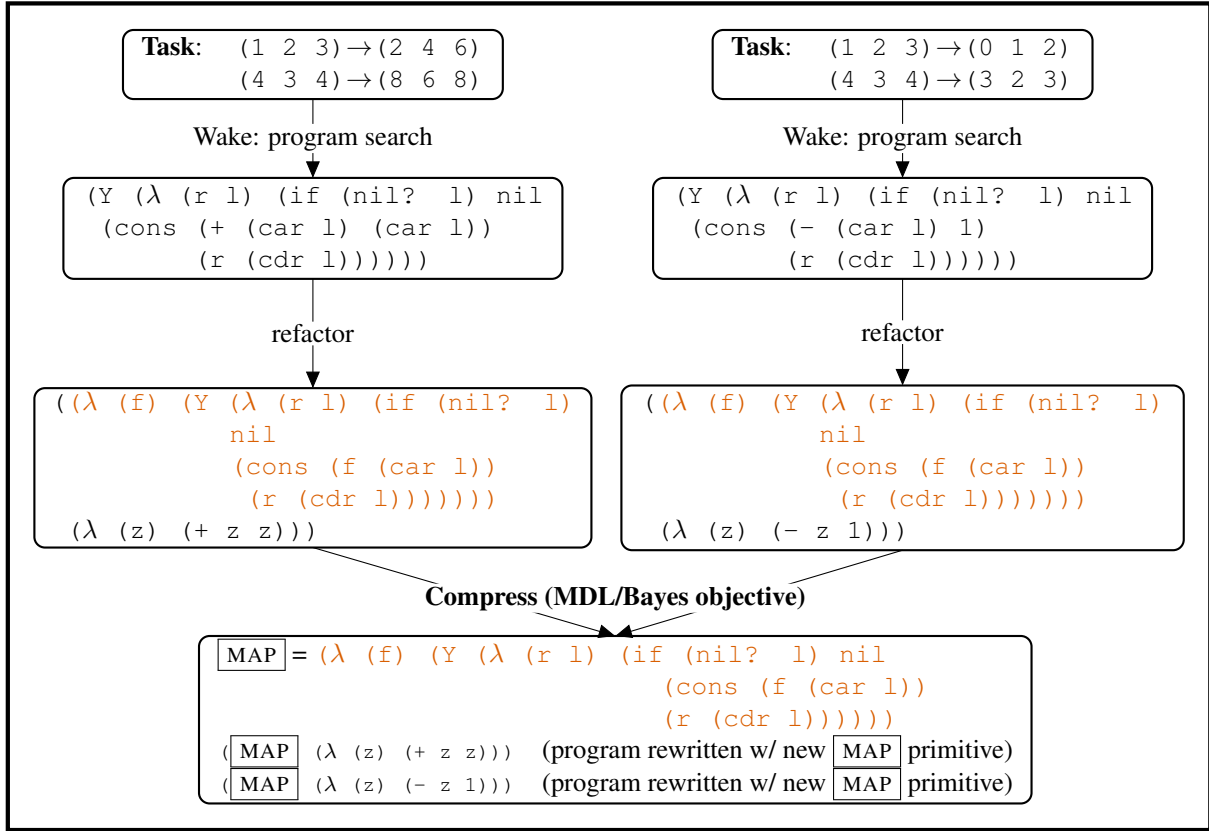
Formally, $Q(p|x)$ should approximate the posterior $\mathbb{P}[p|\mathcal{D}, \theta, x]$

training a neural recognition model, Q , to regress from a task to a distribution over programs that are likely to both solve the task and be likely under the prior. To train Q we need (program, task) pairs.

To get the data to train Q , we take inspiration from the

The purpose of training the recognition model is to amortize the cost of searching for programs. It does this by learning to predict, for each task, programs with high likelihood according to $\mathbb{P}[x|p]$ while also being probable under the prior (\mathcal{D}, θ) , thereby approximating the posterior $\mathbb{P}[\cdot|x, \mathcal{D}, \theta]$.

A



B

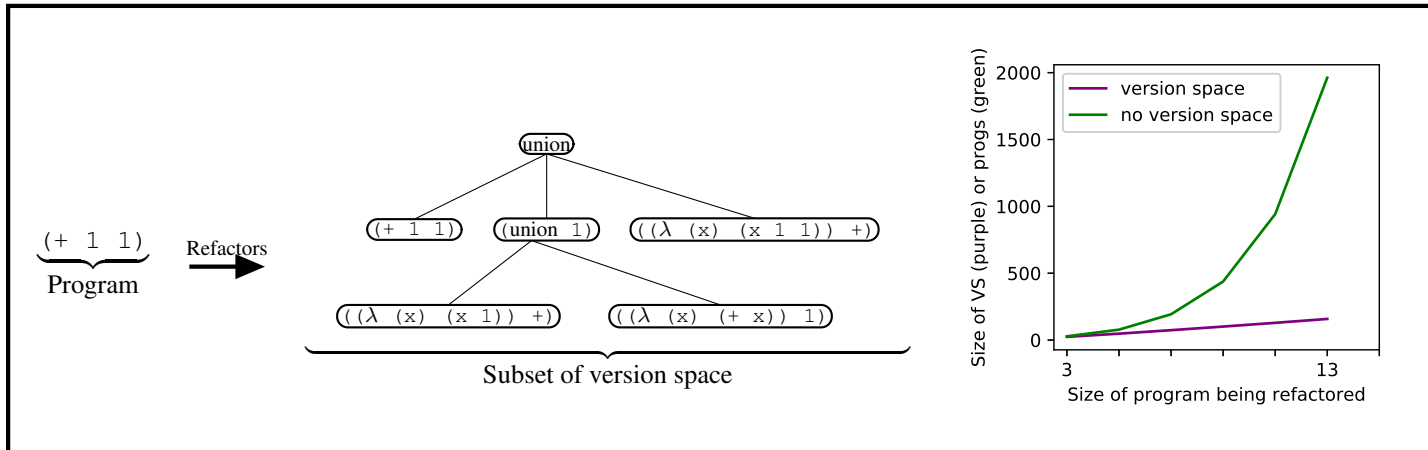


Figure 3: DSL learning as code refactoring. **Panel A:** For each task we discover programs during waking, then refactor the code from those programs to expose common subprograms (highlighted in orange). Common subprograms are incorporated into the DSL when they increase a Bayesian objective. Intuitively, these new DSL components best compress the programs found during waking. **Panel B:** # of possible refactorings grows exponentially with program size. We represent refactorings using version spaces, which augment syntax trees with a *union* operator whose children are themselves version spaces. Right graph: version spaces are exponentially more efficient than explicitly constructing set of refactorings.

We can either train Q to perform full posterior inference by minimizing the expected KL-divergence, $\mathbb{E} [\text{KL} (\mathbb{P}[p|x, \mathcal{D}, \theta] \| Q(p|x))]$, or we can train Q to perform MAP inference by maximizing $\mathbb{E} [\max_{p \text{ maxing } \mathbb{P}[\cdot|x, \mathcal{D}, \theta]} \log Q(p|x)]$, where in both cases the expectation is taken over tasks. Taking this expectation over the empirical distribution of tasks trains Q on the real data; taking it over samples from the generative model trains Q on “dreams.” We define a pair of alternative objectives for the recognition model, $\mathcal{L}_{\text{RM}}^{\text{posterior}}$ and $\mathcal{L}_{\text{RM}}^{\text{MAP}}$, which either train Q to perform full posterior inference or MAP inference, respectively. These objectives combine real-data and dream data:

$$\begin{aligned} \mathcal{L}_{\text{RM}}^{\text{posterior}} &= \mathcal{L}_{\text{Real}}^{\text{posterior}} + \mathcal{L}_{\text{Dream}}^{\text{posterior}} & \mathcal{L}_{\text{RM}}^{\text{MAP}} &= \mathcal{L}_{\text{Real}}^{\text{MAP}} + \mathcal{L}_{\text{Dream}}^{\text{MAP}} \\ \mathcal{L}_{\text{Real}}^{\text{posterior}} &= \mathbb{E}_{x \sim X} \left[\sum_{p \in \mathcal{F}_x} \frac{\mathbb{P}[x, p | \mathcal{D}, \theta] \log Q(p|x)}{\sum_{p' \in \mathcal{F}_x} \mathbb{P}[x, p' | \mathcal{D}, \theta]} \right] & \mathcal{L}_{\text{Real}}^{\text{MAP}} &= \mathbb{E}_{x \sim X} \left[\max_{p \in \mathcal{F}_x} \log Q(p|x) \right] \\ \mathcal{L}_{\text{Dream}}^{\text{posterior}} &= \mathbb{E}_{(p, x) \sim (\mathcal{D}, \theta)} [\log Q(p|x)] & \mathcal{L}_{\text{Dream}}^{\text{MAP}} &= \mathbb{E}_{x \sim (\mathcal{D}, \theta)} \left[\max_p \log Q(p) \right] \end{aligned}$$

The ‘dream’ objectives are essential for data efficiency: all of our experiments train DREAMCODER on only a few hundred tasks, which is too little for a high-capacity neural network. Once we bootstrap a (\mathcal{D}, θ) , we can draw unlimited samples from (\mathcal{D}, θ) and train Q on those samples. But, evaluating $\mathcal{L}_{\text{Dream}}$ involves drawing programs from the current DSL, running them to get their outputs, and then training Q to regress from the input/outputs to the program. Since these programs map inputs to outputs, we need to sample the inputs as well. Our solution is to sample the inputs from the empirical observed distribution of inputs in X .

The $\mathcal{L}_{\text{Dream}}^{\text{MAP}}$ objective involves finding the MAP program solving a task drawn from the DSL. To make this tractable, rather than *sample* programs as training data for $\mathcal{L}_{\text{Dream}}^{\text{MAP}}$, we *enumerate* programs in decreasing order of their prior probability, tracking, for each dreamed task x , the set of enumerated programs maximizing $\mathbb{P}[x, p | \mathcal{D}, \theta]$.

2.3.1 Parameterizing Q

Broadly the literature contains two different approaches to parameterizing conditional distributions over programs. The first approach, seen in systems like RobustFill [7] and [40], is to use a recurrent neural network to predict the program token-by-token as in Seq2Seq or Seq2Tree. The advantage of this approach is that the neural net can predict the entire program, so if the network is sufficiently powerful, it can completely solve the synthesis problem. There are two disadvantages to this approach. First, these models can perform poorly at out-of-sample generalization [], which is critical for our setting, as the agent may need to solve new tasks that are qualitatively different from the tasks it has solved so far. Second, a powerful deep recurrent network may be costly to sample or enumerate from — so if the network cannot easily solve a task, we cannot compensate with rapid sampling or enumeration. In contrast, state-of-the-art enumerative program synthesizers evaluate millions of programs per second [11].

The second approach is to have Q predict a fixed-dimensional weight vector, which then biases a fast enumerator [2] or sampler [24]. This approach can enjoy strong out-of-sample generalization, because it can fall back on enumeration or sampling when the target program is unlike the training programs. A main drawback is that the neural net is deliberately handicapped, and can only send so much information about the target program.

We adopt a middle ground between these two extremes. Our recognition model predicts a distribution over primitives in the DSL, conditioned on the local context in the syntax tree of the program. When predicting the next node to add to the syntax tree of a program, the recognition model conditions on the parent node, as well as which argument is being generated. This is a kind of ‘bigram’ model over trees, where the bigrams take the form of (parent, child, argument index).

