

# 1 Introduction

What would it take to build a machine that can learn everything a person does over their lifetime? Although this goal will remain distant for the foreseeable future, we know that a general learning system like this would need to be able to acquire many different kinds of expertise. Virtually every child becomes an expert in natural language, motor control, and social interaction, while learning rich intuitive theories of kinship, taxonomy and physics. Many of those children will grow into adults with competence in cooking, calculus, tennis, drawing pictures, or writing software. Expertise – the ability to quickly solve new problems in a domain – is crucial. Human learning goes far beyond just memorizing a large set of facts or routines, but involves higher-order skills, like learning to learn new concepts or learning to solve new types of problems more quickly and effectively. Despite great advances in machine learning, we are still far from an AI with these abilities. The modern AI toolkit has given us machines that learn to play challenging games at superhuman levels, but cannot quickly transfer to similar games like a person would; or, which can learn to generate convincing English prose, but which do not then learn to analyze many different languages, like a field linguist would. Here, we contribute a computational model that takes a small step toward the goal of building machines that grow into domain experts, looking at much simpler kinds of problems that humans can learn to solve and indeed become experts in.

Our model is structured around the hypothesis that domain expertise consists primarily of two ingredients. First, domain experts have explicit, declarative concepts that are powerful yet finely-tuned. A visual artist has concepts like arcs, spirals, symmetries, and perspectives; a physicist has concepts like dot products, vector fields, and conservation laws; and an architect has concepts like arches, supports, and bridges. Second, experts have implicit, procedural skill in deploying those concepts to quickly solve new problems: at a glance, human domain experts can intuit which compositions of concepts are likely to solve the task at hand, even before searching for a solution. Studies of human expertise find that, compared to novices, experts can quickly categorize and classify problems based on the “deep structure” of the problem’s solution [6, 7]. In short, we take the stance that expertise means both having the right explicit concepts, and being able to quickly see how those concepts can be composed into a solution.

Because solutions to many kinds of problems can often be described as some kind of program [24, 46, 38], our model solves a specific problem by synthesizing a program solving it. It gradually grows its explicit domain-specific knowledge for a class of problems by assembling a library of code containing concepts useful for the domain, and by training a neural network to quickly infer, for a specific problem, what code in its library is likely to solve it. We think of these two learned components as analogous to the explicit concepts and implicit procedural skill that human experts develop. The model is an instance of what in the machine learning community is called a wake-sleep algorithm, where for us, ‘waking’ corresponds to solving problems, while a pair of ‘sleep’ cycles correspond to improving the library of explicit concepts and training the neural net to improve implicit procedural knowledge of how to write code.

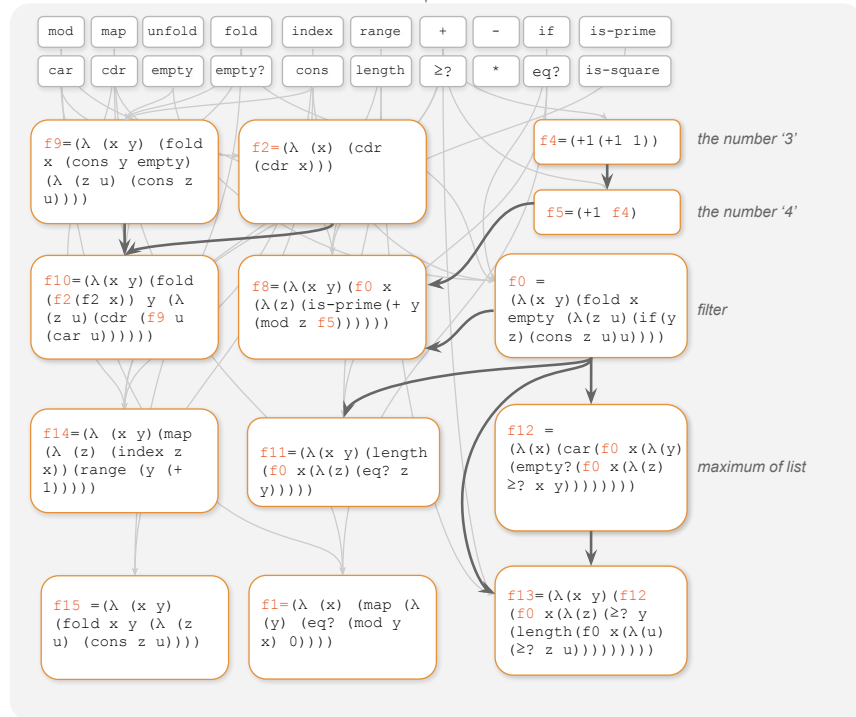
We will first investigate our model within classic program synthesis domains for manipulating sequences of numbers and text, and then consider visual and creative programs for drawing pictures and building towers out of toy blocks, and finally consider programs for basic kinds of equation discovery and programming language design, altogether considering both deterministic and probabilistic programs that act both generatively (e.g., producing an artifact like an image or plan) and conditionally (e.g., mapping inputs to outputs).

Our model iteratively creates new library routines that build on concepts learned earlier in its learning trajectory, growing a library with nested hierarchies of code. We think of this cumulative nesting of abstractions as a variety of deep representation learning [27]. Figure 1 diagrams a subset of these learned networks (the library). For example, the model learns to sort sequences of numbers by invoking a library component 4 layers deep, or draws the leftmost pairs of images in Figure 1 using a depth-3 component. For this reason we refer to our approach as an instance of ‘deep program learning’.

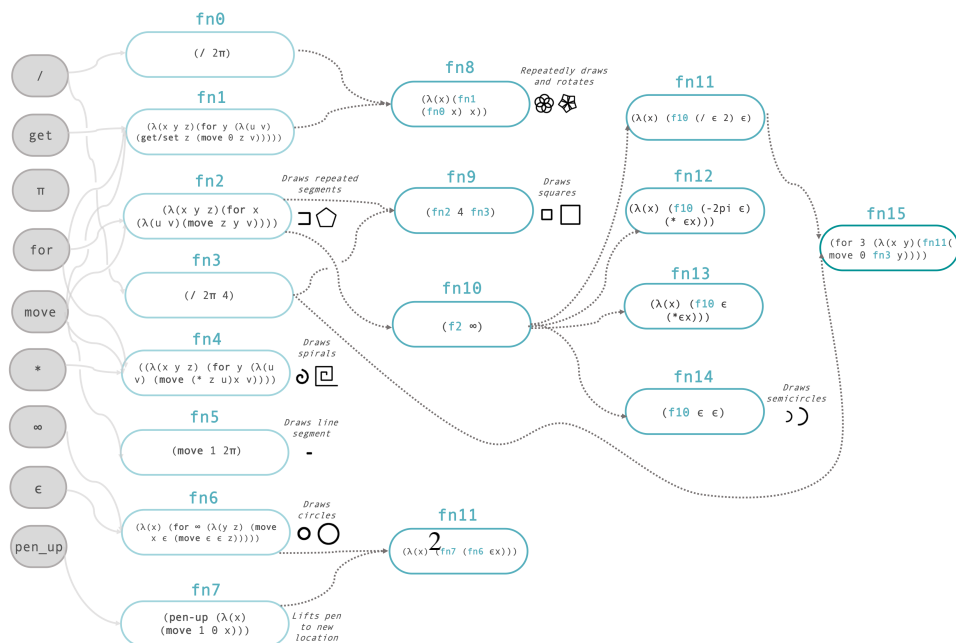
## Sort

$[9 \ 2 \ 3] \rightarrow [2 \ 3 \ 9]$   
 $[8 \ 7 \ 9 \ 2 \ 5] \rightarrow [2 \ 5 \ 7 \ 8 \ 9]$   
 $[9 \ 8 \ 9] \rightarrow [8 \ 9 \ 9]$

It invents a DSL of new complex functions to **filter**, take the **maximum**, and others that **build on its primitives** and **learned functions**.



```
(λ (map (λ (f13 (x y) z (+ u
1))) (range (length u))))
```



## 2 Deep Wake/Sleep Program Induction

Our model, called DREAMCODER, works through a novel kind of wake/sleep or ‘dream’ learning [18], iterating through a wake cycle – where it solves **tasks** by writing programs (Figure 2 top) – and a pair of sleep cycles, both of which are loosely biologically inspired. The first sleep cycle, which we refer to as **consolidation**, grows its library of code by replying experiences from waking and consolidating them into new code abstractions (Figure 2 left). This cycle is inspired by the formation of abstractions during sleep memory consolidation [12]. The second sleep cycle, which we refer to as **dreaming**, improves the agents knowledge of how to write code by training a neural network to help quickly search for programs. The neural net is trained on replayed experiences as well as ‘fantasies’, or samples, from the learned library (Figure 2 right). These two kinds of dreams are inspired by the distinct episodic replay and hallucination components of dream sleep [15].

Viewed as a probabilistic inference problem, DREAMCODER observes a set of tasks, written  $X$ , and infers both a program solving each task, as well as a prior distribution over programs likely to solve tasks in the domain (Figure 2 middle). This prior is encoded by a library, written  $\mathcal{D}$ , which combined with a learned weight vector  $\theta$  defines a generative model over programs (Appendix 2). The neural network learns to invert this generative model by predicting, conditioned on a task, a “posterior” distribution over programs likely to solve that specific task, thus functioning as a **recognition model** and trained jointly with the generative model, in the spirit of the Helmholtz machine [18]. Writing  $Q(p|x)$  for the approximate posterior predicted by the recognition model, wake/sleep cycles correspond to iteratively (and approximately) solving for

$$p_x = \arg \max_p \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}^*, \theta^*], \text{ for each task } x \in X \quad \text{Wake}$$

$$\begin{aligned} \mathcal{D}^* &= \arg \max_{\mathcal{D}} \int \mathbb{P}[\mathcal{D}, \theta] \prod_{x \in X} \sum_{p \text{ found during waking}} \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta] d\theta \\ \theta^* &= \arg \max_{\theta} \mathbb{P}[\mathcal{D}^*, \theta] \prod_{x \in X} \sum_{p \text{ found during waking}} \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}^*, \theta] \end{aligned} \quad \text{Consolidation sleep}$$

$$Q(p|x) \approx \mathbb{P}[p|x, \mathcal{D}^*, \theta^*] \quad \text{Dream sleep}$$

which serves to maximize a lower bound on the posterior over  $(\mathcal{D}, \theta)$  given  $X$  (Appendix A.1).

This 3-phase inference procedure works through two distinct kinds of bootstrapping. During each sleep cycle the next library bootstraps off the concepts learned during earlier cycles, growing an increasingly deep learned program representation. In tandem the generative and recognition models bootstrap each other: a more finely tuned library of concepts yields richer dreams for the recognition model to learn from, while a more accurate recognition model solves more tasks during waking which then feed into the next library.

Waking consists of searching for task-specific programs with high posterior probability, or programs which are a priori likely and which solve a task. During a Wake cycle we sample a minibatch of tasks and find programs solving specific task by enumerating programs in decreasing order of their probability under the recognition model, then checking if a program  $p$  assigns positive probability to a task ( $\mathbb{P}[x|p] > 0$ ). We represent programs as polymorphically typed  $\lambda$ -calculus expressions, an expressive formalism including conditionals, variables, higher-order recursive functions, and the ability to define new functions.

## 3 Related work

We build on several generations of research in AI, program synthesis, and cognitive science, with program induction being one of the oldest theoretical frameworks for concept learning within artificial intelligence [41], and the conceptually allied ‘Language of Thought Hypothesis’ being almost as old [14]. Recent neural program synthesis systems pair a fixed domain-specific programming language (a ‘DSL’) to a learned neural network that guides program search [?, 3, 10], while recent symbolic AI research has developed frameworks for learning the DSL by inferring reusable pieces of code [13, 9, 28, 29]. A main goal of this work is to combine and refine these techniques with the intention of building agents that, like humans, develop domain expertise for new classes of problems. Work in cognitive modeling has often exploited program-like representations for both concept [33, 24, 16] and intuitive theory learning [21, 46], and we view these models of concept learning as a kind of program induction, and see theory learning as similar to inducing a DSL or library.

Deep learning is a dominant machine intelligence paradigm that has had many successes at creating machines with impressive expertise at specific tasks in vision, natural language, gaming, and many other areas. ideas: we want to become an expert from modest amounts of data; we want to be able to transfer to different types of problems, and to do

this zero shot;

### 3.1 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 refactoring programs, where this refactoring exposes common reused subexpressions across the programs found during waking. Second, we use this automatic refactoring 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  $\int \mathbb{P}[\mathcal{D}, \theta] \mathbb{P}[X|\mathcal{D}, \theta] d\theta$  (Sec. 2). We replace this marginal with an AIC approximation [1] and marginalize over refactorings of programs found during waking, minimizing the following expression, which can be interpreted as a kind of compression:

$$\underbrace{-\log \mathbb{P}[\mathcal{D}] + \min_{\theta} \left( -\log \mathbb{P}[\theta|\mathcal{D}] + \|\theta\|_0 + \sum_{x \in X} -\log \sum_{\substack{p \text{ a refactoring of } p' \\ p' \text{ found during waking}}} \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta] \right)}_{\text{Description length of } (\mathcal{D}, \theta)} \quad (1)$$

Description length of programs for task  $x$

But a program has infinitely many possible refactorings, rendering Eq. 1 intractable. Rather than consider every refactoring we bound the number of  $\lambda$ -calculus evaluation steps separating a program from its refactoring. Now the number of refactorings is finite but astronomically large: Figure 3A diagrams a problem where the agent rediscovers the higher-order function `map` starting from the basics of Lisp and the Y-combinator, but where 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?