Formally, Q predicts a $(|\mathcal{D}| + 2) \times (|\mathcal{D}| + 1) \times A$ -dimensional tensor, where A is the maximum arity² of any primitive in the DSL. Slightly abusing notation, we write this tensor as $Q_{ijk}(x)$, where x is a task, $i \in \mathcal{D} \cup \{\text{start}, \text{var}\}$, $j \in \mathcal{D} \cup \{\text{var}\}$, and $k \in \{1, 2, \dots, A\}$. The output $Q_{ijk}(x)$ controls the probability of sampling primitive j given that i is the parent node in the syntax tree and we are sampling the k^{th} argument. Figure 4 diagrams this generative process, Algorithm 2 specifies a sampling procedure for $Q(\cdot|x)$, and Figure 5 diagrams the neurally guided program inference procedure. This parameterization has three main advantages: (1) it supports fast enumeration and sampling of programs, because the recognition model only needs to run once for each task; (2) it allows the recognition model to provide

²The arity of a function is the number of arguments that it takes as input.

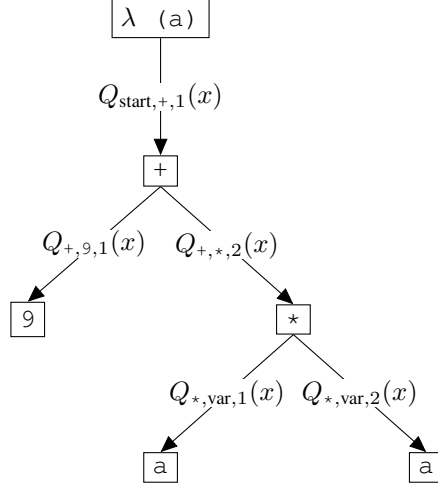


Figure 4: Parameterization of distribution over programs predicted by recognition model. Here the program (syntax tree shown above) is $(\lambda (a) (+ 9 (* a a)))$. Each conditional distribution predicted by the recognition model is written $Q_{\text{parent,child,argument index}}(x)$, where x is a task.

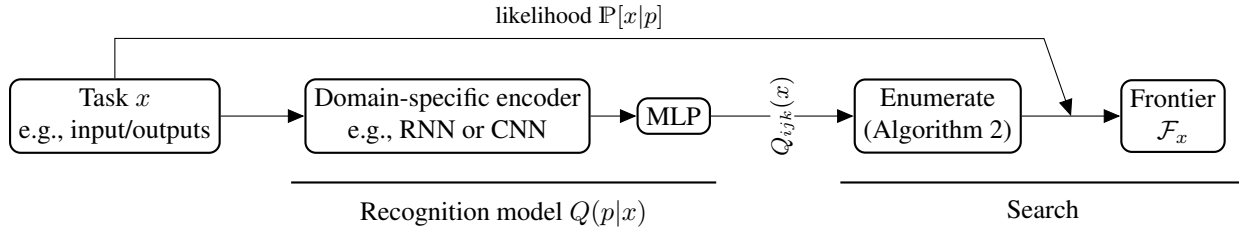


Figure 5: Neurally-guided program inference pipeline. Recognition model outputs distribution over program $Q(p|x)$ parameterized by tensor Q_{ijk} . Program output by enumerative search incorporated into frontier if likelihood $\mathbb{P}[x|p] > 0$

fine-grained information about the structure of the target program; and (3) training this recognition model causes it to learn to break symmetries in the space of programs, described next.

2.3.2 Learning to break symmetries in program space

A good DSL not only exposes high-level building blocks, but also carefully restricts the ways in which those building blocks are allowed to compose. For example, a DSL for list manipulation should contain both the empty list and a routine for appending lists, but should not allow appending the empty list. Similarly a DSL for arithmetic should contain both addition and the number zero but disallow adding zero. These restrictions break symmetries in the space of programs. Here we show that the combination of the \mathcal{L}^{MAP} training objective (Section ??) and bigram parameterization (Section 2.3.1) interact in a way that spontaneously breaks symmetries in the program space, allowing the agent to more efficiently explore the space of programs.

Consider an agent tasked with writing programs built from addition and the constants zero and one. A bigram parameterization of Q allows it to represent the fact that it should never add zero ($Q_{+,0,0} = Q_{+,0,1} = 0$) or that addition should always associate to the right ($Q_{+,+,0} = 0$). The \mathcal{L}^{MAP} training objective encourages learning these canonical forms. Consider two recognition models, Q_1 and Q_2 , and two programs in frontier \mathcal{F}_x , $p_1 = (+ (+ 1 1) 1)$ and $p_2 = (+ 1 (+ 1 1))$, where

$$\begin{aligned} Q_1(p_1|x) &= \frac{\epsilon}{2} & Q_1(p_2|x) &= \frac{\epsilon}{2} \\ Q_2(p_1|x) &= 0 & Q_2(p_2|x) &= \epsilon \end{aligned}$$

i.e., Q_2 breaks a symmetry by forcing right associative addition, but Q_1 does not, instead splitting its probability mass

Algorithm 2 Drawing from distribution over programs predicted by recognition model. Compare w/ Algorithm 3

```

1: function recognitionSample( $Q, x, \mathcal{D}, \tau$ ):
2: Input: recognition model  $Q$ , task  $x$ , DSL  $\mathcal{D}$ , type  $\tau$ 
3: Output: a program whose type unifies with  $\tau$ 
4: return recognitionSample'( $Q, x, \text{start}, 1, \mathcal{D}, \emptyset, \tau$ )

5: function recognitionSample'( $Q, x, \text{parent}, \text{argumentIndex}, \mathcal{D}, \mathcal{E}, \tau$ ):
6: Input: recognition model  $Q$ , task  $x$ , DSL  $\mathcal{D}$ ,  $\text{parent} \in \mathcal{D} \cup \{\text{start}, \text{var}\}$ ,  $\text{argumentIndex} \in \mathbb{N}$ , environment  $\mathcal{E}$ , type  $\tau$ 
7: Output: a program whose type unifies with  $\tau$ 
8: if  $\tau = \alpha \rightarrow \beta$  then ▷ Function type — start with a lambda
9:    $\text{var} \leftarrow$  an unused variable name
10:   $\text{body} \sim \text{recognitionSample}'(Q, x, \text{parent}, \text{argumentIndex}, \mathcal{D}, \{\text{var} : \alpha\} \cup \mathcal{E}, \beta)$ 
11:  return (lambda ( $\text{var}$ )  $\text{body}$ )
12: else ▷ Build an application to give something w/ type  $\tau$ 
13:   $\text{primitives} \leftarrow \{p \mid p : \tau' \in \mathcal{D} \cup \mathcal{E} \text{ if } \tau \text{ can unify with } \text{yield}(\tau')\}$  ▷ Everything in scope w/ type  $\tau$ 
14:   $\text{variables} \leftarrow \{p \mid p \in \text{primitives} \text{ and } p \text{ a variable}\}$ 
15:  Draw  $e \sim \text{primitives}$ , w.p.  $\propto \begin{cases} Q_{\text{parent}, e, \text{argumentIndex}}(x) & \text{if } e \in \mathcal{D} \\ Q_{\text{parent}, \text{var}, \text{argumentIndex}}(x) / |\text{variables}| & \text{if } e \in \mathcal{E} \end{cases}$ 
16:  Unify  $\tau$  with  $\text{yield}(\tau')$ . ▷ Ensure well-typed program
17:   $\text{newParent} \leftarrow \begin{cases} e & \text{if } e \in \mathcal{D} \\ \text{var} & \text{if } e \in \mathcal{E} \end{cases}$ 
18:   $\{\alpha_k\}_{k=1}^K \leftarrow \text{args}(\tau')$ 
19:  for  $k = 1$  to  $K$  do ▷ Recursively sample arguments
20:     $a_k \sim \text{recognitionSample}'(Q, x, \text{newParent}, k, \mathcal{D}, \mathcal{E}, \alpha_k)$ 
21:  end for
22:  return ( $e \ a_1 \ a_2 \ \dots \ a_K$ )
23: end if

```

	Unigram	Bigram
$\mathcal{L}^{\text{posterior}}$	<i>Three samples:</i>	<i>Three samples:</i>
	(+ 1 0)	0
	(+ (+ 0 0) (+ 1 0))	(+ (+ (+ 0 0) (+ 0 1)) 1)
	(+ 1 1)	1
	63.0% right-associative; 37.4% +0's	55.8% right-associative; 31.9% +0's
\mathcal{L}^{MAP}	<i>Three samples:</i>	<i>Three Samples:</i>
	1	(+ 1 (+ 1 (+ 1 (+ 1 (+ 1 1))))
	(+ 1 (+ 1 (+ (+ 1 (+ 1 1)) 1)))	0
	(+ (+ 1 1) 1)	(+ 1 (+ 1 (+ 1 1)))
	48.6% right-associative; 0.5% +0's	97.9% right-associative; 2.5% +0's

Figure 6: Agent learns to break symmetries in program space only when using both bigram parameterization and \mathcal{L}^{MAP} objective, associating addition to the right and avoiding adding zero. % right-associative calculated by drawing 500 samples from Q . \mathcal{L}^{MAP} /Unigram agent incorrectly learns to never generate programs with 0's, while \mathcal{L}^{MAP} /Bigram agent correctly learns that 0 should only be disallowed as an argument of addition. Tasked with building programs from +, 1, and 0.

equally between p_1 and p_2 . Now because $\mathbb{P}[p_1|\mathcal{D}, \theta] = \mathbb{P}[p_2|\mathcal{D}, \theta]$ (Algorithm 3), we have

$$\begin{aligned}
\mathcal{L}_{\text{real}}^{\text{posterior}}(Q_1) &= \frac{\mathbb{P}[p_1|\mathcal{D}, \theta] \log \frac{\epsilon}{2} + \mathbb{P}[p_2|\mathcal{D}, \theta] \log \frac{\epsilon}{2}}{\mathbb{P}[p_1|\mathcal{D}, \theta] + \mathbb{P}[p_2|\mathcal{D}, \theta]} = \log \frac{\epsilon}{2} \\
\mathcal{L}_{\text{real}}^{\text{posterior}}(Q_2) &= \frac{\mathbb{P}[p_1|\mathcal{D}, \theta] \log 0 + \mathbb{P}[p_2|\mathcal{D}, \theta] \log \epsilon}{\mathbb{P}[p_1|\mathcal{D}, \theta] + \mathbb{P}[p_2|\mathcal{D}, \theta]} = +\infty \\
\mathcal{L}_{\text{real}}^{\text{MAP}}(Q_1) &= \log Q_1(p_1) = \log Q_1(p_2) = \log \frac{\epsilon}{2} \\
\mathcal{L}_{\text{real}}^{\text{MAP}}(Q_2) &= \log Q_2(p_2) = \log \epsilon
\end{aligned}$$

So \mathcal{L}^{MAP} prefers Q_2 (the symmetry breaking recognition model), while $\mathcal{L}^{\text{posterior}}$ reverses this preference. We experimentally confirm this symmetry-breaking behavior by training recognition models that minimize either $\mathcal{L}^{\text{MAP}}/\mathcal{L}^{\text{posterior}}$ and which use either a bigram parameterization/unigram³ parameterization. Figure 6 shows the result of training Q in these four regimes for a DSL containing +, 0, and 1 and then sampling programs. On this particular run, the combination of bigrams and \mathcal{L}^{MAP} learns to avoid adding zero and associate addition to the right — different random initializations lead to either right or left association.

To be clear, our recognition model does not learn to break *every* possible symmetry in every possible DSL. But in practice we found that a bigrams combined with \mathcal{L}^{MAP} works well, and we use with this combination throughout the rest of the paper.

3 Experiments

3.1 Programs that manipulate sequences

We first apply DREAMCODER to two classic benchmark domains: list processing and text editing. We find that we can start with generic Lisp-style primitives and recover a domain-specific vocabulary for solving each of these classes of problems.

using a GRU [4] for the recognition model, and initially providing the system with generic sequence manipulation primitives: `foldr`, `unfold`, `if`, `map`, `length`, `index`, `=`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`.

List Processing: Synthesizing programs that manipulate data structures is a widely studied problem in the programming languages community [11], and progress here can translate to better computer-aided programming [34].

³In the unigram variant Q predicts a $|\mathcal{D}| + 1$ -dimensional vector: $Q(p|x) = \mathbb{P}[p|\mathcal{D}, \theta_i = Q_i(x)]$, and was used in our prior work [9]

We consider this problem within the context of learning functions that manipulate lists, and also perform arithmetic operations upon lists (Table 1). We created 236 human-interpretable list manipulation tasks, each with 15 input/output examples. In solving these tasks, the system composed 38 new subroutines, and rediscovered the higher-order function `filter` (f_1 in Table ??, left).

Text Editing: Synthesizing programs that edit text is a classic problem in the programming languages and AI literatures [13, 21]. This prior work uses hand-engineered DSLs. Here, we instead start out with generic sequence manipulation primitives and recover many of the higher-level building blocks that have made these other systems successful.

We trained our system on 109 automatically-generated text editing tasks, with 4 input/output examples each. After three iterations, it assembles a DSL containing a dozen new functions (Fig. ??, center) solving all the training tasks. But, how well does the learned DSL generalize to real text-editing scenarios? We tested, but did not train, on the 108 text editing problems from the SyGuS [1] program synthesis competition. Before any learning, DREAMCODER solves 3.7% of the problems with an average search time of 235 seconds. After learning, it solves 74.1%, and does so much faster, solving them in an average of 29 seconds. As of the 2017 SyGuS competition, the best-performing algorithm solves 79.6% of the problems. But, SyGuS comes with a different hand-engineered DSL *for each text editing problem*. Here we learned a single DSL that applied generically to all of the tasks, and perform comparably to the best prior work.

3.2 Programs that make plans and take actions

We apply DREAMCODER to two domains where the agent plans a series of actions in order to draw a picture (Sec. 3.2.1) or build a tower out of blocks (Sec. 3.2.2). For drawing pictures, the agent controls a ‘pen’ that it can pick up or place down while moving across a canvas. For building towers, the agent controls a ‘hand’ that it moves through a simulated world while placing down differently sized blocks. For each of these domains, the agent observes a target picture or block tower, and must either draw the picture or build the tower. The recognition model is a CNN that observes an image of the target picture or tower. The system is initially provided with two control flow operators: `loop` (a ‘for’ loop) and `get/set`, which saves and then restores the current state of the pen or hand.

3.2.1 Programs that draw pictures

We take inspiration from LOGO Turtle graphics [38], and task our agent with drawing 90 different pictures (Figure 7). The agent has primitives for moving the pen forward and rotating it (`move`), picking the pen up and then putting it down (`pen-up`), arithmetic operations on angles and distances, and constants (2π , 0 through 9, and ∞ ⁴).

Over the course of 5 wake/sleep iterations, the system authors a DSL containing parametric and, sometimes, higher-order drawing routines, allowing it to write programs for objects like polygons, circles, spirals, and snowflakes. Figure 8 illustrates several learned DSL subroutines. Once it has synthesized these new subroutines, the agent generates a rich and highly varied data set of dreams on which to train the recognition model. Figure 9 illustrates dreams before and after learning — one sees that the system has learned to recombine the building blocks and motifs in the training data.

3.2.2 Building towers out of ‘Lego’ blocks

Inspired by the classic AI ‘copy task’ — where an agent must look at an image of a tower made of toy blocks and re-create the tower [?] — we give DREAMCODER 112 tower ‘copy tasks’ (Figure ??). Here the agent observes both an image of a tower and the locations of each of its blocks, and must write a program that plans how a simulated hand

⁴To ensure that programs terminate, we set $\infty = 20$

Name	Input	Output
repeat-3	[7 0]	[7 0 7 0 7 0]
drop-3	[0 3 8 6 4]	[6 4]
rotate-2	[8 14 1 9]	[1 9 8 14]
count-head-in-tail	[1 2 1 1 3]	2
keep-div-5	[5 9 14 6 3 0]	[5 0]
product	[7 1 6 2]	84

Table 1: Some tasks in our list function domain.

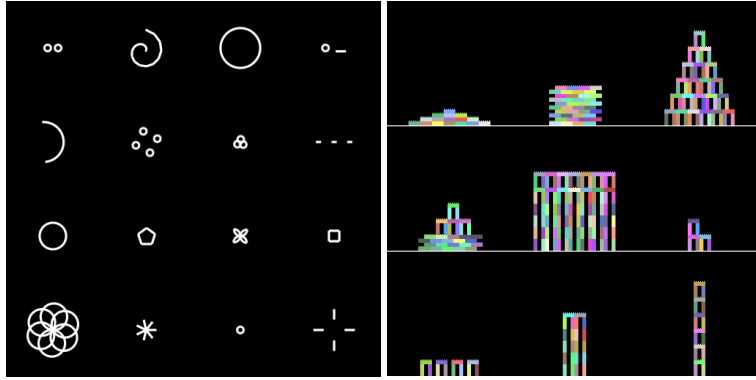


Figure 7: LOGO graphics tasks. The agent must learn to write programs that draw these 90 pictures.


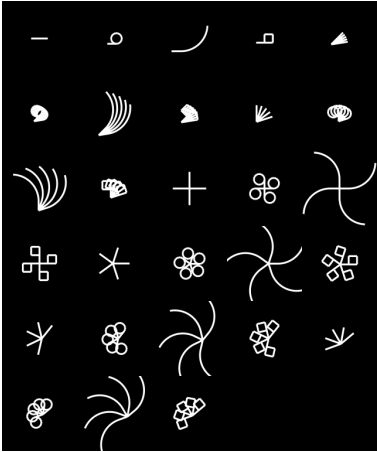
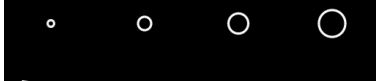
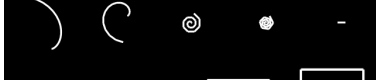
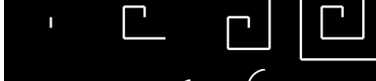


	Parametric drawing routines	Higher-order drawing routine
Semicircle:		Repeat and rotate subprogram: 
Circles:		
Spiral:		
Greek Spiral:		
S-Curves:		
Polygons & Stars:		

Figure 8: Example primitives learned by DREAMCODER when trained on tasks in Figure 7. Agent learns parametric routines for drawing families of curves (left) as well as subroutines that take entire programs as input (right). Each row of images on the left is the same code executed with different parameters. Each image on the right is the same code executed with different parameters and with a different subprogram provided as input.

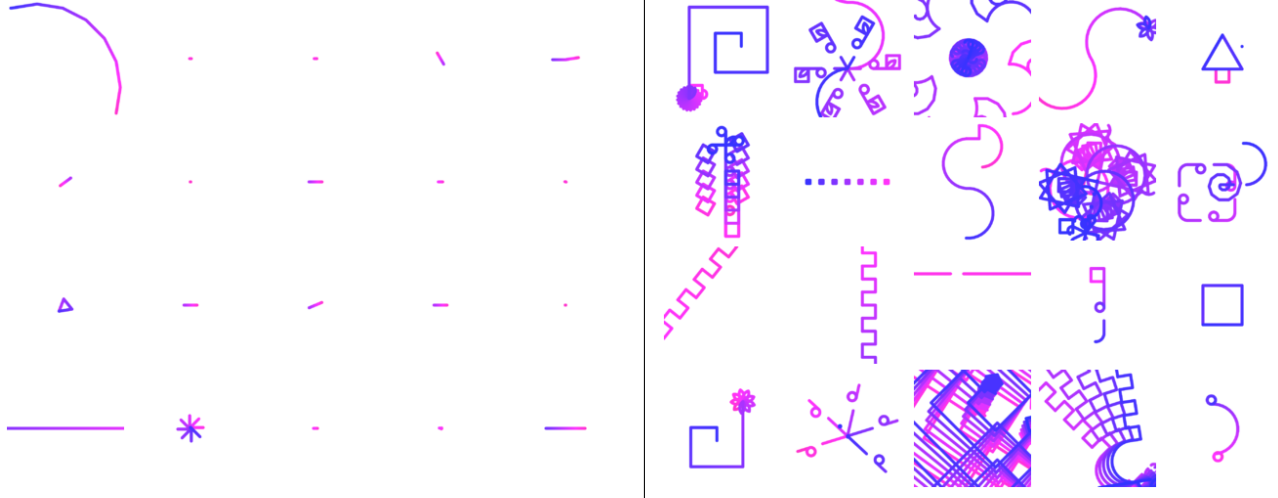


Figure 9: Dreams, or samples, from the DSL before (left) and after (right) training on tasks in Figure 7. Blue: where the agent started drawing. Pink: where the agent ended drawing.

Figure 10: Learned subroutines for building towers out of Lego-style blocks when trained on tasks in Figure ???. These subroutines act like parametric options [?], giving higher-level building blocks that the agent can use to plan.

would build the tower. These towers are built from Lego-style blocks that snap together on a discrete grid (e.g., we do not model gravity and discretize the coordinates of the blocks).

The system starts out with four domain-specific primitives: `left` and `right`, which take an integer n as input and move the hand left/right n Lego grid spaces; along with `horizontal` and `vertical`, which put down horizontal/vertical Lego blocks. As with LOGO graphics we also include our two standard planning primitives (`for` and `get/set`). Starting with this basis the system can only solve 12.5% of the copy tasks. With successive wake/sleep cycles we assemble a DSL containing parametric option-like subroutines (Figure ??), allowing the agent to solve 90% of the tasks.

3.3 Symbolic Regression: Programs from visual input

We apply DREAMCODER to symbolic regression problems. Here, the agent observes points along the curve of a function, and must write a program that fits those points. We initially equip our learner with addition, multiplication, and division, and task it with solving 100 problems, each either a polynomial or rational function. The recognition model is a convnet that observes an image of the target function’s graph (Fig. 11) — visually, different kinds of polynomials and rational functions produce different kinds of graphs, and so the convnet can look at a graph and predict what kind of function best explains it. A key difficulty, however, is that these problems are best solved with programs containing real numbers. Our solution to this difficulty is to enumerate programs with real-valued parameters, and then fit those parameters by automatically differentiating through the programs the system writes and use gradient descent to fit the parameters. We define the likelihood model, $P[x|p]$, by assuming a Gaussian noise model for the input/output examples, and penalize the use of real-valued parameters using the BIC [3]. We learn a DSL containing 13

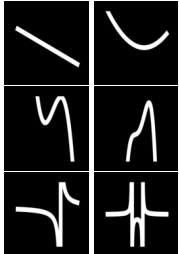


Figure 11: Recognition model input for symbolic regression. DSL learns subroutines for polynomials (top rows) and rational functions (bottom) while the recognition model jointly learns to look at a graph of the function (left) and predict which learned subroutines best explain the observation.

new functions, mainly templates for polynomials of different orders or ratios of polynomials. The model also learns to find programs minimizing the number of continuous parameters — learning to represent linear functions with `(* real (+ x real))`, which has two continuous parameters, and represents quartic functions using f_4 in the rightmost column of Fig. ?? which has five continuous parameters. This phenomenon arises from our Bayesian framing: both the generative model’s bias toward shorter programs, and the likelihood model’s BIC penalty.