To resolve this exponential growth we introduce a new data structure combining ideas from version space algebras [25, 32, 35] and equivalence graphs [44]. 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, while equivalence graphs are data structures that track semantic equivalences between program subexpressions. In Appendix A.5.1, we give a dynamic program that takes as input a program and then outputs a version space containing its refactorings while tracking semantically equivalent subexpressions. Figure 3B diagrams a subtree of a version space containing refactorings of a small program. Our technique is substantially 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. Appendix A.5 specifies how we combine this probabilistic and symbolic machinery to update the DSL. At a high level, our approach is to search locally through the space of DSLs, proposing small changes until Eq. 1 fails to decrease.

### 3.2 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 actual tasks it needs to solve, 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, the recognition model  $Q(p|x)$  should approximate the posterior  $\mathbb{P}[p|\mathcal{D}, \theta, x]$ . 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  $\mathbb{E}[\max_p \max_{\mathbb{P}[\cdot|x, \mathcal{D}, \theta]} \log Q(p|x)]$ , where in both cases the expectation is taken over tasks. Taking this expectation over the empirical distribution of tasks trains  $Q$  on replays; taking it over samples from the generative model trains  $Q$  on fantasies. We define a pair of alternative objectives for the recognition model,  $\mathcal{L}^{\text{posterior}}$  and  $\mathcal{L}^{\text{MAP}}$ , which either train  $Q$  to perform full posterior inference or MAP inference,

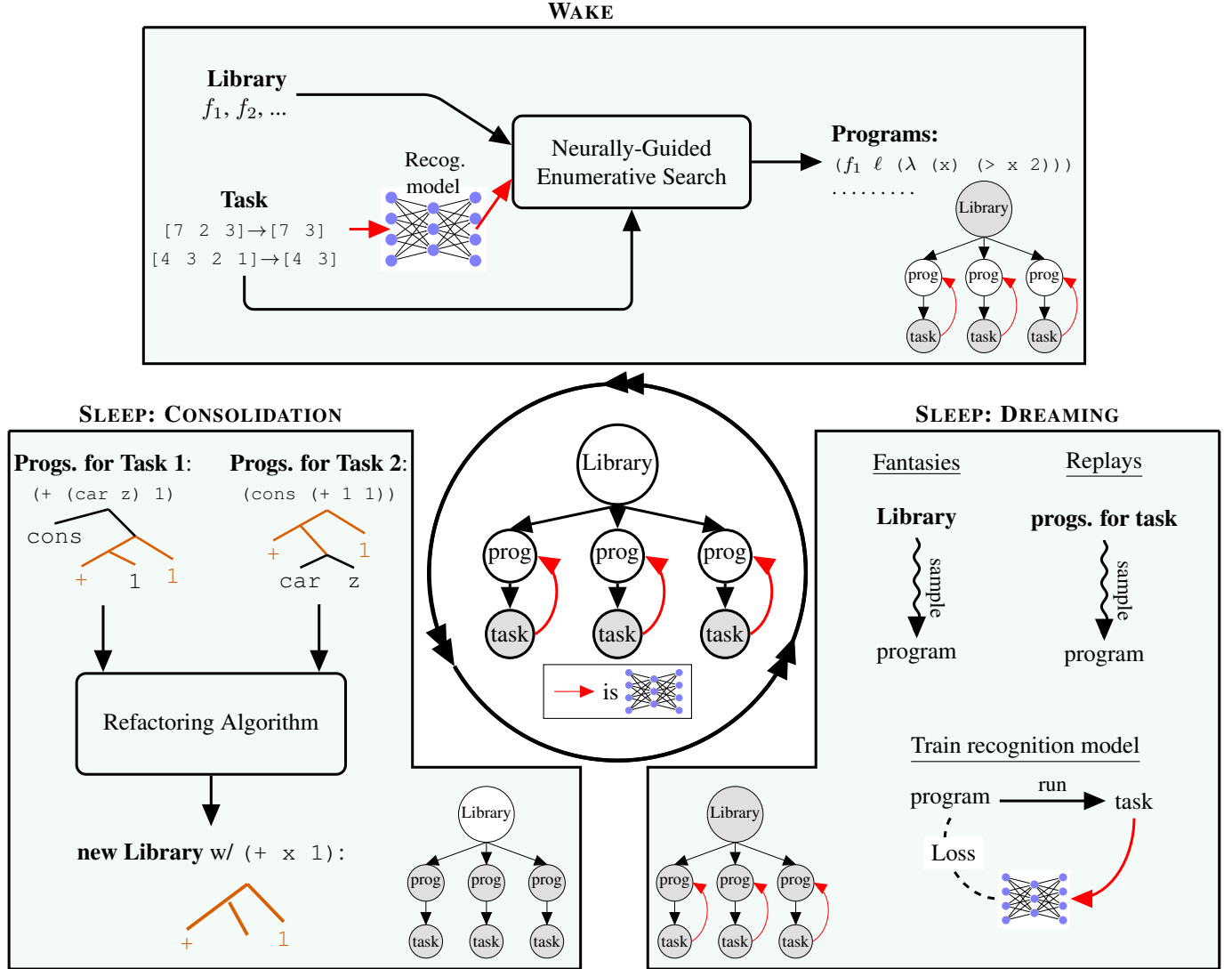
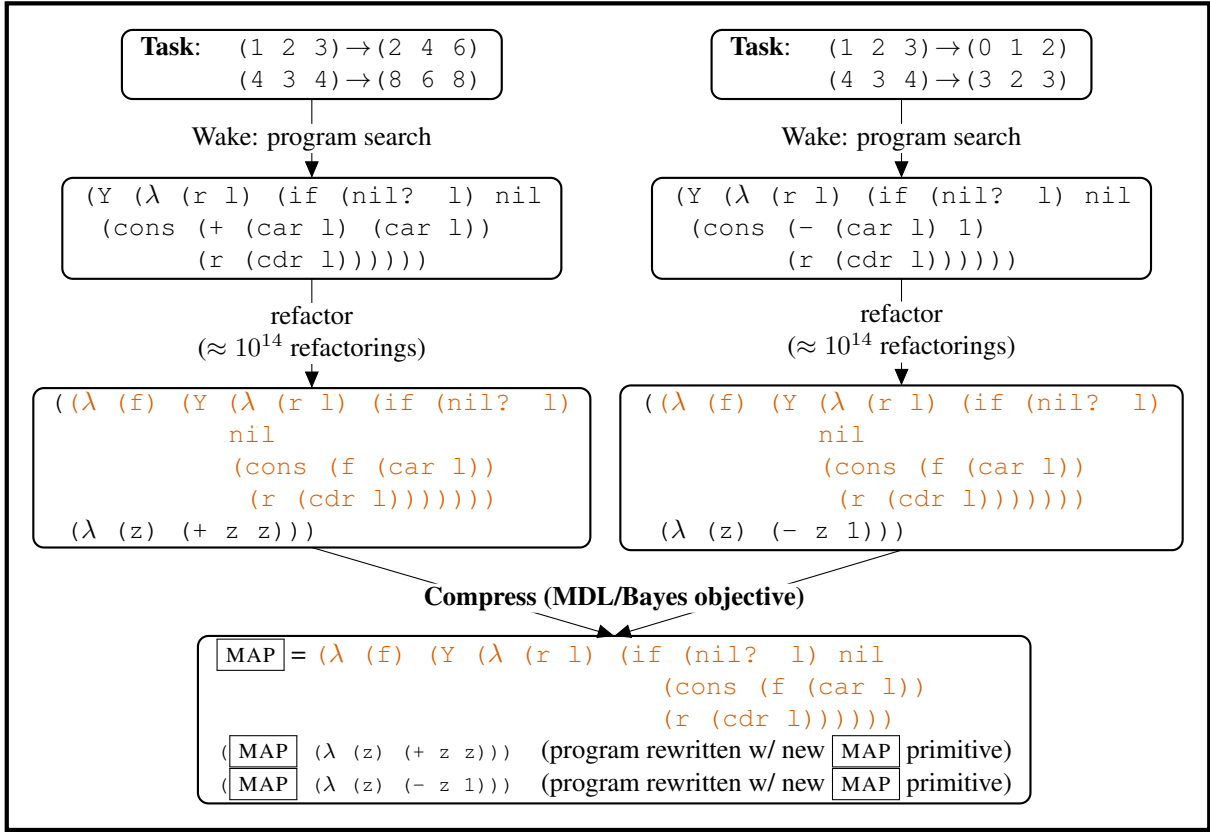


Figure 2: **Middle:** DREAMCODER as a graphical model. Agent observes programming tasks (e.g., input/outputs for list processing or images for graphics programs), which it explains with latent programs, while jointly inferring a latent Domain Specific Language (DSL) capturing cross-program regularities. 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



B

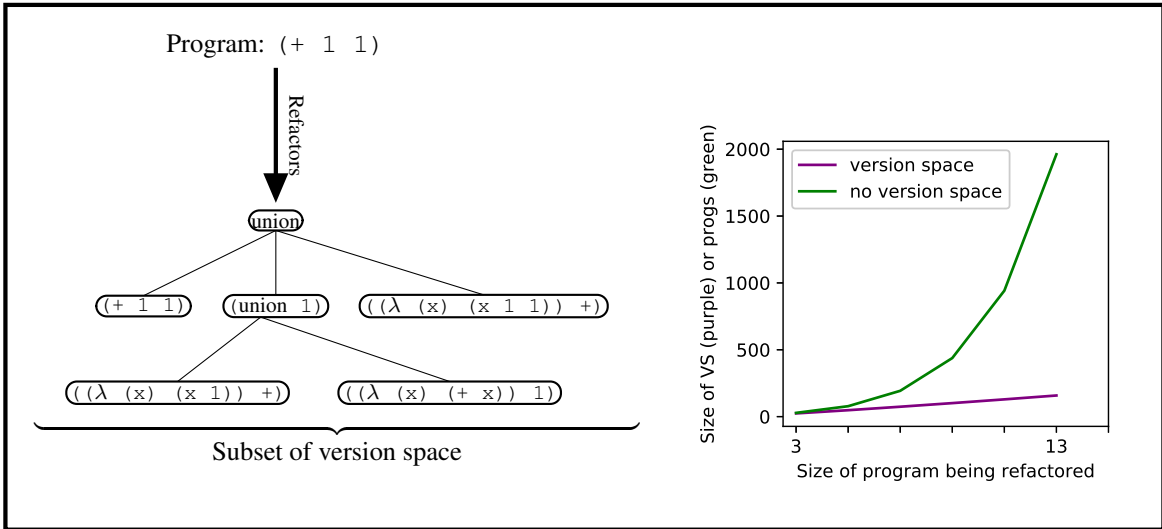


Figure 3: DSL learning as code refactoring. **Panel A:** During waking we discover programs for each task, then refactor the code from those programs to expose common subprograms (highlighted in orange). Common subprograms are incorporated into the DSL when they increase a Bayesian objective. Intuitively, these new DSL components best compress the programs found during waking. **Panel B:** # of possible refactorings grows exponentially with program size, so 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. In this graph, refactored programs are of the form  $1 + 1 + \dots + 1$ .

respectively. These objectives combine replays and fantasies:

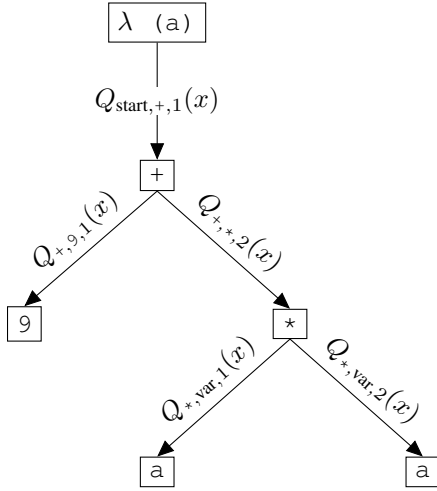
$$\begin{aligned}
\mathcal{L}^{\text{posterior}} &= \mathcal{L}^{\text{posterior}}_{\text{Replay}} + \mathcal{L}^{\text{posterior}}_{\text{Fantasy}} & \mathcal{L}^{\text{MAP}} &= \mathcal{L}^{\text{MAP}}_{\text{Replay}} + \mathcal{L}^{\text{MAP}}_{\text{Fantasy}} \\
\mathcal{L}^{\text{posterior}}_{\text{Replay}} &= \mathbb{E}_{x \sim X} \left[ \sum_{p \in \mathcal{B}_x} \frac{\mathbb{P}[x, p | \mathcal{D}, \theta] \log Q(p|x)}{\sum_{p' \in \mathcal{B}_x} \mathbb{P}[x, p' | \mathcal{D}, \theta]} \right] & \mathcal{L}^{\text{MAP}}_{\text{Replay}} &= \mathbb{E}_{x \sim X} \left[ \max_{\substack{p \in \mathcal{B}_x \\ p \text{ maxing } \mathbb{P}[\cdot | x, \mathcal{D}, \theta]}} \log Q(p|x) \right] \\
\mathcal{L}^{\text{posterior}}_{\text{Fantasy}} &= \mathbb{E}_{(p, x) \sim (\mathcal{D}, \theta)} [\log Q(p|x)] & \mathcal{L}^{\text{MAP}}_{\text{Fantasy}} &= \mathbb{E}_{x \sim (\mathcal{D}, \theta)} \left[ \max_{\substack{p \\ p \text{ maxing } \mathbb{P}[\cdot | x, \mathcal{D}, \theta]}} \log Q(p) \right]
\end{aligned}$$

We maximize  $\mathcal{L}^{\text{MAP}}$  rather than  $\mathcal{L}^{\text{posterior}}$  for two reasons:  $\mathcal{L}^{\text{MAP}}$  prioritizes the shortest program solving a task, thus more strongly accelerating enumerative search during waking; and, combined with our parameterization of  $Q$ , described next, we will show that  $\mathcal{L}^{\text{MAP}}$  forces the recognition model to break symmetries in the space of programs.