3.4 Learning from Scratch

A long-standing dream within the program induction community is “learning from scratch”: starting with a *minimal* Turing-complete programming language, and then learning to solve a wide swath of induction problems [35, 33, 15, 36]. All existing systems, including ours, fall far short of this dream, and it is unclear (and we believe unlikely) that this dream could ever be fully realized. How far can we push in this direction? “Learning from scratch” is subjective, but a reasonable starting point is the set of primitives provided in 1959 Lisp [23]: these include conditionals, recursion, arithmetic, and the list operators `cons`, `car`, `cdr`, and `nil`. A basic first goal is to start with these primitives, and then recover a DSL that more closely resembles modern functional languages like Haskell and OCaml. Recall (Sec. 3.1) that we initially provided our system with functional programming routines like `map` and `fold`.

We ran the following experiment: DREAMCODER was given a subset of the 1959 Lisp primitives, and tasked with solving 22 programming exercises. A key difference between this setup and our previous experiments is that, for this experiment, the system is given primitive recursion, whereas previously we had sequestered recursion within higher-order functions like `map`, `fold`, and `unfold`.

After running for 93 hours on 64 CPUs, our algorithm solves these 22 exercises, along the way assembling a DSL with a modern repertoire of functional programming idioms and subroutines, including `map`, `fold`, `zip`, `unfold`, `index`, `length`, and arithmetic operations like building lists of natural numbers between an interval (see Figure 12). We did not use the recognition model for this experiment: a bottom-up pattern recognizer is of little use for acquiring this abstract knowledge from less than two dozen problems.

We believe that program learners should *not* start from scratch, but instead should start from a rich, domain-agnostic basis like those embodied in the standard libraries of modern languages. What this experiment shows is that DREAMCODER doesn’t *need* to start from a rich basis, and can in principle recover many of the amenities of modern programming systems, provided it is given enough computational power and a suitable spectrum of tasks.

3.5 Learning Generative Models

We apply DREAMCODER to learning generative models for images and text (Fig. ??-??). For images, we learn programs controlling a simulated “pen,” and the task is to look at an image and explain it in terms of a graphics program. For text, we learn probabilistic regular expressions – a simple probabilistic program for which inference is always tractable – and the task is to infer a regex from a collection of strings.

3.6 Quantitative Results on Held-Out Tasks

We evaluate on held-out testing tasks, measuring how many tasks are solved and how long it takes to solve them (Fig. 13). Prior to any learning, the system cannot find solutions for most of the tasks, and those it does solve take a long time; with more wake/sleep iterations, we converge upon DSLs and recognition models more closely matching the domain.

Programs & Tasks	DSL
<pre> [1 9]→2 [5 3 8]→3 f(ℓ) = (f₄ ℓ) [true false]→2 [false false false]→3 f(ℓ) = (f₄ ℓ) [2 1 4]→[2 1 4 0] [9 8]→[9 8 0] f(ℓ) = (f₂ cons ℓ (cons 0 nil)) [2 1 4]→[2 1] [9 8]→[9] f(ℓ) = (f₀ (λ (z) (empty? (cdr z))) car cdr ℓ) [2 5 6 0 6]→19 [9 2 7 6 3]→27 f(ℓ) = (f₂ + ℓ 0) [4 2 6 4]→[8 4 12 8] [2 3 0 7]→[4 6 0 14] f(ℓ) = (f₃ (λ (x) (+ x x)) ℓ) [4 2 6 4]→[-4 -2 -6 -4] [2 3 0 7]→[-2 -3 -0 -7] f(ℓ) = (f₃ (- 0) ℓ) [4 2 6 4]→[5 3 7 5] [2 3 0 7]→[3 4 1 8] f(ℓ) = (f₃ (+ 1) ℓ) [1 5 2 9]→[1 2] [3 8 1 3 1 2]→[3 1 1] f(ℓ) = (f₀ empty? car (λ (l) (cdr (cdr l))) ℓ) 3→[0 1 2] 2→[0 1] f(n) = (f₅ (λ (x) x) 0 n) 3→[0 1 2 3] 2→[0 1 2] f(n) = (f₅ (λ (x) x) 0 (+ 1 n)) </pre>	<pre> f₀(p,f,n,x) = (if (p x) nil (cons (f x) (f₀ (n x)))) (f₀: <i>unfold</i>) f₁(i,l) = (if (= i 0) (car l) (f₁ (- i 1) (cdr l))) (f₁: <i>index</i>) f₂(f,l,x) = (if (empty? l) x (f (car l) (f₂ (cdr l)))) (f₂: <i>fold</i>) f₃(f,l) = (f₂ nil l (λ (x a) (cons (f x) a))) (f₃: <i>map</i>) f₄(ℓ) = (if (empty? ℓ) 0 (+ 1 (f₄ (cdr ℓ)))) (f₄: <i>length</i>) f₅(f,m,n) = (f₀ (= m) f (+ 1) n) (f₅: <i>generalization of range</i>) </pre>

Figure 12: Bootstrapping a standard library of functional programming routines, starting from recursion along with primitive operations found in 1959 Lisp.

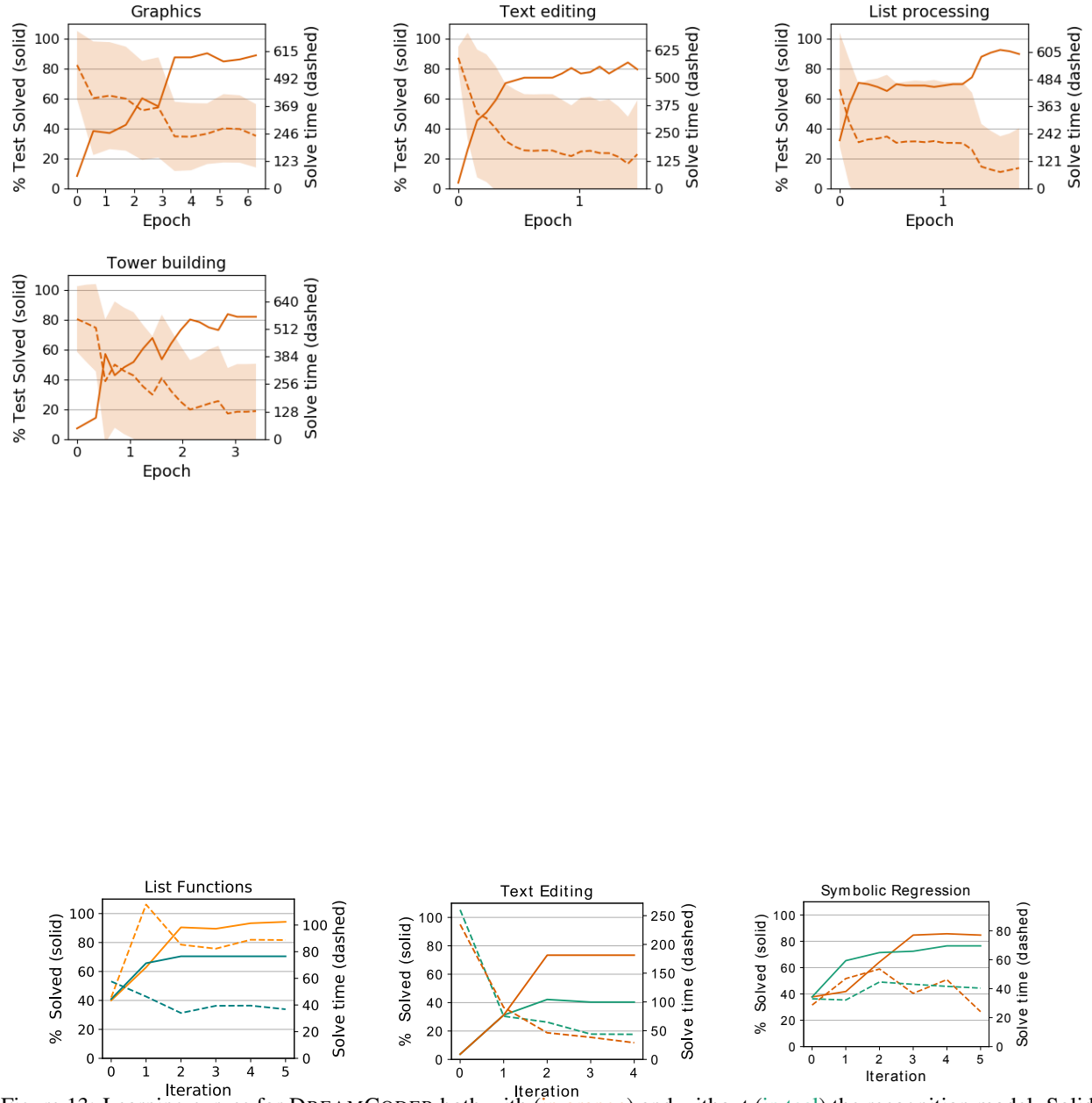


Figure 13: Learning curves for DREAMCODER both with (in orange) and without (in teal) the recognition model. Solid lines: % holdout testing tasks solved w/ 10m timeout. Dashed lines: Average solve time, averaged only over tasks that are solved.

4 Discussion

We contribute an algorithm, DREAMCODER, that learns to program by bootstrapping a DSL with new domain-specific primitives that the algorithm itself discovers, together with a neural recognition model that learns to efficiently deploy the DSL on new tasks. Two immediate future goals are to integrate more sophisticated neural recognition models [7] and program synthesizers [34], which may improve performance in some domains over the generic methods used here. Another direction is DSL meta-learning: can we find a *single* universal primitive set that could bootstrap DSLs for new domains, including the domains considered here, but also many others?

References

- [1] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.
- [2] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *ICLR*, 2016.
- [3] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. 2006.
- [4] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.
- [5] Peter Dayan, Geoffrey E Hinton, Radford M Neal, and Richard S Zemel. The helmholtz machine. *Neural computation*, 7(5):889–904, 1995.
- [6] Eyal Dechter, Jon Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. Bootstrap learning via modular concept discovery. In *IJCAI*, 2013.
- [7] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. *ICML*, 2017.
- [8] Yadin Dudai, Avi Karni, and Jan Born. The consolidation and transformation of memory. *Neuron*, 88(1):20 – 32, 2015.
- [9] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Library learning for neurally-guided bayesian program induction. In *NIPS*, 2017.
- [10] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B Tenenbaum. Learning to infer graphics programs from hand-drawn images. *NIPS*, 2018.
- [11] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, 2015.
- [12] Magdalena J Fosse, Roar Fosse, J Allan Hobson, and Robert J Stickgold. Dreaming and episodic memory: a functional dissociation? *Journal of cognitive neuroscience*, 15(1):1–9, 2003.
- [13] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.
- [14] Geoffrey E Hinton, Peter Dayan, Brendan J Frey, and Radford M Neal. The “wake-sleep” algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995.
- [15] Marcus Hutter. *Universal artificial intelligence: Sequential decisions based on algorithmic probability*. Springer Science & Business Media, 2004.
- [16] Irvin Hwang, Andreas Stuhlmüller, and Noah D. Goodman. Inducing probabilistic programs by bayesian program merging. *CoRR*, abs/1110.5667, 2011.