**Parameterizing  $Q$ .** The recognition model predicts a fixed-dimensional tensor encoding a distribution over subroutines in the DSL, conditioned on the local context in the syntax tree of the program. This local context consists of the parent node in the syntax tree, as well as which argument is being generated, functioning as a kind of ‘bigram’ model over trees. Figure 4 (left) diagrams this generative process. This parameterization confers three main advantages: (1) it supports fast enumeration and sampling of programs, because the recognition model only runs once per task, like in [3, 13, 31] – thus we can fall back on fast enumeration if the target program is unlike the training programs; (2) the recognition model provides fine-grained information about the structure of the target program, similar to [11, 47]; and (3) in conjunction with  $\mathcal{L}^{\text{MAP}}$  the recognition model learns to break symmetries in the space of programs.

**Symmetry breaking.** 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 arithmetic should disallow adding zero, or force right-associative addition. A bigram parameterization of the recognition model, combined with the  $\mathcal{L}^{\text{MAP}}$  training objective, interact in a way that breaks symmetries like these, allowing the agent to more efficiently explore the space of programs. This interaction occurs because the bigram parameterization can disallow DSL primitives depending on their local syntactic context, while the  $\mathcal{L}^{\text{MAP}}$  objective forces all probability mass onto a single member of a set of syntactically distinct but semantically equivalent expressions (Appendix A.6). 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/unigram<sup>1</sup> parameterization. Figure 4 (right) shows the result of training  $Q$  in these four regimes and then sampling programs. On this particular run, the combination of bigrams and  $\mathcal{L}^{\text{MAP}}$  learns to avoid adding zero and associate addition to the right — different random initializations lead to either right or left association.

<sup>1</sup> In 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 [13]



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

Figure 4: **Left:** Bigram 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  $Q_{\text{parent}, \text{child}, \text{argument index}}(x)$ , where  $x$  is a task. **Right:** Agent learns to break symmetries in program space only when using both bigram parameterization and  $\mathcal{L}^{\text{MAP}}$  objective, associating addition to the right and avoiding adding zero. % right-associative calculated by drawing 500 samples from  $Q$ .  $\mathcal{L}^{\text{MAP}}$ /Unigram agent incorrectly learns to never generate programs with 0's, while  $\mathcal{L}^{\text{MAP}}$ /Bigram agent correctly learns that 0 should only be disallowed as an argument of addition. Tasked with building programs from +, 1, and 0.

## 4 Experiments

### 4.1 Programs that manipulate sequences

We first apply DREAMCODER to two classic benchmark domains: list processing and text editing. In both cases we solve tasks specified by a input/output examples, starting with a generic functional programming basis: `foldr`, `unfold`, `if`, `map`, `length`, `index`, `=`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`.

#### 4.1.1 List Processing

We took 218 list manipulation tasks from our previous work [13], each with 15 input/output examples. In solving these tasks, the system composed 16 new DSL components, and discovered multiple higher-order functions. Each round of memory consolidation built on components discovered in earlier sleep cycles — for example the agent first learns the higher-order function `filter`, uses `filter` to learn to take the maximum element of a list, then uses that routine to learn a new component for extracting the  $n^{\text{th}}$  largest element of a list, which it finally uses to solve a task involving sorting a list of numbers (Figure ??). This incremental, modular learning of deep hierarchies of DSL components occurs because of the alternation between code writing (during waking) and code refactoring (during the consolidation phase of sleep).

#### 4.1.2 Text Editing

Synthesizing programs that edit text is a classic problem in the programming languages and AI literatures [25], and algorithms that synthesize text editing programs ship in Microsoft Excel [?]. This prior work uses hand-engineered DSLs and hand-engineered search strategies. Here, we will show that we can jointly learn both these ingredients and surpass the state-of-the-art domain-general program synthesizers on a standard text editing benchmark.

We trained our system on 128 automatically-generated text editing tasks, with 4 input/output examples each. We tested, but did not train, on the 108 text editing problems from the SyGuS [2] program synthesis competition. Before any learning, DREAMCODER solves 3.7% of the problems within 10 minutes with an average search time of 235 seconds. After learning, it solves 79.6%, and does so much faster, solving them in an average of 40 seconds. As of



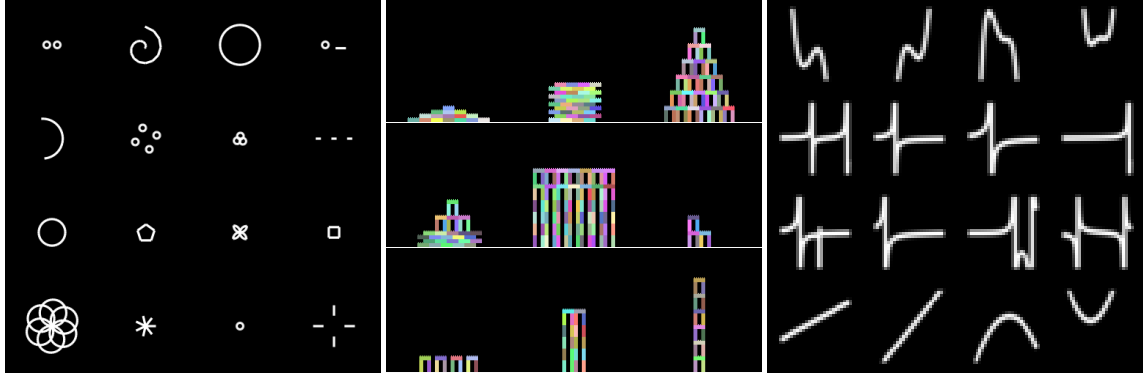


Figure 5: Three domains where the agent infers a program from visual input. **Left:** 16 (out of 160) LOGO graphics tasks. Agent writes a program controlling a ‘pen’ that draws the target picture. **Middle:** 9 (out of 112) tower building tasks. Agent writes a program controlling a ‘hand’ that builds the target tower. **Right:** 16 (out of 200) symbolic regression tasks. Agent writes a program containing continuous real numbers that fits the points along the curve.

the 2017 SyGuS competition, the best-performing synthesizer (CVC4) solves 82.4% of the problems — but here, the competition conditions are 1 hour & 8 CPUs per problem, and with this more generous compute budget we surpass this previous result and solve 84.3% of the problems. SyGuS additionally 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.

## 4.2 Programs from visual input

We consider three domains where the agent must infer a program from an image (Figure 5). First we consider programs that make plans and take actions: drawing pictures and building towers out of blocks (Sec. 4.2.1-4.2.2).

### 4.2.1 Programs that draw pictures

Procedural visual concepts are studied across AI and cognitive science — Bongard problems [5], Raven’s progressive matrices [36], and Lake et al.’s BPL model of omniglot [24] are prominent examples. Here we take inspiration from LOGO Turtle graphics [45], tasking our agent with drawing a modest corpus of images while equipping it with control over a ‘pen’, along with arithmetic operations on angles and distances.

Inside its learned DSL we find interpretable parametric drawing routines corresponding to the families of visual objects in its training data, like polygons, circles, and spirals (Figure 6, left) – without supervision the agent has learned the basic types of objects in its visual world. It additionally learns more abstract visual relationships, like rotational symmetry, which it models by incorporating a higher-order function into its DSL (Figure 6, right).

What does DREAMCODER dream of? Prior to learning samples from the DSL are simple and largely unstructured (Figure 7, left). After training the samples become richly structured (Figure 7, right), compositionally recombining latent building blocks and motifs acquired from the training data. This offers a visual window into how the generative model bootstraps recognition model training: as the DSL grows more finely tuned to the domain, the neural net gets richer and more highly varied training data.

### 4.2.2 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 would build the tower. These towers are built from Lego-style blocks that snap together on a discrete grid. The system starts with the same control flow primitives as with LOGO graphics, and learns parametric ‘options’ for building blocks towers (Figure 8), inferring concepts like arches, staircases, bridges and brick walls.


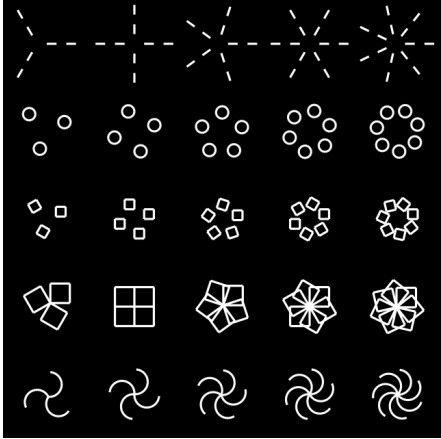
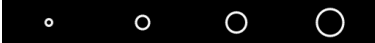




	Parametric drawing routines	Higher-order drawing routine
Semicircle:		
Circles:		
Spiral:		
Greek Spiral:		
S-Curves:		
Polygons & Stars:		

Figure 6: Example primitives learned by DREAMCODER when trained on tasks in Figure ?? . 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.

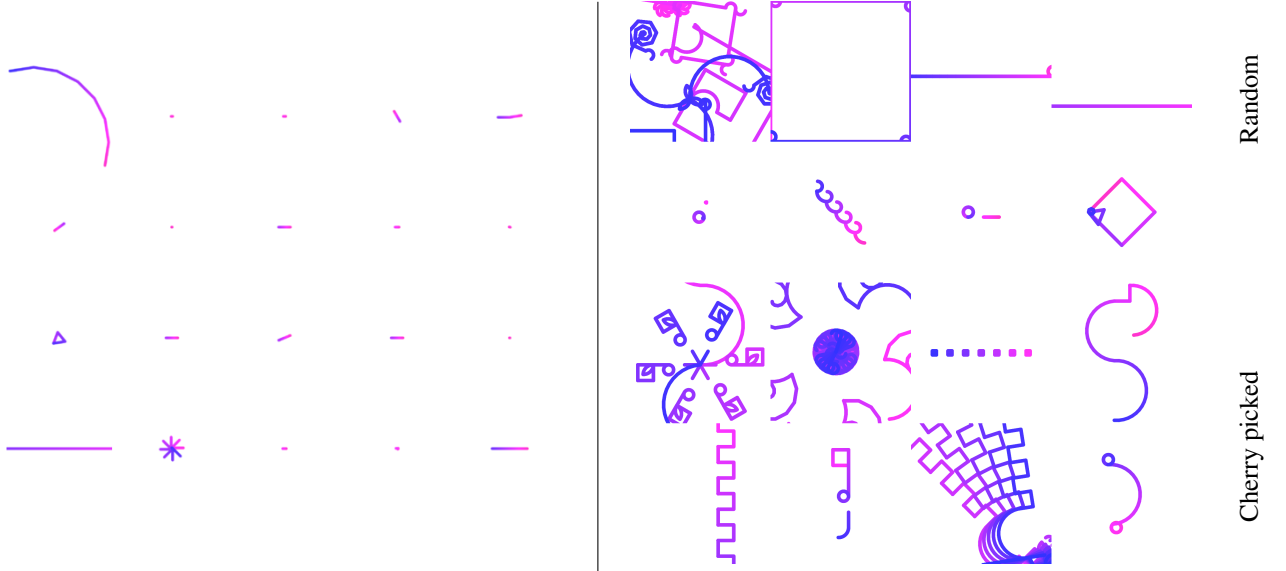


Figure 7: Sixteen dreams, or samples, from the DSL before (left) and after (right) training on tasks in Figure ?? . Blue: where the agent started drawing. Pink: where the agent ended drawing.

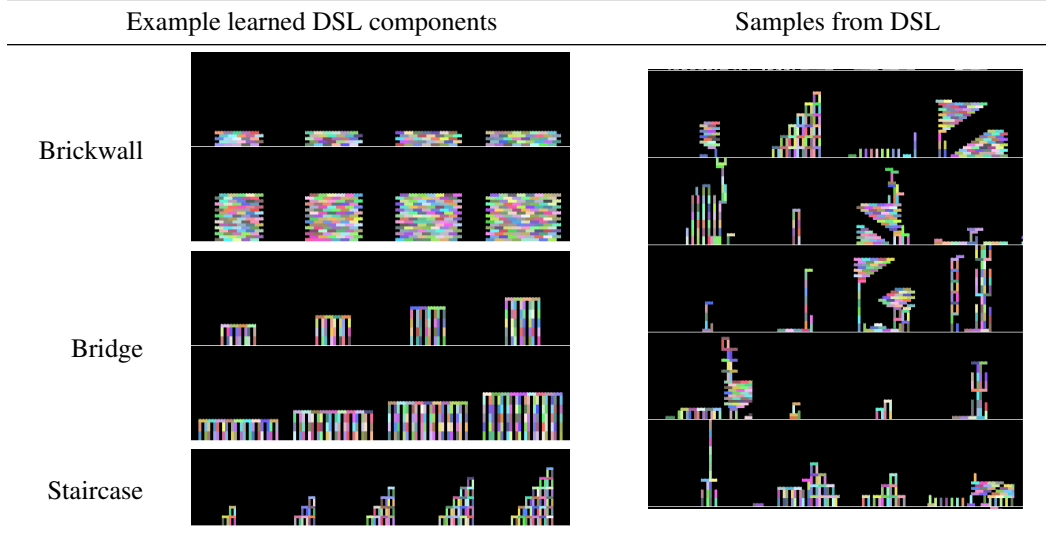


Figure 8: **Left:** Three (out of 19) learned DSL components for building towers out of Lego-style blocks. These components act like parametric options [43], giving higher-level building blocks that the agent can use to plan. **Right:** 16 random samples, or ‘dreams’, from learned DSL.

#### 4.2.3 Symbolic Regression

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 200 symbolic regression 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. 5, rightmost) — 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,  $\mathbb{P}[x|p]$ , by assuming a Gaussian noise model for the input/output examples, and penalize the use of real-valued parameters using the BIC [4].

We learn a DSL containing 13 new functions, mainly templates for different pieces of polynomials or ratios of polynomials. The model also learns to find programs minimizing the number of continuous parameters — for example, learning to represent linear functions with `(* real (+ x real))`. This phenomenon arises from our Bayesian framing: both the generative model’s bias toward shorter programs, and the likelihood model’s BIC penalty.

#### 4.2.4 Generative Modeling of Text

We investigate few-shot learning of generative models by tasking our agent with inferring a probabilistic regular expression from a small number (5) of strings - for example, observing the strings \$1.20 and \$9.42 and predicting the regex \$d

For this task, we used the dataset from [cite Luke], consisting of a large corpus of string concepts. Each string concept has 5 “train” strings and 5 “test” strings.

We trained DREAMCODER using 128 string concepts, randomly sampled from the corpus, and tested on another 128 randomly sampled string concepts.

Our initial DSL consisted of all printable characters, four special character classes (lowercase letters *l*, uppercase letters

*u*, digits

*d*, and `'.'`, which represents all possible characters), and 3 functions: **Kleene Star**, **Maybe**, **Or**. We also included a task-dependent ‘constant’ token, which prints the largest constant string shared across the 5 training tasks. Concatenation was done via continuation-style, as above in the logo domain ... [idk how you wanted this to be mentioned?]

Because this domain has a relatively large number of parameters, we used a modified version of the recognition model, where only unigram parameters were inferred. We found that this recognition model trained quicker and more had more stable convergence.

Probabilistic regexes have no ground truth notion of “solved,” so we evaluate differently from other domains. Our main evaluation metric is to report a lower bound on the posterior predictive of each task given the learned grammar:

$$[x_{test}|x_{train}, \mathcal{D}, \theta] = dp[x_{test}|p][p|x_{train}, \mathcal{D}, \theta] \quad (2)$$

$$= dp[x_{test}|p][x_{train}|p][p|\mathcal{D}, \theta] \quad (3)$$

$$\geq \sum_{p \in F} [x_{test}|p][x_{train}|p][p|\mathcal{D}, \theta] \quad (4)$$

$\theta]$   $[x \rightarrow p] [p \rightarrow \mathcal{D}, \theta]$

where  $x_{train}$  are the training strings for task  $x$ , and  $x_{test}$  are the held-out test strings, not seen during search.

For this domain we use an additional post-processing step, in order to generate realistic samples from the probabilistic programs. For each task and each found probabilistic regex, we separately optimize the continuous regex production parameters in order to maximize the likelihood on the training strings for that task. We apply additive smoothing to ensure optimization does not over-fit to the training tasks.

---

Observed text	Inferred regex	Held out testing text
---------------	----------------	-----------------------

---

### 4.3 Quantitative Results on Held-Out Tasks

To evaluate the relative importance of DSL learning and recognition model training, we evaluate on held-out testing tasks for each of our domains, measuring both how many tasks are solved and how long it takes to solve them across successive wake/sleep iterations (Fig. 9). We always solve more held-out tasks – and generally solve them in less time – with both components combined. Why? One hypothesis is that some tasks are best solved by DSL learning and others by a neural network, and so including both of our sleep cycles takes the union of these sets of tasks. Another hypothesis is that the DSL and neural network interact synergistically, bootstrapping off each other to solve tasks for which neither alone suffice. We evaluate the relative weight of these interactions by computing the ratio of the tasks solved uniquely by the full model to

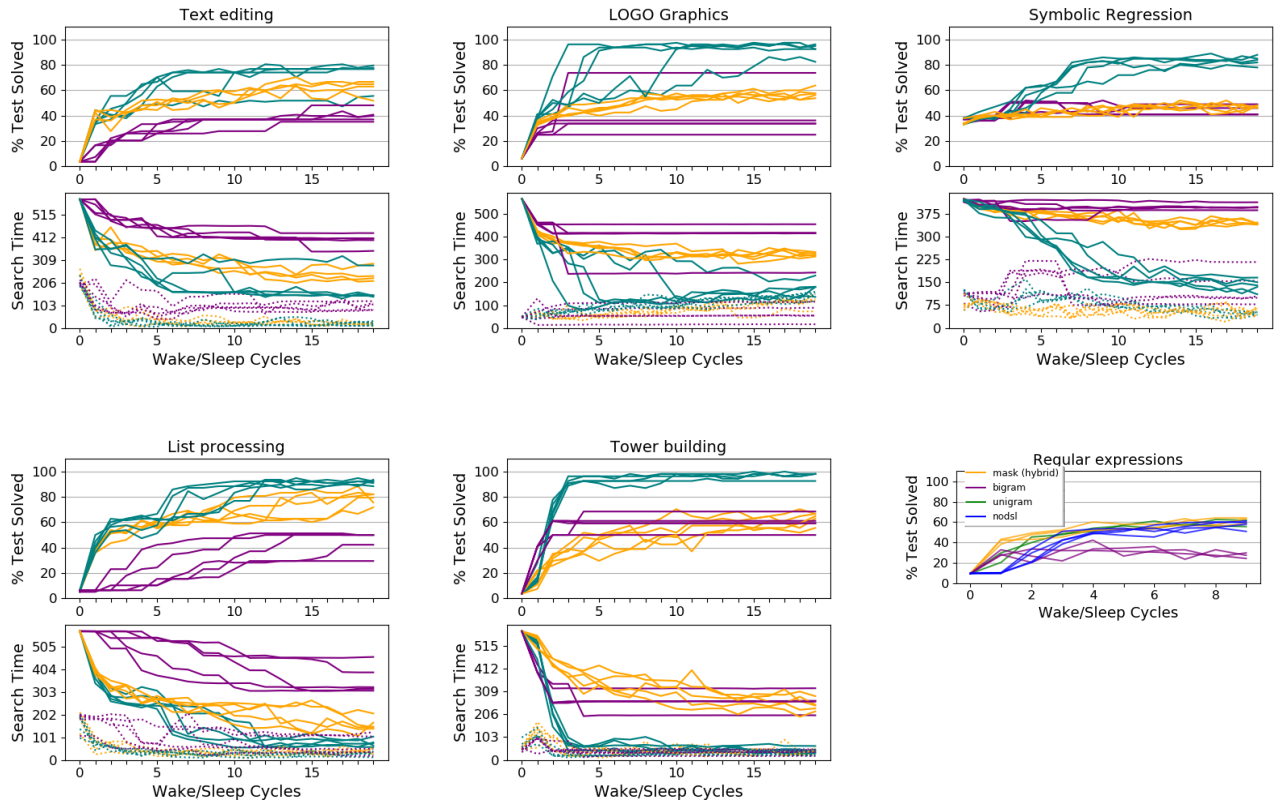


Figure 9: Test set performance across wake/sleep iterations. Each curve is a run with a different random seed. Teal: Full model. Orange: Dreaming only (no DSL learning). Purple: Consolidation only (no recognition model). Search time plots show solid lines (time averaged over all tasks) and dotted lines (time averaged over solved tasks).

## 5 Discussion

(*what we found*) Our goal was to build a model capturing humanlike aspects of learning to be an expert in a new domain of problems. Two main findings of our work are that, first, a single program-induction system can learn to solve large sets of problems from many qualitatively different domains; and second, that fully acquiring expertise in these domains hinges both upon explicit declarative knowledge, and implicit procedural skill. We have deliberately placed the interplay of declarative and procedural knowledge at the center of our model, and believe that both these kinds of learning are crucial for building agents that, like humans, autonomously learn to navigate a new domain of problems.

Our model’s domain knowledge grows over a series of wake/sleep cycles, with the solutions to easier problems bootstrapping solving of harder tasks. This dynamic is related to, but distinct from, curriculum learning. In curriculum learning approaches an agent solves a stream of pedagogically selected, increasingly difficult tasks; here we consider the case with the agent lacks a ‘teacher’ and instead sees randomly sampled batches of tasks, effectively self-pacing its way through an implicit curriculum. But humans select their tasks in much richer ways, and can even generate their own tasks to solve, either as stepping stones toward hard, unsolved problems or motivated by curiosity and aesthetics. Building agents that generate their own problems in these humanlike ways is a necessary next step if we want machines that push against boundary of human domain knowledge or which even discover whole new domains of problems, both of which go far beyond the relatively small-scale demonstrations in this work.

A basic representational challenge is to learn concepts fundamentally inexpressible in terms of the starting basis: DREAMCODER invents procedures for adding vectors, taking dot products, and calculating forces, but can not make the jump needed to define new terms like ‘acceleration’ and ‘coordinate system,’ concepts inexpressible as a library of procedures. We believe approaches based either on term rewriting systems or defining new generalized algebraic datatypes could give traction to fundamental problems like these, strategies which are closely allied with conceptual role semantics. Indeed, humans often open up whole new fields of inquiry by introducing concepts not definable in terms of prior abstractions, like the meaning of ‘allele’ in genetics or ‘force’ in physics.

related work

limitations

how will go beyond? generate own tasks; real DSL learning (all expressible in base language); lifelong learning (forgetting?)

### 5.1 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 [41, 39, 19, 42]. 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 [30]: 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. 4.1) 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 18 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 48 CPUs, our algorithm solves these 18 exercises, along the way assembling a DSL with a modern repertoire of functional programming idioms and subroutines, including `map`, `fold`, `unfold`, `index`, `length`, and arithmetic operations like building lists of natural numbers between an interval (see Appendix A.9).

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.

## 5.2 DREAMCODER and the Exploration-Compression family of algorithms

Our work sits within the Exploration-Compression (EC) family of algorithms. EC [9] is a program induction framework where an agent alternates between searching, or ‘exploring’, the space of programs, and then updating its search procedure by compressing programs found during exploration. DREAMCODER grew directly out of research on EC-style systems and both of our sleep phases can be interpreted as a kind of compression: consolidation aims to compactly refactor code, while the recognition model aims to encode a program in as few bits as possible, conditioned on a task. Our previous work, EC<sup>2</sup> [13], introduced neurally-guided search into the EC framework and served as the starting point for DREAMCODER. Our work here extends EC<sup>2</sup> by (1) introducing a new refactoring compression algorithm, (2) enriching the neural network with a new bigram-over-trees parameterization and loss function that together learn to break symmetries, accelerating program search during waking, and (3) demonstrating how this family of approaches can be applied to planning and generative modeling problems. Other offshoots of EC include the neurosymbolic framework in [26] and the hierarchical Bayesian program learner in [28]. Closely aligned ideas go even further back [42, 39].