- [17] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *ICLR*, 2018.
- [18] John R. Koza. *Genetic programming - on the programming of computers by means of natural selection*. MIT Press, 1993.
- [19] J.D. Lafferty. *A Derivation of the Inside-outside Algorithm from the EM Algorithm*. Research report.
- [20] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.
- [21] Tessa Lau. *Programming by demonstration: a machine learning approach*. PhD thesis, 2001.
- [22] Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *ICML*, 2010.
- [23] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.
- [24] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *ICML*, pages 187–195, 2013.
- [25] Tom M Mitchell. Version spaces: A candidate elimination approach to rule learning. In *Proceedings of the 5th international joint conference on Artificial intelligence-Volume 1*, pages 305–310. Morgan Kaufmann Publishers Inc., 1977.
- [26] Stephen H Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.
- [27] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, volume 50, pages 619–630. ACM, 2015.
- [28] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
- [29] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.
- [30] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.
- [31] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 305–316. ACM, 2013.
- [32] Ute Schmid and Emanuel Kitzelmann. Inductive rule learning on the knowledge level. *Cognitive Systems Research*, 12(3-4):237–248, 2011.
- [33] Jürgen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.
- [34] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, 2008.
- [35] Ray J Solomonoff. A formal theory of inductive inference. *Information and control*, 7(1):1–22, 1964.
- [36] Ray J Solomonoff. A system for incremental learning based on algorithmic probability. Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition, 1989.
- [37] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pages 212–223. Springer, 2002.
- [38] David D. Thornburg. Friends of the turtle. *Compute!*, March 1983.
- [39] T. Ullman, N. D. Goodman, and J. B. Tenenbaum. Theory learning as stochastic search in the language of thought. *Cognitive Development*, 27(4):455–480, 2012.
- [40] Maksym Zavershynskyi, Alex Skidanov, and Illia Polosukhin. Naps: Natural program synthesis dataset. In *ICML*, 2018.

A Appendix

A.1 Generative model over programs

Algorithm 3 gives a stochastic procedure for drawing samples from $\mathbb{P}[\cdot | \mathcal{D}, \theta]$. It takes as input the desired type of the unknown program, and performs type inference during sampling to ensure that the program has the desired type. It also maintains a *environment* mapping variables to types, which ensures that lexical scoping rules are obeyed.

Algorithm 3 Generative model over programs

```

1: function sample( $\mathcal{D}, \theta, \tau$ ):
2: Input: DSL ( $\mathcal{D}, \theta$ ), type  $\tau$ 
3: Output: a program whose type unifies with  $\tau$ 
4: return sample'( $\mathcal{D}, \theta, \emptyset, \tau$ )

5: function sample'( $\mathcal{D}, \theta, \mathcal{E}, \tau$ ):
6: Input: DSL ( $\mathcal{D}, \theta$ ), environment  $\mathcal{E}$ , type  $\tau$                                 ▷ Environment  $\mathcal{E}$  starts out as  $\emptyset$ 
7: Output: a program whose type unifies with  $\tau$ 
8: if  $\tau = \alpha \rightarrow \beta$  then                                                    ▷ Function type — start with a lambda
9:   var  $\leftarrow$  an unused variable name
10:  body  $\sim$  sample'( $\mathcal{D}, \theta, \{\text{var} : \alpha\} \cup \mathcal{E}, \beta$ )                                ▷ Recursively sample function body
11:  return (lambda (var) body)
12: else                                                                    ▷ Build an application to give something w/ type  $\tau$ 
13:  primitives  $\leftarrow \{p | p : \tau' \in \mathcal{D} \cup \mathcal{E} \text{ if } \tau \text{ can unify with } \text{yield}(\tau')\}$     ▷ Everything in scope w/ type  $\tau$ 
14:  variables  $\leftarrow \{p \mid p \in \text{primitives and } p \text{ a variable}\}$ 
15:  Draw  $e \sim \text{primitives}$ , w.p.  $\propto \begin{cases} \theta_e & \text{if } e \in \mathcal{D} \\ \theta_{var}/|\text{variables}| & \text{if } e \in \mathcal{E} \end{cases}$ 
16:  Unify  $\tau$  with  $\text{yield}(\tau')$ .                                                    ▷ Ensure well-typed program
17:   $\{\alpha_k\}_{k=1}^K \leftarrow \text{args}(\tau')$ 
18:  for  $k = 1$  to  $K$  do                                                        ▷ Recursively sample arguments
19:     $a_k \sim \text{sample}'(\mathcal{D}, \theta, \mathcal{E}, \alpha_k)$ 
20:  end for
21:  return ( $e \ a_1 \ a_2 \ \dots \ a_K$ )
22: end if

where:
23:  $\text{yield}(\tau) = \begin{cases} \text{yield}(\beta) & \text{if } \tau = \alpha \rightarrow \beta \\ \tau & \text{otherwise.} \end{cases}$                                 ▷ Final return type of  $\tau$ 
24:  $\text{args}(\tau) = \begin{cases} [\alpha] + \text{args}(\beta) & \text{if } \tau = \alpha \rightarrow \beta \\ [] & \text{otherwise.} \end{cases}$                                 ▷ Types of arguments needed to get something w/ type  $\tau$ 

```

A.2 Enumerative program search

Our current implementation of DREAMCODER takes the simple and generic strategy of enumerating programs in descending order of their probability under either (\mathcal{D}, θ) or $Q(p|x)$. Algorithm 3 and 2 specify procedures for sampling from these distributions, but not for enumerating from them. We combine two different enumeration strategies, which allowed us to build a massively parallel program enumerator:

- **Best-first search:** Best-first search maintains a heap of partial programs ordered by their probability — here a partial program means a program whose syntax tree may contain unspecified ‘holes’. Best-first search is guaranteed to enumerate programs in decreasing order of their probability, and has memory requirements that in general grow exponentially as a function of the description length of programs in the heap (thus linearly as a function of run time).

- **Depth-first search:** Depth first search recursively explores the space of execution traces through Algorithm 3 and 2, equivalently maintaining a stack of partial programs. In general it does not enumerate programs in decreasing order of probability, but has memory requirements that grow linearly as a function of the description length of the programs in the stack (thus logarithmically as a function of run time).

Our parallel enumeration algorithm (Algorithm 4) first performs a best-first search until the best-first heap is much larger than the number of CPUs. At this point, it switches to performing many depth-first searches in parallel, initializing a depth first search with one of the entries in the best-first heap. Because depth-first search does not produce programs in decreasing order of their probability, we wrap this entire procedure up into an outer loop that first enumerates programs whose description length is between 0 to Δ , then programs with description length between Δ and 2Δ , then 2Δ to 3Δ , etc., until a timeout is reached. This is similar in spirit to iterative deepening depth first search [30].

A.3 Refactoring code with version spaces

Formally, a version space is either:

- A deBuijn⁵ index: written $\$i$, where i is a natural number
- An abstraction: written λv , where v is a version space
- An application: written $(f \ x)$, where both f and x are version spaces
- A union: $\uplus V$, where V is a set of version spaces
- The empty set, \emptyset
- The set of all λ -calculus expressions, Λ

The purpose of a version space to compactly represent a set of programs. We refer to this set as the **extension** of the version space:

Definition 2. The **extension** of a version space v is written $\llbracket v \rrbracket$ and is defined recursively as:

$$\begin{aligned} \llbracket \$i \rrbracket &= \{\$i\} & \llbracket \lambda v \rrbracket &= \{\lambda e : e \in \llbracket v \rrbracket\} & \llbracket (v_1 \ v_2) \rrbracket &= \{(e_1 \ e_2) : e_1 \in \llbracket v_1 \rrbracket, \ e_2 \in \llbracket v_2 \rrbracket\} \\ \llbracket \uplus V \rrbracket &= \{e : v \in V, \ e \in \llbracket v \rrbracket\} & \llbracket \emptyset \rrbracket &= \emptyset & \llbracket \Lambda \rrbracket &= \Lambda \end{aligned}$$

Version spaces also support efficient membership checking, which we write as $e \in \llbracket v \rrbracket$. Important for our purposes, it is also efficient to refactor the members of a version space's extension in terms of a new DSL. We define $\text{REFACTOR}(v|\mathcal{D})$ inductively as:

$$\text{REFACTOR}(v|\mathcal{D}) = \begin{cases} e, & \text{if } e \in \mathcal{D} \text{ and } e \in \llbracket v \rrbracket. \text{ Exploits the fact that } \llbracket e \rrbracket \in v \text{ can be efficiently computed.} \\ \text{REFACTOR}'(v|\mathcal{D}), & \text{otherwise.} \end{cases}$$

$$\begin{aligned} \text{REFACTOR}'(e|\mathcal{D}) &= e, \text{ if } e \text{ is a leaf} & \text{REFACTOR}'(\lambda b|\mathcal{D}) &= \lambda \text{REFACTOR}(b|\mathcal{D}) \\ \text{REFACTOR}'(f \ x|\mathcal{D}) &= \text{REFACTOR}(f|\mathcal{D}) \ \text{REFACTOR}(x|\mathcal{D}) & \text{REFACTOR}'(\uplus V|\mathcal{D}) &= \arg \min_{e \in \{\text{REFACTOR}(v|\mathcal{D}) : v \in V\}} \text{size}(e|\mathcal{D}) \end{aligned}$$

where $\text{size}(e|\mathcal{D})$ for program e and DSL \mathcal{D} is the size of the syntax tree of e , when members of \mathcal{D} are counted as having size 1. Concretely, $\text{REFACTOR}(v|\mathcal{D})$ calculates $\arg \min_{p \in \llbracket v \rrbracket} \text{size}(p|\mathcal{D})$.

Recall that our goal is to define an operator over version spaces, $I\beta_n$, which calculates the set of n -step refactorings. We define this operator in terms of another operator, $I\beta'$, which performs a single step of refactoring:

$$I\beta_n(v) = \uplus \left\{ \underbrace{I\beta'(I\beta'(I\beta'(\dots v)))}_{i \text{ times}} : 0 \leq i \leq n \right\}$$

⁵deBuijn indices are an alternative way of naming variables in λ -calculus. When using deBuijn indices, λ -abstractions are written *without* a variable name, and variables are written as the count of the number of λ -abstractions up in the syntax tree the variable is bound to. For example, $\lambda x. \lambda y. (x \ y)$ is written $\lambda \lambda (\$1 \ \$0)$ using deBuijn indices. See [28] for more details.