## 5.3 Acquiring Domain Expertise

One interpretation of our system is as a model of the acquisition of domain expertise. Humans can acquire expertise across many domains – cooking, coding, music, architecture, painting, tennis, or calculus, to name a handful of examples, and every child develops expertise in natural language, intuitive physics, motor control, kinship relationships, and more. Domain experts learn domain-specific abstractions, similar to a DSL: for example, a expert chef knows what combinations of seasonings go together, or an expert mathematician knows a wide set of useful theorems and lemmas. Jointly, experts learn how to recognize when to use these abstractions to solve problems. Becoming an expert involves a learning trajectory that unfolds over relatively long time scales, but has modest data requirements relative to the dominant machine learning paradigms. For example, basic competency in cooking or coding might require on the order of learning a hundred recipes or solving a hundred programming exercises. It is this learning regime that we have targeted with DREAMCODER: Learning about a domain from at most several hundred tasks, but where the learning unfolds over many wake/sleep cycles.

## 5.4 Prospects for program Induction as part of the generic AI toolkit

Our aim with DREAMCODER is to chart a path by which program induction can become more broadly useful for AI. This means viewing the AI landscape through the lens of program learning, including the terrain considered here — simple kinds of generative modeling, inverse graphics, planning, and programming by example — but also many others like reinforcement learning, commonsense reasoning, natural language understanding, and causal inference. Can program induction rise to the challenge? We believe it can, provided we push jointly along many different axes of AI research; and provided we continue to integrate learning algorithms – both symbolic and neural, both top-down and bottom-up – into our artificial agents.

Our system, with its learned DSL and neural recognition model, is one embodiment of this hybrid symbolic/neural approach, and enjoys some success across small-scale problems in the different domains considered here. Scaling to larger problems, such as inferring 3-D object models (vs LOGO/Turtle), learning natural image grammars (vs ),

## References

- [1] Hirotogu Akaike. Information theory and an extension of the maximum likelihood principle. In *Selected papers of hirotogu akaike*, pages 199–213. Springer, 1998.
- [2] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.
- [3] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *ICLR*, 2016.
- [4] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. 2006.
- [5] M. M. Bongard. *Pattern Recognition*. Spartan Books, 1970.

- [6] Michelene TH Chi, Robert Glaser, and Ernest Rees. Expertise in problem solving. Technical report, PITTSBURGH UNIV PA LEARNING RESEARCH AND DEVELOPMENT CENTER, 1981.
- [7] Michelene TH Chi and Kurt A VanLehn. Seeing deep structure from the interactions of surface features. *Educational Psychologist*, 2012.
- [8] 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.
- [9] Eyal Dechter, Jon Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. Bootstrap learning via modular concept discovery. In *IJCAI*, 2013.
- [10] Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural program meta-induction. In *NIPS*, 2017.
- [11] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. *ICML*, 2017.
- [12] Yadin Dudai, Avi Karni, and Jan Born. The consolidation and transformation of memory. *Neuron*, 88(1):20 – 32, 2015.
- [13] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Library learning for neurally-guided bayesian program induction. In *NeurIPS*, 2018.
- [14] Jerry A Fodor. *The language of thought*, volume 5. Harvard University Press, 1975.
- [15] 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.
- [16] Noah D Goodman, Joshua B Tenenbaum, and T Gerstenberg. *Concepts in a probabilistic language of thought*. MIT Press, 2015.
- [17] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.
- [18] 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.
- [19] Marcus Hutter. *Universal artificial intelligence: Sequential decisions based on algorithmic probability*. Springer Science & Business Media, 2004.
- [20] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, volume 1, pages 215–224. IEEE, 2010.
- [21] Yarden Katz, Noah D. Goodman, Kristian Kersting, Charles Kemp, and Joshua B. Tenenbaum. Modeling semantic cognition as logical dimensionality reduction. In *CogSci*, pages 71–76, 2008.
- [22] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [23] J.D. Lafferty. *A Derivation of the Inside-outside Algorithm from the EM Algorithm*. Research report.
- [24] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.
- [25] Tessa Lau. *Programming by demonstration: a machine learning approach*. PhD thesis, 2001.
- [26] Miguel Lázaro-Gredilla, Dianhuan Lin, J Swaroop Guntupalli, and Dileep George. Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs. *Science Robotics*, 4(26):eaav3150, 2019.



- [27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.
- [28] Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *ICML*, 2010.
- [29] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenenbaum, and Stephen Muggleton. Bias reformulation for one-shot function induction. In *ECAI 2014*, 2014.
- [30] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.
- [31] 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.
- [32] 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.
- [33] Steven Thomas Piantadosi. *Learning and the language of thought*. PhD thesis, Massachusetts Institute of Technology, 2011.
- [34] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
- [35] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.
- [36] Jean Raven et al. Raven progressive matrices. In *Handbook of nonverbal assessment*, pages 223–237. Springer, 2003.
- [37] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.
- [38] Ute Schmid and Emanuel Kitzelmann. Inductive rule learning on the knowledge level. *Cognitive Systems Research*, 12(3-4):237–248, 2011.
- [39] Jürgen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.
- [40] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In *Advances in Neural Information Processing Systems*, 2017.
- [41] Ray J Solomonoff. A formal theory of inductive inference. *Information and control*, 7(1):1–22, 1964.
- [42] Ray J Solomonoff. A system for incremental learning based on algorithmic probability. Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition, 1989.
- [43] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pages 212–223. Springer, 2002.
- [44] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *ACM SIGPLAN Notices*, volume 44, pages 264–276. ACM, 2009.
- [45] David D. Thornburg. Friends of the turtle. *Compute!*, March 1983.
- [46] 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.
- [47] Maksym Zavershynskyi, Alex Skidanov, and Illia Polosukhin. Naps: Natural program synthesis dataset. In *ICML*, 2018.

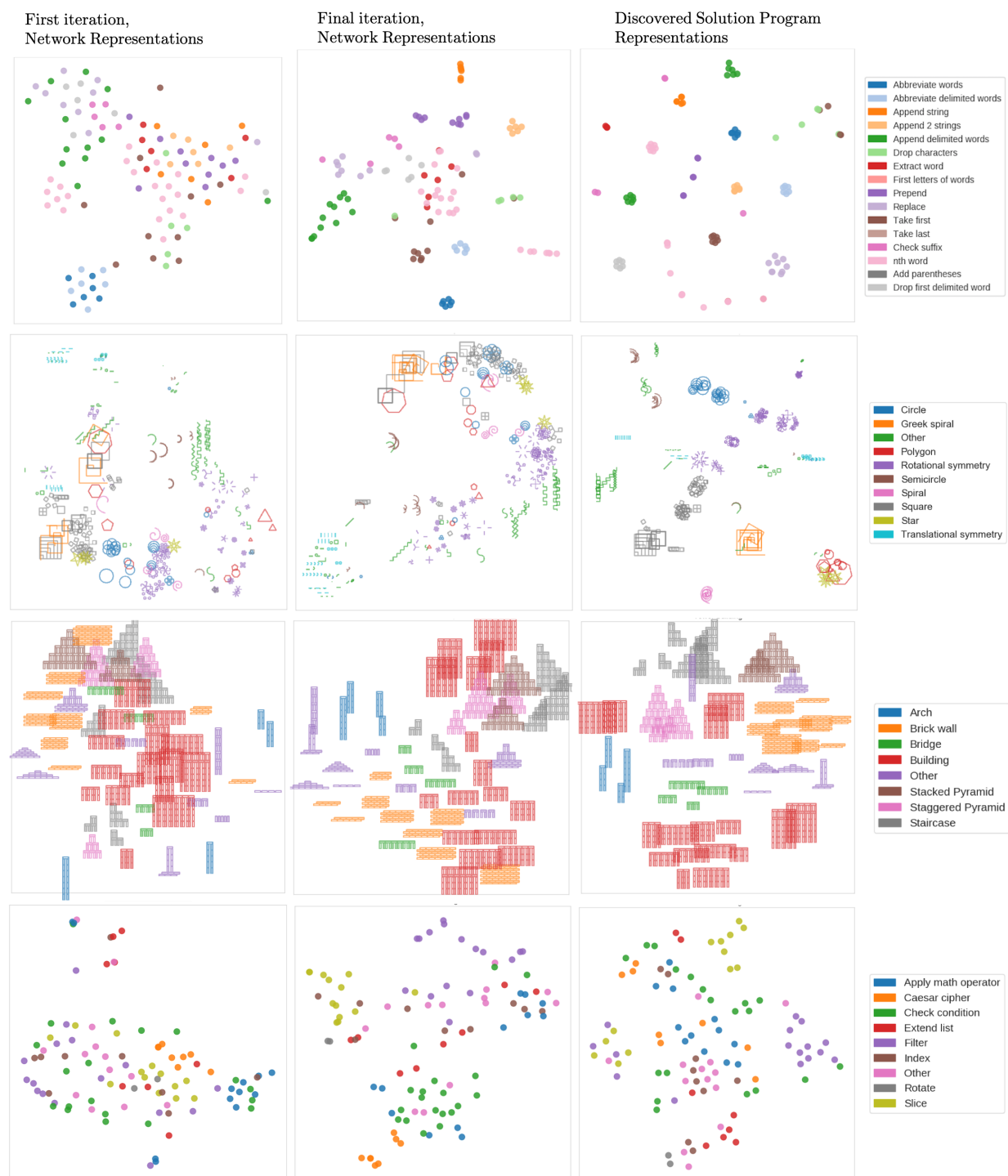


Figure 10

## A Appendix

### A.1 Neural Initial Problem Representations

How do DREAMCODER’s representations of problems themselves change, as the model acquires expertise? Within cognitive science, a rich body of evidence suggests that human experts learn to ‘see’ problems differently in their domains – even before solving a problem, human novices and experts seem to construct fundamentally different initial representations [?]. Novices notice a problems ‘surface features’, the literal objects and details made obvious by the task description. Experts, on the other hand, somehow see past these details to pick out each problems ‘deep structures’, the underlying principles and concepts that govern its ultimate solution [7, ?].

DREAMCODER’s neural recognition model can be seen as forming analogous ‘initial’ problem representations, implicitly encoded within the activations of the neural network. Intuitively, these initial neural representations map problem features to a ‘gist’ over the models current domain-specific concepts – they guide and constrain how the model searches for problem solutions, but are not symbolically structured solutions themselves. To probe how these representations change as the model gains a richer conceptual vocabulary within each domain, we compare how they cluster problems within that domain over the course of DREAMCODER’s learning trajectory. Figure ?? depicts TSNE visualizations of these neural problem representations at the first and last iteration of the algorithm. We color-code each problem using semantic, human-interpretable categories determined apriori by the task designers, but critically, the model itself never has access to these categories. Rather, as DREAMCODER acquires domain expertise, the neural network adapts how it transforms each task’s surface features, restructuring its initial problem representation space to draw together problems that share deeper, more abstract conceptual similarities.

### A.2 Probabilistic Formulation of DREAMCODER

Our objective is to infer the maximum a posteriori DSL  $\mathcal{D}$  and parameters  $\theta$ . Writing  $J$  for the joint probability of  $(\mathcal{D}, \theta)$ , this corresponds to solving

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

$$\mathcal{D}^* = \arg \max_{\mathcal{D}} \int J(\mathcal{D}, \theta) d\theta \quad \theta^* = \arg \max_{\theta} J(\mathcal{D}^*, \theta) \quad (5)$$

where  $\mathbb{P}[x|p]$  scores the likelihood of a task  $x \in X$  given a program  $p$ .<sup>2</sup>

Evaluating Eq. 5 entails marginalizing over the infinite set of all programs – which is impossible. We make a particle-based approximation to Eq. 5 and instead marginalize over a finite **beam** of programs, with one beam per task, collectively written  $\{\mathcal{B}_x\}_{x \in X}$ . This particle-based approximation is written  $\mathcal{L}(\mathcal{D}, \theta, \{\mathcal{B}_x\})$  and acts as a lower bound on the joint density:

$$J(\mathcal{D}, \theta) \geq \mathcal{L}(\mathcal{D}, \theta, \{\mathcal{B}_x\}) \triangleq \mathbb{P}[\mathcal{D}, \theta] \prod_{x \in X} \sum_{p \in \mathcal{B}_x} \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta], \text{ where } |\mathcal{B}_x| \text{ is small} \quad (6)$$

In all of our experiments we set the maximum beam size  $|\mathcal{B}_x|$  to 5.

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

**Wake: Maxing  $\mathcal{L}$  w.r.t. the beams.** Here  $(\mathcal{D}, \theta)$  is fixed and we want to find new programs to add to the beams 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, which is the search objective during waking.

**Sleep (Consolidation): Maxing  $\mathcal{L}$  w.r.t. the DSL.** Here  $\{\mathcal{B}_x\}_{x \in X}$  is held fixed and the problem is to search the discrete space of DSLs and find one maximizing  $\int \mathcal{L} d\theta$ , and then update  $\theta$  to  $\arg \max_{\theta} \mathcal{L}(\mathcal{D}, \theta, \{\mathcal{B}_x\})$ .

Finding programs solving tasks is difficult because of the infinitely large, combinatorial search landscape. We ease this difficulty by training a neural recognition model,  $Q(p|x)$ , 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 beams.** 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 beams will most increase  $\mathcal{L}$ .

<sup>2</sup>For 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.

### A.3 DREAMCODER pseudocode

Algorithm 1 specifies how we integrate wake and sleep cycles.

---

**Algorithm 1** Full DREAMCODER algorithm

---

```

1: function DREAMCODER( $\mathcal{D}, X$ ):
2: Input: Initial DSL  $\mathcal{D}$ , tasks  $X$ 
3: Output: Infinite stream of DSLs, recognition models, and beams
4: Hyperparameters: Batch size  $B$ , enumeration timeout  $T$ , maximum beam size  $F$ 
5:  $\theta \leftarrow$  uniform distribution
6:  $\mathcal{F}_x \leftarrow \emptyset, \forall x \in X$  ▷ Initialize beams to be empty
7: while true do ▷ Loop over epochs
8:   shuffle  $\leftarrow$  random permutation of  $X$  ▷ Randomize minibatches
9:   while shuffle is not empty do ▷ Loop over minibatches
10:    batch  $\leftarrow$  first  $B$  elements of shuffle ▷ Next minibatch of tasks
11:    shuffle  $\leftarrow$  shuffle with first  $B$  elements removed
12:     $\forall x \in$  batch:  $\mathcal{F}_x \leftarrow \mathcal{F}_x \cup \{p \mid p \in \text{enumerate}(\mathbb{P}[\cdot|\mathcal{D}, \theta], T) \text{ if } \mathbb{P}[x|p] > 0\}$  ▷ Wake
13:    Train  $Q(\cdot|\cdot)$  to minimize  $\mathcal{L}^{\text{MAP}}$  across all  $\{\mathcal{F}_x\}_{x \in X}$  ▷ Dream Sleep
14:     $\forall x \in$  batch:  $\mathcal{F}_x \leftarrow \mathcal{F}_x \cup \{p \mid p \in \text{enumerate}(Q(\cdot|x), T) \text{ if } \mathbb{P}[x|p] > 0\}$  ▷ Wake
15:     $\forall x \in$  batch:  $\mathcal{F}_x \leftarrow$  top  $F$  elements of  $\mathcal{F}_x$  as measured by  $\mathbb{P}[\cdot|x, \mathcal{D}, \theta]$  ▷ Keep top  $F$  programs
16:     $\mathcal{D}, \theta, \{\mathcal{F}_x\}_{x \in X} \leftarrow \text{CONSOLIDATE}(\mathcal{D}, \theta, \{\mathcal{F}_x\}_{x \in X})$  ▷ Consolidation Sleep
17:    yield  $(\mathcal{D}, \theta), Q, \{\mathcal{F}_x\}_{x \in X}$  ▷ Yield the updated DSL, recognition model, and solutions found to tasks
18:  end while
19: end while

```

---

### A.4 Generative model over programs

Algorithm 2 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.

### A.5 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  $(\mathcal{D}, \theta)$  or  $Q(p|x)$ . Algorithm 2 and 5 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 2 and 5, 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 3) 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  $\Delta$ , then programs with description length between  $\Delta$  and  $2\Delta$ , then  $2\Delta$  to  $3\Delta$ , etc., until a timeout is reached. This is similar in spirit to iterative deepening depth first search [37].

---

**Algorithm 2** 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  $\leftarrow \{p \mid p : \tau' \in \mathcal{D} \cup \mathcal{E} \text{ if } \tau \text{ can unify with } \text{yield}(\tau')\}$  ▷ Everything in scope w/ type  $\tau$ 
14:  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  $\tau$ 

```

---

---

**Algorithm 3** 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
```

---

## A.6 Details of DSL learning

Algorithm 4 specifies our DSL learning procedure. This integrates two toolkits: the machinery of version spaces and equivalence graphs (Appendix A.5.1) along with a probabilistic objective favoring compressive DSLs. 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 4, lines 5-6, 14; Appendix A.5.1). To define the prior distribution over  $(\mathcal{D}, \theta)$  (Algorithm 4, lines 7-8), we penalize the syntactic complexity of the  $\lambda$ -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  $\lambda$  controls how strongly we regularize the size of the DSL. We place a symmetric Dirichlet prior over the weight vector  $\theta$ .

To appropriately score each proposed  $\mathcal{D}$  we must reestimate the weight vector  $\theta$  (Algorithm 4, 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 [23], our DSLs are context-sensitive due to the presence of variables in the programs and also due to the polymorphic typing system. Appendix A.5.4 derives a tractable MAP estimator for  $\theta$ .

---

### Algorithm 4 DSL Induction Algorithm

---

```

1: Input: Set of beams  $\{\mathcal{B}_x\}$ 
2: Output: DSL  $\mathcal{D}$ , weight vector  $\theta$ 
3:  $\mathcal{D} \leftarrow$  every primitive in  $\{\mathcal{B}_x\}$ 
4: while true do
5:    $\forall p \in \bigcup_x \mathcal{B}_x : v_p \leftarrow I\beta(p)$  ▷ Construct a version space for each program
6:   Define  $L(\mathcal{D}', \theta) = \prod_x \sum_{p \in \mathcal{B}_x} \mathbb{P}[x|p] \mathbb{P}[\text{REFACTOR}(v_p|\mathcal{D}')|\mathcal{D}', \theta]$  ▷ 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  $\text{score}(\mathcal{D}') = \log \mathbb{P}[\mathcal{D}'] + L(\mathcal{D}', \theta^*(\mathcal{D}')) - \|\theta^*(\mathcal{D}')\|_0$  ▷ objective function
9:   components  $\leftarrow \{\text{REFACTOR}(v|\mathcal{D}) : \forall x, \forall p \in \mathcal{B}_x, \forall v \in \text{children}(v_p)\}$  ▷ 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:   $\forall x : \mathcal{B}_x \leftarrow \{\text{REFACTOR}(v_p|\mathcal{D}) : p \in \mathcal{B}_x\}$  ▷ Refactor beams in terms of new DSL
15: end while

```

---

### A.6.1 Refactoring code with version spaces

Formally, a version space is either:

- A deBuijn<sup>3</sup> index: written  $\$i$ , where  $i$  is a natural number
- An abstraction: written  $\lambda v$ , 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  $\lambda$ -calculus expressions,  $\Lambda$

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

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

<sup>3</sup>deBuijn indices are an alternative way of naming variables in  $\lambda$ -calculus. When using deBuijn indices,  $\lambda$ -abstractions are written *without* a variable name, and variables are written as the count of the number of  $\lambda$ -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 [34] for more details.

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:

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

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

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 of a program  $p$ , e.g., the set of all programs  $p'$  where  $p' \xrightarrow{\leq n \text{ times}} q \xrightarrow{\leq n \text{ times}} \dots \xrightarrow{\leq n \text{ times}} p$ , where  $a \xrightarrow{\leq n \text{ times}} b$  is the standard notation for  $a$  rewriting to  $b$  according to the standard rewrite rules of  $\lambda$ -calculus [34].

We define this operator in terms of another operator,  $I\beta'$ , which performs a single step of refactoring:

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

where

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

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

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

$$\downarrow_c^k \$i = \$i, \text{ when } i < c$$

$$\downarrow_c^k \$i = \$(i-k), \text{ when } i \geq c+k$$

$$\downarrow_c^k \$i = \emptyset, \text{ when } c \leq i < c+k$$

$$\downarrow_c^k \lambda b = \lambda \downarrow_{c+1}^k b$$

$$\downarrow_c^k (f \ x) = (\downarrow_c^k f \ \downarrow_c^k x)$$

$$\downarrow_c^k \oplus V = \oplus \{ \downarrow_c^k v \mid v \in V \}$$

$$\downarrow_c^k v = v, \text{ when } v \text{ is a primitive or } \emptyset \text{ or } \Lambda$$



where  $\uparrow^k$  is the shifting operator [34], which adds  $k$  to all of the free variables in a  $\lambda$ -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  $\lambda a$ , 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  $S_n(u) \subset \{v_f \cap v_x \mapsto (b_f \ b_x) \mid (v_f \mapsto b_f) \in S_n(f), (v_x \mapsto b_x) \in S_n(x)\}$ . 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  $\Lambda$  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  $\lambda a$  then, because of the assumption that  $b' \neq \$n$ , we know that  $b'$  is of the form  $\lambda c'$  and that  $\lambda \uparrow_{n+1}^{-1} [\$n \mapsto \uparrow^{1+n} v'] c' \in \lambda a$ . 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  $b' \in v$ . 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  $\Lambda$  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  $\beta$ -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  $\Lambda$ : 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  $p \underbrace{\longrightarrow q \longrightarrow \dots \longrightarrow p'}_{\leq n \text{ times}}$  if and only if  $p \in \llbracket I\beta_n(p') \rrbracket$ .

*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  $p \underbrace{\longrightarrow q \longrightarrow \dots \longrightarrow p'}_{\leq n \text{ times}}$  then  $q \underbrace{\longrightarrow \dots \longrightarrow p'}_{\leq n-1 \text{ times}}$ ; induction on  $n$  gives  $q \in \llbracket I\beta_{n-1}(p') \rrbracket$ . Combined with  $p \longrightarrow q$  we can invoke the completeness of  $I\beta'$  to get  $p \in \llbracket I\beta'(I\beta_{n-1}(p')) \rrbracket \subseteq \llbracket I\beta_n(p') \rrbracket$ .

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  $p \longrightarrow q$  for a program  $q \in \llbracket I\beta'(I\beta'(\dots p')) \rrbracket \subseteq \llbracket I\beta_{i-1}(p') \rrbracket$ . By induction  $q \underbrace{\longrightarrow \dots \longrightarrow p'}_{\leq i-1 \text{ times}}$ , which combined with  $p \longrightarrow q$  gives  $p \underbrace{\longrightarrow q \longrightarrow \dots \longrightarrow p'}_{\leq i \leq n \text{ times}}$ . □

## A.6.2 Tracking equivalences

### A.6.3 Computational complexity of DSL learning

How long does each update to the DSL in Algorithm 4 take? Constructing the version spaces takes time linear in the number of programs (written  $P$ ) in the beams (Algorithm 4, 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  $O(P^{2^n})$ . 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  $O(V^2) = O(P^2 2^n)$  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 4 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 4, 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  $\theta$  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 ( $\theta$ ) of the DSL.

#### A.6.4 Estimating the continuous parameters $\theta$ of a DSL

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

$$\theta = \arg \max_{\theta} \log P(\theta) + \sum_x \mathbb{E}_{q_x} [\log \mathbb{P}[p|\theta]] \quad (7)$$

$$q_x(p) \propto \mathbb{P}[x|p] \mathbb{P}[p|\theta] \mathbb{1}[p \in \mathcal{F}_x] \quad (8)$$

In the M step of EM we will update  $\theta$  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  $\tau$  was the input to sample in Algorithm 2 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  $\beta$  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:

$$\sum_e (C(e) + \alpha) \log \theta_e - \beta \log \sum_{\tau} \frac{R(\tau)}{\beta} \sum_{\substack{e: \tau' \in \mathcal{D} \\ \text{unify}(\tau, \tau')}} \theta_e \quad (9)$$

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

$$\frac{C(e) + \alpha}{\theta_e} \propto \sum_{\tau'} \mathbb{1} [\text{unify}(\tau, \tau')] R(\tau') \quad (10)$$

$$\theta_e \propto \frac{C(e) + \alpha}{\sum_{\tau'} \mathbb{1} [\text{unify}(\tau, \tau')] R(\tau')} \quad (11)$$

The above is our estimator for  $\theta_e$ . 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  $\alpha$  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  $\theta$  becomes intractable, so for domains with polymorphic types we use this estimator.

## A.7 Recognition model training

Recall that our goal is to maximize either  $\mathcal{L}^{\text{posterior}}$  or  $\mathcal{L}^{\text{MAP}}$ , defined as:

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

The fantasy 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  $(\mathcal{D}, \theta)$ , we can draw unlimited samples from  $(\mathcal{D}, \theta)$  and train  $Q$  on those samples. But, evaluating  $\mathcal{L}_{\text{Fantasy}}$  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  $\mathcal{L}_{\text{Fantasy}}^{\text{MAP}}$  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  $\mathcal{L}_{\text{Fantasy}}^{\text{MAP}}$ , we *enumerate* programs in decreasing order of their prior probability, tracking, for each dreamed task  $x$ , the set of enumerated programs maximizing  $\mathbb{P}[x, p | \mathcal{D}, \theta]$ .