Algorithm 4 Parallel enumerative program search algorithm

```
1: function enumerate( $\mu, T, \text{CPUs}$ ):  
2: Input: Distribution over programs  $\mu$ , timeout  $T$ , CPU count  
3: Output: stream of programs in approximately descending order of probability under  $\mu$   
4: Hyperparameter: nat increase rate  $\Delta$  ▷ We set  $\Delta = 1.5$   
5: lowerBound  $\leftarrow$  0  
6: while total elapsed time  $< T$  do  
7:   heap  $\leftarrow$  newMaxHeap() ▷ Heap for best-first search  
8:   heap.insert(priority = 0, value = empty syntax tree) ▷ Initialize heap with start state of search space  
9:   while  $0 < |\text{heap}| \leq 10 \times \text{CPUs}$  do ▷ Each CPU will get approximately 10 jobs (a partial program)  
10:    priority, partialProgram  $\leftarrow$  heap.popMaximum()  
11:    if partialProgram is finished then ▷ Nothing more to fill in in the syntax tree  
12:      if lowerBound  $\leq -\text{priority} < \text{lowerBound} + \Delta$  then  
13:        yield partialProgram  
14:      end if  
15:    else  
16:      for child  $\in$  children(partialProgram) do ▷ children( $\cdot$ ) fills in next random choice in syntax tree.  
17:        if  $-\log \mu(\text{child}) < \text{lowerBound} + \Delta$  then ▷ Child's description length small enough  
18:          heap.insert(priority =  $\log \mu(\text{child})$ , value = child)  
19:        end if  
20:      end for  
21:    end if  
22:  end while  
23:  yield from ParallelMapCPUs(depthFirst( $\mu, T - \text{elapsed time}, \text{lowerBound}, \cdot$ ), heap.values())  
24:  lowerBound  $\leftarrow$  lowerBound +  $\Delta$  ▷ Push up lower bound on MDL by  $\Delta$   
25: end while  
  
26: function depthFirst( $\mu, T, \text{lowerBound}, \text{partialProgram}$ ): ▷ Each worker does a depth first search. Enumerates completions of partialProgram whose MDL is between lowerBound and lowerBound +  $\Delta$   
27: stack  $\leftarrow$  [partialProgram]  
28: while total elapsed time  $< T$  and stack is not empty do  
29:   partialProgram  $\leftarrow$  stack.pop()  
30:   if partialProgram is finished then  
31:     if lowerBound  $\leq -\log \mu(\text{partialProgram}) < \text{lowerBound} + \Delta$  then  
32:       yield partialProgram  
33:     end if  
34:   else  
35:     for child  $\in$  children(partialProgram) do  
36:       if  $-\log \mu(\text{child}) < \text{lowerBound} + \Delta$  then ▷ Child's description length small enough  
37:         stack.push(child)  
38:       end if  
39:     end for  
40:   end if  
41: end while
```

where

$$I\beta'(u) = \uplus \{(\lambda b)v : v \mapsto b \in S_0(u)\} \cup \begin{cases} \text{if } u \text{ is a primitive or index or } \emptyset: & \emptyset \\ \text{if } u \text{ is } \Lambda: & \{\Lambda\} \\ \text{if } u = \lambda b: & \{\lambda I\beta'(b)\} \\ \text{if } u = (f \ x): & \{(I\beta'(f) \ x), (f \ I\beta'(x))\} \\ \text{if } u = \uplus V: & \{I\beta'(u') \mid u' \in V\} \end{cases}$$

where we have defined $I\beta'$ in terms of another operator, $S_k : VS \rightarrow 2^{VS \times VS}$, whose purpose is to construct the set of substitutions that are refactorings of a program in a version space. We define S as:

$$S_k(v) = \{\downarrow_0^k v \mapsto \$k\} \cup \begin{cases} \text{if } v \text{ is primitive:} & \{\Lambda \mapsto v\} \\ \text{if } v = \$i \text{ and } i < k: & \{\Lambda \mapsto \$i\} \\ \text{if } v = \$i \text{ and } i \geq k: & \{\Lambda \mapsto \$(i+1)\} \\ \text{if } v = \lambda b: & \{v' \mapsto \lambda b' : v' \mapsto b' \in S_{k+1}(b)\} \\ \text{if } v = (f \ x): & \{v_1 \cap v_2 \mapsto (f' \ x') : v_1 \mapsto f' \in S_k(f), v_2 \mapsto x' \in S_k(x)\} \\ \text{if } v = \uplus V: & \bigcup_{v' \in V} S_n(v') \\ \text{if } v \text{ is } \emptyset: & \emptyset \\ \text{if } v \text{ is } \Lambda: & \{\Lambda \mapsto \Lambda\} \end{cases}$$

$$\begin{aligned} \downarrow_c^k \$i &= \$i, \text{ when } i < c \\ \downarrow_c^k \$i &= \$(i-k), \text{ when } i \geq c+k \\ \downarrow_c^k \$i &= \emptyset, \text{ when } c \leq i < c+k \\ \downarrow_c^k \lambda b &= \lambda \downarrow_{c+1}^k b \\ \downarrow_c^k (f \ x) &= (\downarrow_c^k f \ \downarrow_c^k x) \\ \downarrow_c^k \uplus V &= \uplus \{\downarrow_c^k v \mid v \in V\} \\ \downarrow_c^k v &= v, \text{ when } v \text{ is a primitive or } \emptyset \text{ or } \Lambda \end{aligned}$$

where \uparrow^k is the shifting operator [28], which adds k to all of the free variables in a λ -expression or version space, and we have defined a new operator, \downarrow , whose purpose is to undo the action of \uparrow . We have written definitions recursively, but implement them using a dynamic program: we hash cons each version space, and only calculate the operators $I\beta_n$, $I\beta'$, and S_k once per each version space.

We now formally prove that $I\beta$ exhaustively enumerates the space of possible refactorings. Our approach is to first prove that S_k exhaustively enumerates the space of possible substitutions that could give rise to a program. The following pair of technical lemmas are useful; both are easily proven by structural induction.

Lemma 1. *Let e be a program or version space and n, c be natural numbers. Then $\uparrow_{n+c}^{-1} \uparrow_c^{n+1} e = \uparrow_c^n e$, and in particular $\uparrow_n^{-1} \uparrow^{n+1} e = \uparrow^n e$.*

Lemma 2. *Let e be a program or version space and n, c be natural numbers. Then $\downarrow_c^n \uparrow_c^n e = e$, and in particular $\downarrow^n \uparrow^n e = e$.*

Theorem 1. Consistency of S_n .

If $(v \mapsto b) \in S_n(u)$ then for every $v' \in v$ and $b' \in b$ we have $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v'] b' \in u$.

Proof. Suppose $b = \$n$ and therefore, by the definition of S_n , also $v = \downarrow_0^n u$. Invoking Lemmas 1 and 2 we know that $u = \uparrow_n^{-1} \uparrow^{n+1} v$ and so for every $v' \in v$ we have $\uparrow_n^{-1} \uparrow^{n+1} v' \in u$. Because $b = \$n = b'$ we can rewrite this to $\uparrow_n^{-1} [\$n \mapsto \uparrow^{n+1} v'] b' \in u$.

Otherwise assume $b \neq \$n$ and proceed by structural induction on u :

- If $u = \$i < n$ then we have to consider the case that $v = \Lambda$ and $b = u = \$i = b'$. Pick $v' \in \Lambda$ arbitrarily. Then $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v'] b' = \uparrow_n^{-1} \$i = \$i \in u$.

- If $u = \$i \geq n$ then we have consider the case that $v = \Lambda$ and $b = \$(i + 1) = b'$. Pick $v' \in \Lambda$ arbitrarily. Then $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']b' = \uparrow_n^{-1} \$(i + 1) = \$i \in u$.
- If u is primitive then we have to consider the case that $v = \Lambda$ and $b = u = b'$. Pick $v' \in \Lambda$ arbitrarily. Then $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']b' = \uparrow_n^{-1} u = u \in u$.
- If u is of the form λa , then $S_n(u) \subset \{v \mapsto \lambda b \mid (v \mapsto b) \in S_{n+1}(a)\}$. Let $v \mapsto \lambda b \in S_n(u)$. By induction for every $v' \in v$ and $b' \in b$ we have $\uparrow_{n+1}^{-1} [\$n \mapsto \uparrow^{2+n} v']b' \in a$, which we can rewrite to $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']\lambda b' \in \lambda a = u$.
- If u is of the form $(f \ x)$ then then $S_n(u) \subset \{v_f \cap v_x \mapsto (b_f \ b_x) \mid (v_f \mapsto b_f) \in S_n(f), (v_x \mapsto b_x) \in S_n(x)\}$. Pick $v' \in v_f \cap v_x$ arbitrarily. By induction for every $v'_f \in v_f, v'_x \in v_x, b'_f \in b_f, b'_x \in b_x$ we have $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v'_f]b'_f \in f$ and $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v'_x]b'_x \in x$. Combining these facts gives $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v'](b'_f \ b'_x) \in (f \ x) = u$.
- If u is of the form $\uplus U$ then pick $(v \mapsto b) \in S_n(u)$ arbitrarily. By the definition of S_n there is a z such that $(v \mapsto b) \in S_n(z)$, and the theorem holds immediately by induction.
- If u is \emptyset or Λ then the theorem holds vacuously.

□

Theorem 2. Completeness of S_n .

If there exists programs v' and b' , and a version space u , such that $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']b' \in u$, then there also exists $(v \mapsto b) \in S_n(u)$ such that $v' \in v$ and $b' \in b$.

Proof. As before we first consider the case that $b' = \$n$. If so then $\uparrow_n^{-1} \uparrow^{1+n} v' \in u$ or (invoking Lemma 1) that $\uparrow^n v' \in u$ and (invoking Lemma 2) that $v' \in \downarrow^n u$. From the definition of S_n we know that $(\downarrow^n u \mapsto \$n) \in S_n(u)$ which is what was to be shown.

Otherwise assume that $b' \neq \$n$. Proceeding by structural induction on u :

- If $u = \$i$ then, because b' is not $\$n$, we have $\uparrow_n^{-1} b' = \$i$. Let $b' = \$j$, and so

$$i = \begin{cases} j & \text{if } j < n \\ j - 1 & \text{if } j > n \end{cases}$$

where $j = n$ is impossible because by assumption $b' \neq \$n$.

If $j < n$ then $i = j$ and so $u = b'$. By the definition of S_n we have $(\Lambda \mapsto \$i) \in S_n(u)$, completing this inductive step because $v' \in \Lambda$ and $b' \in \$i$. Otherwise assume $j > n$ and so $\$i = \$(j - 1) = u$. By the definition of S_n we have $(\Lambda \mapsto \$(i + 1)) \in S_n(u)$, completing this inductive step because $v' \in \Lambda$ and $b' = \$j = \$(i + 1)$.

- If u is a primitive then, because b' is not $\$n$, we have $\uparrow_n^{-1} b' = u$, and so $b' = u$. By the definition of S_n we have $(\Lambda \mapsto u) \in S_n(u)$ completing this inductive step because $v' \in \Lambda$ and $b' = u$.
- If u is of the form λa then, because of the assumption that $b' \neq \$n$, we know that b' is of the form $\lambda c'$ and that $\lambda \uparrow_{n+1}^{-1} [\$n \mapsto \uparrow^{2+n} v']c' \in \lambda a$. By induction this means that there is a $(v \mapsto c) \in S_{n+1}(a)$ satisfying $v' \in v$ and $c' \in c$. By the definition of S_n we also know that $(v \mapsto \lambda c) \in S_n(u)$, completing this inductive step because $b' = \lambda c' \in \lambda c$.
- If u is of the form $(f \ x)$ then, because of the assumption that $b' \neq \$n$, we know that b' is of the form $(b'_f \ b'_x)$ and that both $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']b'_f \in f$ and $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']b'_x \in x$. Invoking the inductive hypothesis twice gives a $(v_f \mapsto b_f) \in S_n(f)$ satisfying $v' \in v_f, b'_f \in b_f$ and a $(v_x \mapsto b_x) \in S_n(x)$ satisfying $v' \in v_x, b'_x \in b_x$. By the definition of S_n we know that $(v_f \cap v_x \mapsto b_f \ b_x) \in S_n(u)$ completing the inductive step because v' is guaranteed to be in both v_f and v_x and we know that $b' = (b'_f \ b'_x) \in (b_f \ b_x)$.
- If u is of the form $\uplus U$ then there must be a $z \in U$ such that $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']b' \in z$. By induction there is a $(v \mapsto b) \in S_n(z)$ such that $v' \in v$ and $b' \in b$. By the definition of S_n we know that $(v \mapsto b)$ is also in $S_n(u)$ completing the inductive step.

- If u is \emptyset or Λ then the theorem holds vacuously.

□

From these results the consistency and completeness of $I\beta$ follows:

Theorem 3. Consistency of $I\beta'$.

If $p \in \llbracket I\beta'(u) \rrbracket$ then there exists $p' \in \llbracket u \rrbracket$ such that $p \longrightarrow p'$.