We parameterize  $Q$  using a bigram model over syntax trees. Formally,  $Q$  predicts a  $(|\mathcal{D}| + 2) \times (|\mathcal{D}| + 1) \times A$ -dimensional tensor, where  $A$  is the maximum arity<sup>4</sup> 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  $k^{\text{th}}$  argument. Algorithm 5 specifies a procedure for drawing samples from  $Q(\cdot|X)$ .

---

**Algorithm 5** Drawing from distribution over programs predicted by recognition model. Compare w/ Algorithm 2

---

```

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:     $\text{body} \sim$  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:      $\text{primitives} \leftarrow \{p | p : \tau' \in \mathcal{D} \cup \mathcal{E} \text{ if } \tau \text{ can unify with } \text{yield}(\tau')\}$       ▷ Everything in scope w/ type  $\tau$ 
14:      $\text{variables} \leftarrow \{p | p \in \text{primitives and } p \text{ a variable}\}$ 
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:        $a_k \sim$  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

```

---

**Symmetry breaking.** Why does the combination of  $\mathcal{L}^{\text{MAP}}$  and the bigram parameterization lead to symmetry breaking? The reason is twofold: (1) the objective  $\mathcal{L}^{\text{MAP}}$  prefers symmetry breaking recognition models; and (2) the bigram parameterization permits certain kinds of symmetry breaking. To sharpen these intuitions, we prove (Theorem 6) that any global optimizer of  $\mathcal{L}^{\text{MAP}}$  breaks symmetries, and then give a concrete worked out example contrasting the behavior of  $\mathcal{L}^{\text{MAP}}$  and  $\mathcal{L}^{\text{posterior}}$ .

---

<sup>4</sup>The arity of a function is the number of arguments that it takes as input.

**Theorem 6.** Let  $\mu(\cdot)$  be a distribution over tasks and let  $Q^*(\cdot|\cdot)$  be a task-conditional distribution over programs satisfying

$$Q^* = \arg \max_Q \mathbb{E}_\mu \left[ \max_{p \text{ maximizing } \mathbb{P}[\cdot|x, \mathcal{D}, \theta]} \log Q(p|x) \right]$$

where  $(\mathcal{D}, \theta)$  is a generative model over programs. Pick a task  $x$  where  $\mu(x) > 0$ . Partition  $\Lambda$  into expressions that are observationally equivalent under  $x$ :

$$\Lambda = \bigcup_i \mathcal{E}_i^x \text{ where for any } p_1 \in \mathcal{E}_i^x \text{ and } p_2 \in \mathcal{E}_j^x: \mathbb{P}[x|p_1] = \mathbb{P}[x|p_2] \iff i = j$$

Then there exists an equivalence class  $\mathcal{E}_i^x$  that gets all the probability mass of  $Q^*$  – e.g.,  $Q^*(p|x) = 0$  whenever  $p \notin \mathcal{E}_i^x$  – and there exists a program in that equivalence class which gets all of the probability mass assigned by  $Q^*(\cdot|x)$  – e.g., there is a  $p \in \mathcal{E}_i^x$  such that  $Q^*(p|x) = 1$  – and that program maximizes  $\mathbb{P}[\cdot|x, \mathcal{D}, \theta]$ .

*Proof.* We proceed by defining the set of “best programs” – programs maximizing the posterior  $\mathbb{P}[\cdot|x, \mathcal{D}, \theta]$  – and then showing that a best program satisfies  $Q^*(p|x) = 1$ . Define the set of best programs  $\mathcal{B}_x$  for the task  $x$  by

$$\mathcal{B}_x = \left\{ p \mid \mathbb{P}[p|x, \mathcal{D}, \theta] = \max_{p' \in \Lambda} \mathbb{P}[p'|x, \mathcal{D}, \theta] \right\}$$

For convenience define

$$f(Q) = \mathbb{E}_\mu \left[ \max_{p \in \mathcal{B}_x} \log Q(p|x) \right]$$

and observe that  $Q^* = \arg \max_Q f(Q)$ .

Suppose by way of contradiction that there is a  $q \notin \mathcal{B}_x$  where  $Q^*(q|x) = \epsilon > 0$ . Let  $p^* = \arg \max_{p \in \mathcal{B}_x} \log Q^*(p|x)$ . Define

$$Q'(p|x) = \begin{cases} 0 & \text{if } p = q \\ Q^*(p|x) + \epsilon & \text{if } p = p^* \\ Q^*(p|x) & \text{otherwise.} \end{cases}$$

Then

$$f(Q') - f(Q^*) = \mu(x) \left( \max_{p \in \mathcal{B}_x} \log Q'(p|x) - \max_{p \in \mathcal{B}_x} \log Q^*(p|x) \right) = \mu(x) (\log(Q^*(p^*|x) + \epsilon) - \log Q^*(p^*|x)) > 0$$

which contradicts the assumption that  $Q^*$  maximizes  $f(\cdot)$ . Therefore for any  $p \notin \mathcal{B}_x$  we have  $Q^*(p|x) = 0$ .

Suppose by way of contradiction that there are two distinct programs,  $q$  and  $r$ , both members of  $\mathcal{B}_x$ , where  $Q^*(q|x) = \alpha > 0$  and  $Q^*(r|x) = \beta > 0$ . Let  $p^* = \arg \max_{p \in \mathcal{B}_x} \log Q^*(p|x)$ . If  $p^* \notin \{q, r\}$  then define

$$Q'(p|x) = \begin{cases} 0 & \text{if } p \in \{q, r\} \\ Q^*(p|x) + \alpha + \beta & \text{if } p = p^* \\ Q^*(p|x) & \text{otherwise.} \end{cases}$$

Then

$$\begin{aligned} f(Q') - f(Q^*) &= \mu(x) \left( \max_{p \in \mathcal{B}_x} \log Q'(p|x) - \max_{p \in \mathcal{B}_x} \log Q^*(p|x) \right) \\ &= \mu(x) (\log(Q^*(p^*|x) + \alpha + \beta) - \log Q^*(p^*|x)) > 0 \end{aligned}$$

which contradicts the assumption that  $Q^*$  maximizes  $f(\cdot)$ . Otherwise assume  $p^* \in \{q, r\}$ . Without loss of generality let  $p^* = q$ . Define

$$Q'(p|x) = \begin{cases} 0 & \text{if } p = r \\ Q^*(p|x) + \beta & \text{if } p = p^* \\ Q^*(p|x) & \text{otherwise.} \end{cases}$$

Then

$$f(Q') - f(Q^*) = \mu(x) \left( \max_{p \in \mathcal{B}_x} \log Q'(p|x) - \max_{p \in \mathcal{B}_x} \log Q^*(p|x) \right) = \mu(x) (\log(Q^*(p^*|x) + \beta) - \log Q^*(p^*|x)) > 0$$

which contradicts the assumption that  $Q^*$  maximizes  $f(\cdot)$ . Therefore  $Q^*(p|x) > 0$  for at most one  $p \in \mathcal{B}_x$ . But we already know that  $Q^*(p|x) = 0$  for any  $p \notin \mathcal{B}_x$ , so it must be the case that  $Q^*(\cdot|x)$  places all of its probability mass on exactly one  $p \in \mathcal{B}_x$ . Call that program  $p^*$ .

Because the equivalence classes  $\{\mathcal{E}_i^x\}$  form a partition of  $\Lambda$  we know that  $p^*$  is a member of exactly one equivalence class; call it  $\mathcal{E}_i^x$ . Let  $q \in \mathcal{E}_j^x \neq \mathcal{E}_i^x$ . Then because the equivalence classes form a partition we know that  $q \neq p^*$  and so  $Q^*(q|x) = 0$ , which was our first goal: *any* program not in  $\mathcal{E}_i^x$  gets no probability mass.

Our second goal — that there is a member of  $\mathcal{E}_i^x$  which gets all the probability mass assigned by  $Q^*(\cdot|x)$  — is immediate from  $Q^*(p^*|x) = 1$ .

Our final goal — that  $p^*$  maximizes  $\mathbb{P}[\cdot|x, \mathcal{D}, \theta]$  — follows from the fact that  $p^* \in \mathcal{B}_x$ .  $\square$

Notice that Theorem 6 makes no guarantees as to the cross-task systematicity of the symmetry breaking; for example, an optimal recognition model could associate addition to the right for one task and associate addition to the left on another task. *Systematic* breaking of symmetries must arise only as a consequence as the network architecture (i.e., it is more parsimonious to break symmetries the same way for every task than it is to break them differently for each task).

As a concrete example of symmetry breaking, 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  $\mathcal{L}^{\text{MAP}}$  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 equally between  $p_1$  and  $p_2$ . Now because  $\mathbb{P}[p_1|\mathcal{D}, \theta] = \mathbb{P}[p_2|\mathcal{D}, \theta]$  (Algorithm 2), 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  $\mathcal{L}^{\text{MAP}}$  prefers  $Q_2$  (the symmetry breaking recognition model), while  $\mathcal{L}^{\text{posterior}}$  reverses this preference.

How would this example work out if we did not have a bigram parameterization of  $Q$ ? With a unigram parameterization,  $Q_2$  would be impossible to express, because it depends on local context within the syntax tree of a program. So even though the objective function would prefer symmetry breaking, a simple unigram model lacks the expressive power to encode it.

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  $\mathcal{L}^{\text{MAP}}$  works well, and we use with this combination throughout the paper.