Proof. Proceed by structural induction on u . If $p \in \llbracket I\beta'(u) \rrbracket$ then, from the definition of $I\beta'$ and $\llbracket \cdot \rrbracket$, at least one of the following holds:

- Case $p = (\lambda b')v'$ where $v' \in v$, $b' \in b$, and $v \mapsto b \in S_0(u)$: From the definition of β -reduction we know that $p \longrightarrow \uparrow^{-1} [\$0 \mapsto \uparrow^1 v']b'$. From the consistency of S_n we know that $\uparrow^{-1} [\$0 \mapsto \uparrow^1 v']b' \in u$. Identify $p' = \uparrow^{-1} [\$0 \mapsto \uparrow^1 v']b'$.
- Case $u = \lambda b$ and $p = \lambda b'$ where $b' \in \llbracket I\beta'(b) \rrbracket$: By induction there exists $b'' \in \llbracket b \rrbracket$ such that $b' \longrightarrow b''$. So $p \longrightarrow \lambda b''$. But $\lambda b'' \in \llbracket \lambda b \rrbracket = \llbracket u \rrbracket$, so identify $p' = \lambda b''$.
- Case $u = (f x)$ and $p = (f' x')$ where $f' \in \llbracket I\beta'(f) \rrbracket$ and $x' \in \llbracket x \rrbracket$: By induction there exists $f'' \in \llbracket f \rrbracket$ such that $f' \longrightarrow f''$. So $(f' x') \longrightarrow (f'' x')$. But $(f'' x') \in \llbracket (f x) \rrbracket = \llbracket u \rrbracket$, so identify $p' = (f'' x')$.
- Case $u = (f x)$ and $p = (f' x')$ where $x' \in \llbracket I\beta'(x) \rrbracket$ and $f' \in \llbracket f \rrbracket$: Symmetric to the previous case.
- Case $u = \uplus U$ and $p \in \llbracket I\beta'(u') \rrbracket$ where $u' \in U$: By induction there is a $p' \in \llbracket u' \rrbracket$ satisfying $p' \longrightarrow p$. But $\llbracket u' \rrbracket \subseteq \llbracket u \rrbracket$, so also $p' \in \llbracket u \rrbracket$.
- Case u is an index, primitive, \emptyset , or Λ : The theorem holds vacuously.

□

Theorem 4. Completeness of $I\beta'$.

Let $p \longrightarrow p'$ and $p' \in \llbracket u \rrbracket$. Then $p \in \llbracket I\beta'(u) \rrbracket$.

Proof. Structural induction on u . If $u = \uplus V$ then there is a $v \in V$ such that $p' \in \llbracket v \rrbracket$; by induction on v combined with the definition of $I\beta'$ we have $p \in \llbracket I\beta'(v) \rrbracket \subseteq \llbracket I\beta'(u) \rrbracket$, which is what we were to show. Otherwise assume that $u \neq \uplus V$.

From the definition of $p \longrightarrow p'$ at least one of these cases must hold:

- Case $p = (\lambda b')v'$ and $p' = \uparrow^{-1} [\$0 \mapsto \uparrow^1 v']b'$: Using the fact that $\uparrow^{-1} [\$0 \mapsto \uparrow^1 v']b' \in \llbracket u \rrbracket$, we can invoke the completeness of S_n to construct a $(v \mapsto b) \in S_0(u)$ such that $v' \in \llbracket v \rrbracket$ and $b' \in \llbracket b \rrbracket$. Combine these facts with the definition of $I\beta'$ to get $p = (\lambda b')v' \in \llbracket (\lambda b)v \rrbracket \subseteq \llbracket I\beta'(u) \rrbracket$.
- Case $p = \lambda b$ and $p' = \lambda b'$ where $b \longrightarrow b'$: Because $p' = \lambda b' \in \llbracket u \rrbracket$ and by assumption $u \neq \uplus V$, we know that $u = \lambda v$ and $b' \in \llbracket v \rrbracket$. By induction $b \in \llbracket I\beta'(v) \rrbracket$. Combine with the definition of $I\beta'$ to get $p = \lambda b \in \llbracket \lambda I\beta'(v) \rrbracket \subseteq \llbracket I\beta'(u) \rrbracket$.
- Case $p = (f x)$ and $p' = (f' x)$ where $f \longrightarrow f'$: Because $p' = (f' x) \in \llbracket u \rrbracket$ and by assumption $u \neq \uplus V$ we know that $u = (a b)$ where $f' \in \llbracket a \rrbracket$ and $x \in \llbracket b \rrbracket$. By induction on a we know $f \in \llbracket I\beta'(a) \rrbracket$. Therefore $p = (f x) \in \llbracket (I\beta'(a) b) \rrbracket \subseteq \llbracket I\beta'((a b)) \rrbracket \subseteq \llbracket I\beta'(u) \rrbracket$.
- Case $p = (f x)$ and $p' = (f x')$ where $x \longrightarrow x'$: Symmetric to the previous case.

□

Finally we have our main result:

Theorem 5. Consistency and completeness of $I\beta_n$. Let p and p' be programs. Then $p \longrightarrow q \longrightarrow \cdots \longrightarrow p'$ if and only if $p \in \llbracket I\beta_n(p') \rrbracket$.

$\leq n \text{ times}$

Proof. Induction on n .

If $n = 0$ then $\llbracket I\beta_n(p') \rrbracket = \{p\}$ and $p = p'$; the theorem holds immediately. Assume $n > 0$.

If $p \xrightarrow{\leq n \text{ times}} q \xrightarrow{\leq n-1 \text{ times}} \dots \xrightarrow{\leq n-1 \text{ times}} p'$ then $q \xrightarrow{\leq n-1 \text{ times}} \dots \xrightarrow{\leq n-1 \text{ times}} p'$; induction on n gives $q \in \llbracket I\beta_{n-1}(p') \rrbracket$. Combined with $p \xrightarrow{\leq n \text{ times}} q$

we can invoke the completeness of $I\beta'$ to get $p \in \llbracket I\beta'(I\beta_{n-1}(p')) \rrbracket \subset \llbracket I\beta_n(p') \rrbracket$.

If $p \in \llbracket I\beta_n(p') \rrbracket$ then there exists a $i \leq n$ such that $p \in \llbracket I\beta'(I\beta'(I\beta'(\dots p')) \rrbracket$. If $i = 0$ then $p = p'$ and p reduces to p' in $0 \leq n$ steps. Otherwise $i > 0$ and $p \in \llbracket I\beta'(I\beta'(I\beta'(\dots p')) \rrbracket$. Invoking the consistency of $I\beta'$ we know that $p \xrightarrow{\leq i-1 \text{ times}} q$ for a program $q \in \llbracket I\beta'(I\beta'(\dots p')) \rrbracket \subseteq \llbracket I\beta_{i-1}(p') \rrbracket$. By induction $q \xrightarrow{\leq i-1 \text{ times}} \dots \xrightarrow{\leq i-1 \text{ times}} p'$, which combined with $p \xrightarrow{\leq i-1 \text{ times}} q$ gives $p \xrightarrow{\leq i \leq n \text{ times}} q \xrightarrow{\leq i-1 \text{ times}} \dots \xrightarrow{\leq i-1 \text{ times}} p'$. □

A.3.1 Computational complexity of DSL learning

How long does each update to the DSL in Algorithm 1 take? Constructing the version spaces takes time linear in the number of programs (written P) in the frontiers (Algorithm 1, line 5), and, in the worst case, exponential time as a function of the number of refactoring steps n — but we bound the number of steps to be a small number (typically $n = 3$). Writing V for the number of version spaces, this means that V is $O(P2^n)$. The number of proposals (line 10) is linear in the number of distinct version spaces, so is $O(V)$. For each proposal we have to refactor every program (line 6), so this means we spend $O(V^2) = O(P^22^n)$ per DSL update. In practice this quadratic dependence on P (the number of programs) is prohibitively slow. We now describe a linear time approximation to the refactor step in Algorithm 1 based on beam search.

For each version space v we calculate a *beam*, which is a function from a DSL \mathcal{D} to a shortest program in $\llbracket v \rrbracket$ using primitives in \mathcal{D} . Our strategy will be to only maintain the top B shortest programs in the beam; throughout all of the experiments in this paper, we set $B = 10^6$, and in the limit $B \rightarrow \infty$ we recover the exact behavior of REFACTOR. The following recursive equations define how we calculate these beams; the set ‘proposals’ is defined in line 10 of Algorithm 1, and \mathcal{D} is the current DSL:

$$\begin{aligned} \text{beam}_v(\mathcal{D}') &= \begin{cases} \text{if } \mathcal{D}' \in \text{dom}(b_v): & b_v(\mathcal{D}') \\ \text{if } \mathcal{D}' \notin \text{dom}(b_v): & \text{REFACTOR}(v, \mathcal{D}) \end{cases} \\ b_v &= \text{the } B \text{ pairs } (\mathcal{D}' \mapsto p) \text{ in } b'_v \text{ where the syntax tree of } p \text{ is smallest} \\ b'_v(\mathcal{D}') &= \begin{cases} \text{if } \mathcal{D}' \in \text{proposals and } e \in \mathcal{D}' \text{ and } e \in v: & e \\ \text{otherwise if } v \text{ is a primitive or index:} & v \\ \text{otherwise if } v = \lambda b: & \lambda \text{beam}_b(\mathcal{D}') \\ \text{otherwise if } v = (f \ x): & (\text{beam}_f(\mathcal{D}') \text{ beam}_x \mathcal{D}') \\ \text{otherwise if } v = \oplus V: & \arg \min_{e \in \{b'_{v'}(\mathcal{D}') : v' \in V\}} \text{size}(e|\mathcal{D}') \end{cases} \end{aligned}$$

We calculate $\text{beam}_v(\cdot)$ for each version space using dynamic programming. Using a minheap to represent $\text{beam}_v(\cdot)$, this takes time $O(VB \log B)$, replacing the quadratic dependence on V (and therefore the number of programs, P) with a $B \log B$ term, where the parameter B can be chosen freely, but at the cost of a less accurate beam search.

After performing this beam search, we take only the top I proposals as measured by $-\sum_x \min_{p \in \mathcal{F}_x} \text{beam}_{v_p}(\mathcal{D}')$. We set $I = 300$ in all of our experiments, so $I \ll B$. The reason why we only take the top I proposals (rather than take the top B) is because parameter estimation (estimating θ for each proposal) is much more expensive than performing the beam search — so we perform a very wide beam search and then at the very end trim the beam down to only $I = 300$ proposals. Next, we describe our MAP estimator for the continuous parameters (θ) of the DSL.

A.4 Estimating the continuous parameters θ of a DSL

We use an EM algorithm to estimate the continuous parameters of the DSL, i.e. θ . Suppressing dependencies on \mathcal{D} , the EM updates are

$$\theta = \arg \max_{\theta} \log P(\theta) + \sum_x \mathbb{E}_{q_x} [\log \mathbb{P}[p|\theta]] \quad (5)$$

$$q_x(p) \propto \mathbb{P}[x|p] \mathbb{P}[p|\theta] \mathbb{1}[p \in \mathcal{F}_x] \quad (6)$$

In the M step of EM we will update θ by instead maximizing a lower bound on $\log \mathbb{P}[p|\theta]$, making our approach an instance of Generalized EM.

We write $c(e, p)$ to mean the number of times that primitive e was used in program p ; $c(p) = \sum_{e \in \mathcal{D}} c(e, p)$ to mean the total number of primitives used in program p ; $c(\tau, p)$ to mean the number of times that type τ was the input to sample in Algorithm 3 while sampling program p . Jensen’s inequality gives a lower bound on the likelihood:

$$\begin{aligned} & \sum_x \mathbb{E}_{q_x} [\log \mathbb{P}[p|\theta]] = \\ & \sum_{e \in \mathcal{D}} \log \theta_e \sum_x \mathbb{E}_{q_x} [c(e, p_x)] - \sum_{\tau} \mathbb{E}_{q_x} \left[\sum_x c(\tau, p_x) \right] \log \sum_{\substack{e: \tau' \in \mathcal{D} \\ \text{unify}(\tau, \tau')}} \theta_e \\ & = \sum_e C(e) \log \theta_e - \beta \sum_{\tau} \frac{\mathbb{E}_{q_x} [\sum_x c(\tau, p_x)]}{\beta} \log \sum_{\substack{e: \tau' \in \mathcal{D} \\ \text{unify}(\tau, \tau')}} \theta_e \\ & \geq \sum_e C(e) \log \theta_e - \beta \log \sum_{\tau} \frac{\mathbb{E}_{q_x} [\sum_x c(\tau, p_x)]}{\beta} \sum_{\substack{e: \tau' \in \mathcal{D} \\ \text{unify}(\tau, \tau')}} \theta_e \\ & = \sum_e C(e) \log \theta_e - \beta \log \sum_{\tau} \frac{R(\tau)}{\beta} \sum_{\substack{e: \tau' \in \mathcal{D} \\ \text{unify}(\tau, \tau')}} \theta_e \end{aligned}$$

where we have defined

$$\begin{aligned} C(e) &\triangleq \sum_x \mathbb{E}_{q_x} [c(e, p_x)] \\ R(\tau) &\triangleq \mathbb{E}_{q_x} \left[\sum_x c(\tau, p_x) \right] \\ \beta &\triangleq \sum_{\tau} \mathbb{E}_{q_x} \left[\sum_x c(\tau, p_x) \right] \end{aligned}$$

Crucially it was defining β that let us use Jensen’s inequality. Recalling from the main paper that $P(\theta) \triangleq \text{Dir}(\alpha)$, we have the following lower bound on M-step objective:

$$\sum_e (C(e) + \alpha) \log \theta_e - \beta \log \sum_{\tau} \frac{R(\tau)}{\beta} \sum_{\substack{e: \tau' \in \mathcal{D} \\ \text{unify}(\tau, \tau')}} \theta_e \quad (7)$$

Differentiate with respect to θ_e , where $e : \tau$, and set to zero to obtain:

$$\frac{C(e) + \alpha}{\theta_e} \propto \sum_{\tau'} \mathbb{1}[\text{unify}(\tau, \tau')] R(\tau') \quad (8)$$

$$\theta_e \propto \frac{C(e) + \alpha}{\sum_{\tau'} \mathbb{1}[\text{unify}(\tau, \tau')] R(\tau')} \quad (9)$$

The above is our estimator for θ_e . The above estimator has an intuitive interpretation. The quantity $C(e)$ is the expected number of times that we used e . The quantity $\sum_{\tau'} \mathbb{1}[\text{unify}(\tau, \tau')] R(\tau')$ is the expected number of times that we *could have* used e . The hyperparameter α acts as pseudocounts that are added to the number of times that we used each primitive, and are not added to the number of times that we could have used each primitive.

We are only maximizing a lower bound on the log posterior; when is this lower bound tight? This lower bound is tight whenever all of the types of the expressions in the DSL are not polymorphic, in which case our DSL is equivalent to a PCFG and this estimator is equivalent to the inside/outside algorithm. Polymorphism introduces context-sensitivity to the DSL, and exactly maximizing the likelihood with respect to θ becomes intractable, so for domains with polymorphic types we use this estimator.

A.5 Full set of LOGO tasks

A.6 Full set of tower tasks

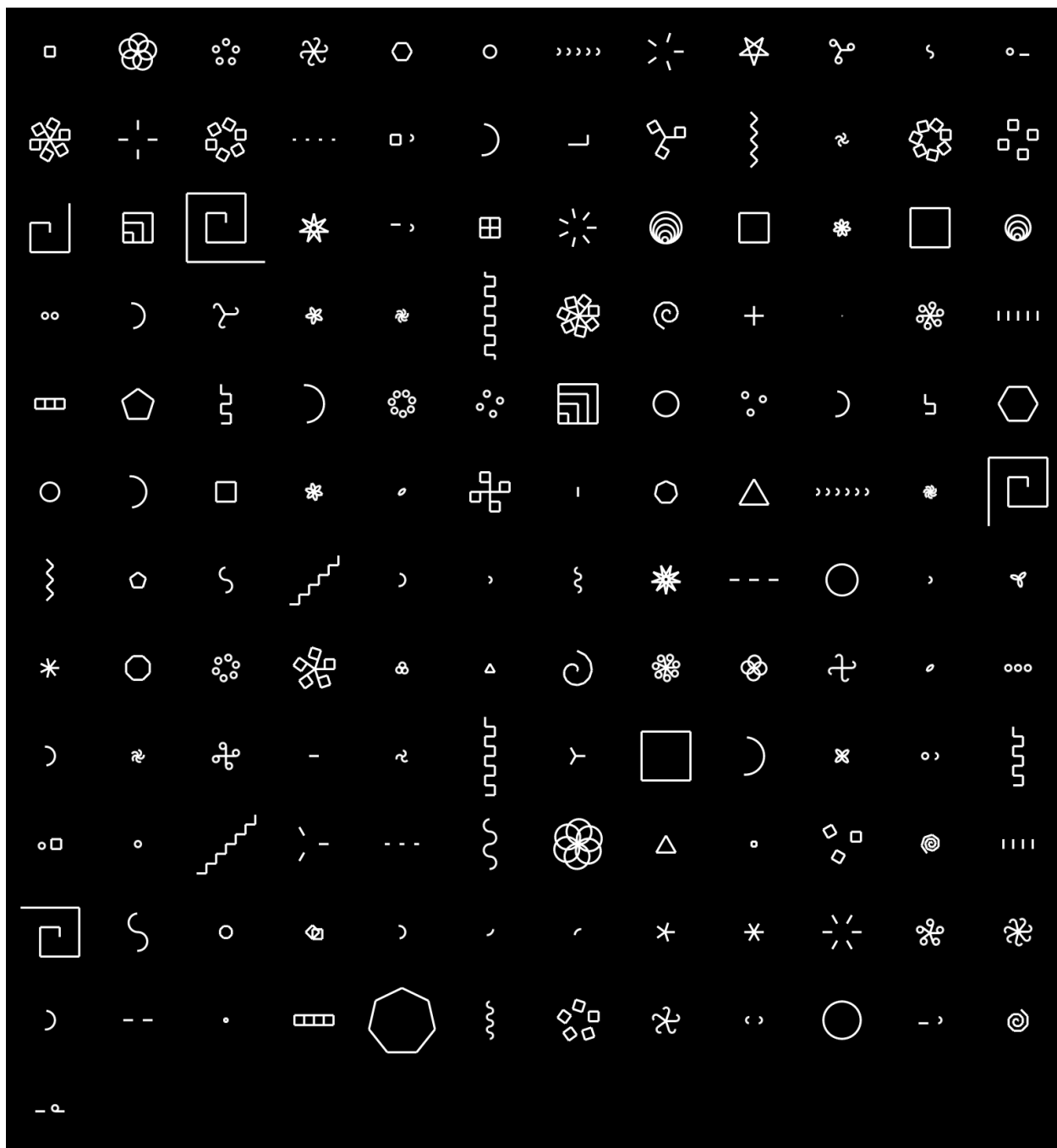


Figure 14: Full set of LOGO graphics tasks that we apply our system to

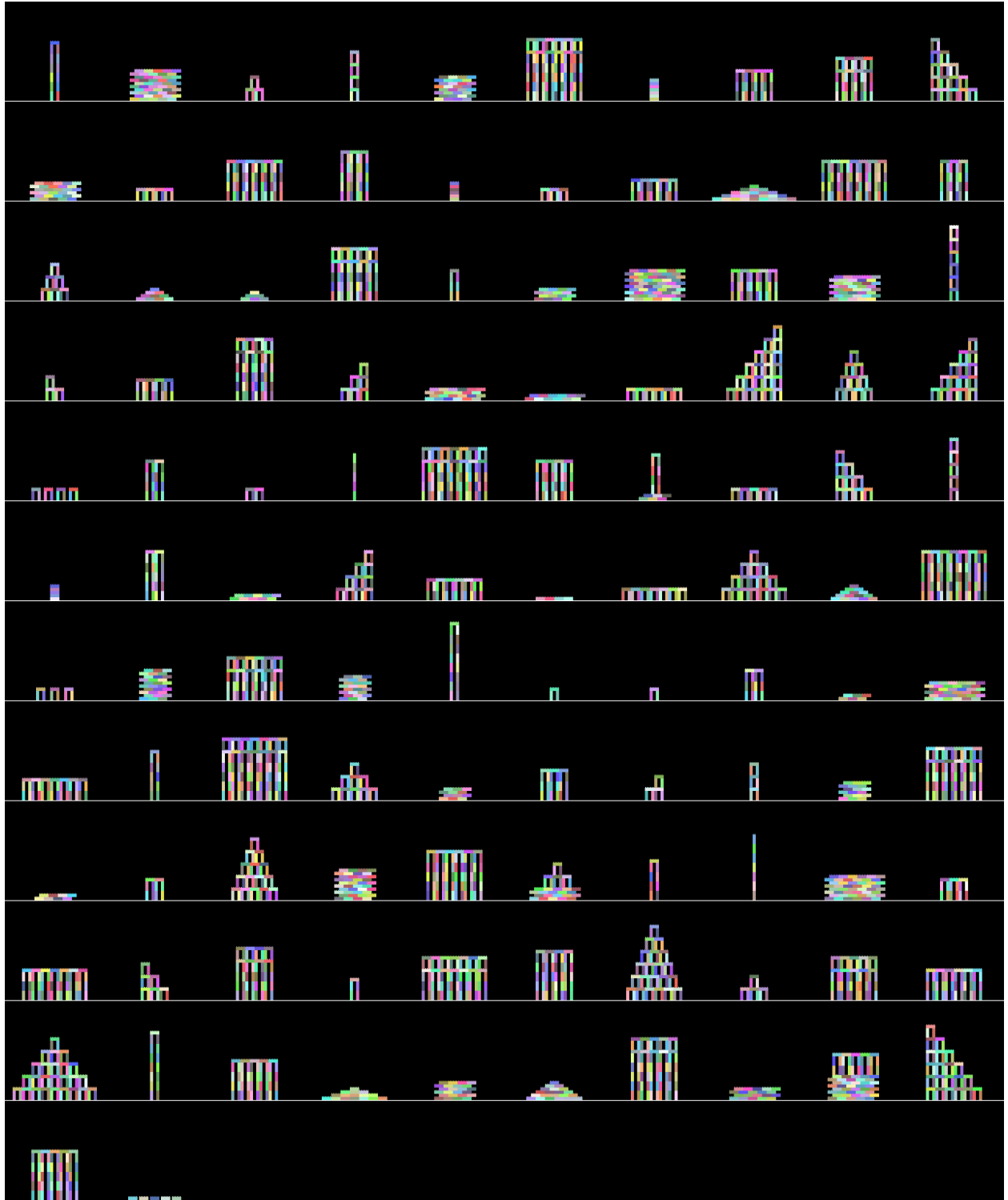


Figure 15: Full set of tower building tasks that we apply our system to