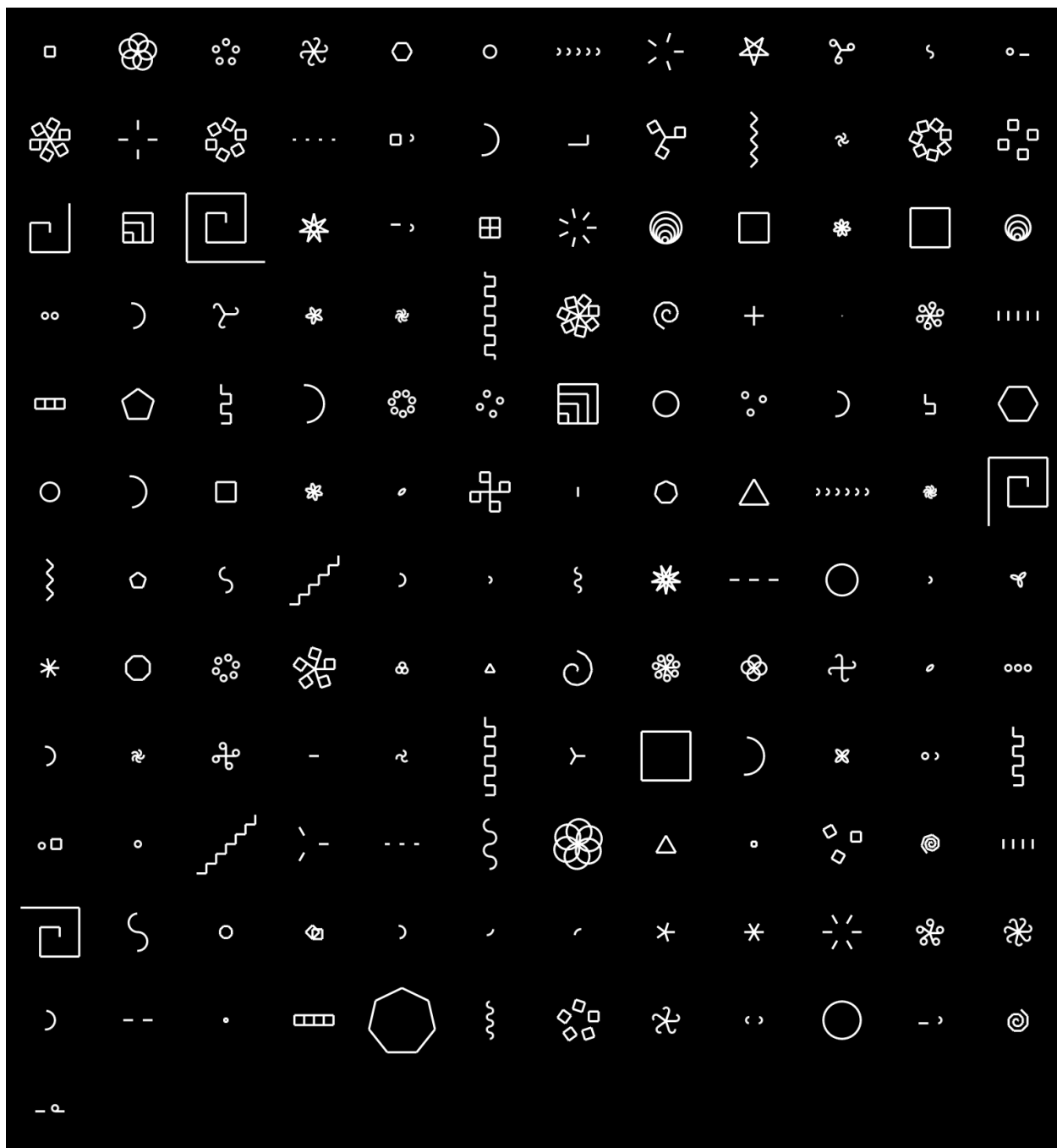


Figure 11: Full set of LOGO graphics tasks that we apply our system to

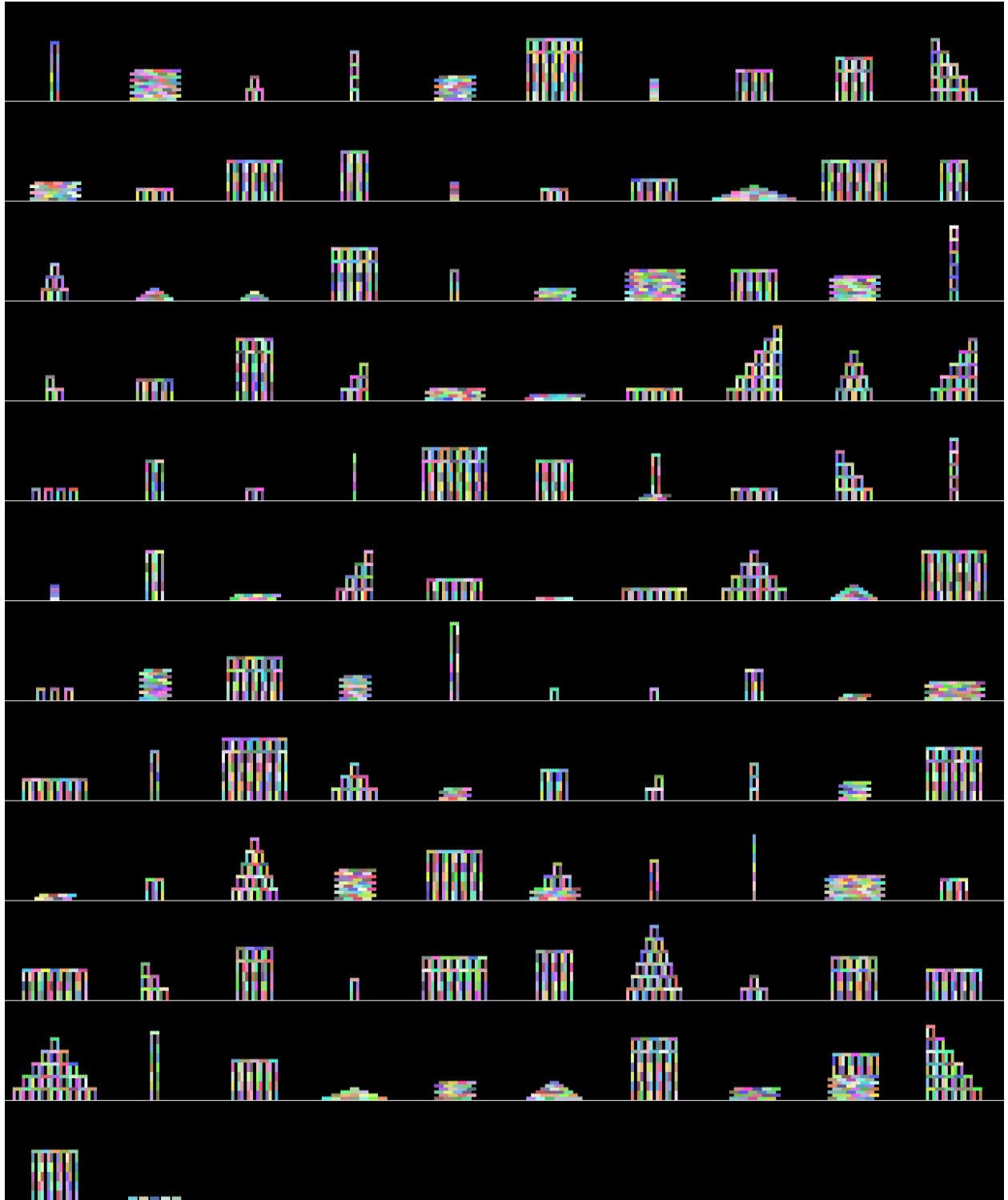


Figure 12: Full set of tower building tasks that we apply our system to

## A.8 Full set of LOGO tasks

## A.9 Full set of tower tasks

## A.10 Learning from Scratch: Tasks and DSL

We gave our system the following primitives: `if`, `=`, `>`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`, all of which are present in some form in McCarthy’s 1959 Lisp [30].<sup>5</sup> We furthermore allowed functions to call themselves, which we modeled using the Y combinator. 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.

Figure 12 shows the full set of tasks and the learned DSL.

## A.11 Learned DSLs

Here we present representative DSLs learned by our model. DSL primitives discovered by the algorithm are prefixed with `#`. Variables are prefixed with `$`, and we adopt De Bruijn indices to model bound variables [34].

### A.11.1 List processing

```
#(λ (λ (map (λ (index $0 $2)) (range ($0 (+ 1))))))
#(λ (λ (fold $1 empty (λ (λ (if ($2 $1) (cons $1 $0) $0))))))
#(+ 1 (+ 1 1))
#(λ (λ (fold $1 (cons $0 empty) (λ (λ (cons $1 $0))))))
#(+ 1 #(+ 1 (+ 1 1)))
#(λ (map (λ (if ($1 $0) (+ $0 1) 0))))
#(λ (cdr (cdr $0)))
#(λ (λ (fold (#(λ (cdr (cdr $0))) (#(λ (cdr (cdr $0))) $1)) $0 (λ (λ (cdr (#(λ (λ (fold $1 (cons
→ $0 empty) (λ (λ (cons $1 $0)))))) $0 (car $0))))))
#(λ (car (#(λ (λ (fold $1 empty (λ (λ (if ($2 $1) (cons $1 $0) $0)))))) $0 (λ (empty? (#(λ (λ
→ (fold $1 empty (λ (λ (if ($2 $1) (cons $1 $0) $0)))))) $1 (λ (gt? $0 $1))))))
#(λ (λ (length (#(λ (λ (fold $1 empty (λ (λ (if ($2 $1) (cons $1 $0) $0)))))) $1 (λ (eq? $0
→ $1))))))
#(λ (λ (#(λ (λ (fold $1 empty (λ (λ (if ($2 $1) (cons $1 $0) $0)))))) $1 (λ (is-prime (+ $1 (mod
→ $0 #(+ 1 #(+ 1 (+ 1 1))))))))))
#(λ (λ (fold $1 $0 (λ (λ (cons $1 $0))))))
#(λ (map (λ (mod $0 $1))))
#(λ (map (λ (eq? (mod $0 $1) 0))))
#(λ (gt? (mod $0 #(+ 1 (+ 1 1))) 0))
#(λ (λ (#(λ (car (#(λ (λ (fold $1 empty (λ (λ (if ($2 $1) (cons $1 $0) $0)))))) $0 (λ (empty?
→ (#(λ (λ (fold $1 empty (λ (λ (if ($2 $1) (cons $1 $0) $0)))))) $1 (λ (gt? $0 $1))))))
→ (#(λ (λ (fold $1 empty (λ (λ (if ($2 $1) (cons $1 $0) $0)))))) $1 (λ (gt? $1 (length (#(λ
→ (λ (fold $1 empty (λ (λ (if ($2 $1) (cons $1 $0) $0)))))) $2 (λ (gt? $1 $0)))))))))
```

### A.11.2 Text editing

```
#(λ (λ (fold $1 $0 (λ (λ (cons $1 $0))))))
#(λ (λ (cons '.' (cons $0 $1))))
#(λ (λ (map (λ (index $0 $2)) (range $0))))
#(+ 1)
#(λ (λ (#(λ (λ (map (λ (index $0 $2)) (range $0)))) $1 (fold (cdr $1) 0 (λ (λ (#(+ 1) (if
→ (char-eq? $2 $1) 0 $0))))))
#(λ (λ (cons (car $0) (#(λ (λ (cons '.' (cons $0 $1))) (cons '.' empty) (car $1))))))
#(λ (λ (map (λ (if (char-eq? $2 $0) $1 $0))))
#(λ (#(λ (λ (fold $1 $0 (λ (λ (cons $1 $0)))))) (cons LPAREN $0) (cons RPAREN empty)))
#(λ (λ (#(λ (λ (map (λ (index $0 $2)) (range $0)))) $0 (length (cdr (cdr $1))))))
#(λ (λ (cdr (fold (#(λ (λ (#(λ (λ (map (λ (index $0 $2)) (range $0)))) $1 (fold (cdr $1) 0 (λ (λ
→ (#(+ 1) (if (char-eq? $2 $1) 0 $0)))))) $1 $0) $1 (λ (λ (cdr $0))))))
#(λ (#(λ (λ (#(λ (λ (map (λ (index $0 $2)) (range $0)))) $0 (length (cdr (cdr $1)))))) (cdr (cdr
→ $0))))
#(#(+ 1) 1)
#(λ (λ (cons (car $1) (cons $0 empty))))
```

<sup>5</sup>McCarthy’s first version of Lisp used `cond` instead of `if`. Because we are using a typed language, we instead used `if`, because Lisp-style `cond` is unwieldy to express as a function in typed languages.

Programs & Tasks	$[1\ 9] \rightarrow 2$ $[5\ 3\ 8] \rightarrow 3$ $f(\ell) = (f_5\ \ell)$	$[[2\ 1]\ []] \rightarrow [1\ 0]$ $[[[]\ []\ [9\ 8\ 9\ 9]] \rightarrow [0\ 0\ 4]$ $f(\ell) = (f_2\ f_4\ \ell)$
	$[0\ 1\ 1\ 0\ 0] \rightarrow [1\ 1]$ $[9\ 0\ 8] \rightarrow [9\ 8]$ $f(\ell) = (f_3\ (\text{eq?}\ 0)\ \ell)$	$[1\ -1\ 0\ 2] \rightarrow [1\ 2]$ $[9\ -5\ 5\ 0\ 8] \rightarrow [9\ 5\ 8]$ $f(\ell) = (f_3\ (\text{gt?}\ 1)\ \ell)$
	$[2\ 1\ 4] \rightarrow [2\ 1\ 4\ 0]$ $[9\ 8] \rightarrow [9\ 8\ 0]$ $f(\ell) = (f_0\ \text{cons}\ \ell\ (\text{cons}\ 0\ \text{nil}))$	$[2\ 1\ 4] \rightarrow [2\ 1]$ $[9\ 8] \rightarrow [9]$ $f(\ell) = (f_6\ \text{cdr}\ (\lambda\ (z)\ (\text{empty?}\ (\text{cdr}\ z))))\ \ell$
	$[2\ 5\ 6\ 0\ 6] \rightarrow 19$ $[9\ 2\ 7\ 6\ 3] \rightarrow 27$ $f(\ell) = (f_0\ +\ \ell\ 0)$	$[4\ 2\ 6\ 4] \rightarrow [8\ 4\ 12\ 8]$ $[2\ 3\ 0\ 7] \rightarrow [4\ 6\ 0\ 14]$ $f(\ell) = (f_2\ (\lambda\ (x)\ (+\ x\ x))\ \ell)$
	$[4\ 2\ 6\ 4] \rightarrow [-4\ -2\ -6\ -4]$ $[2\ 3\ 0\ 7] \rightarrow [-2\ -3\ -0\ -7]$ $f(\ell) = (f_2\ (-\ 0)\ \ell)$	$[4\ 2\ 6\ 4] \rightarrow [5\ 3\ 7\ 5]$ $[2\ 3\ 0\ 7] \rightarrow [3\ 4\ 1\ 8]$ $f(\ell) = (f_2\ (+\ 1)\ \ell)$
	$[1\ 5\ 2\ 9] \rightarrow [1\ 2]$ $[3\ 8\ 1\ 3\ 1\ 2] \rightarrow [3\ 1\ 1]$ $f(\ell) = (f_6\ (\lambda\ (l)\ (\text{cdr}\ (\text{cdr}\ l))))\ \text{empty?}\ \ell$	$3 \rightarrow [0\ 1\ 2]$ $2 \rightarrow [0\ 1]$ $f(n) = (f_9\ n)$
	$3 \rightarrow [0\ 1\ 2\ 3]$ $2 \rightarrow [0\ 1\ 2]$ $f(n) = (f_9\ (+\ 1\ n))$	$[9\ 2] \rightarrow [9\ 9\ 2\ 2]$ $[1\ 2\ 3\ 4] \rightarrow [1\ 1\ 2\ 2\ 3\ 3\ 4\ 4]$ $f(l) = (f_0\ (\lambda\ (a\ x)\ (\text{cons}\ x\ (\text{cons}\ x\ a))))\ l\ \text{nil}$
	$0, [9\ 2\ 3] \rightarrow 9$ $3, [0\ 2\ 8\ 4\ 5\ 6] \rightarrow 4$ $f(n, l) = (f_{10}\ l\ n)$	$1, [9\ 2\ 3] \rightarrow 9$ $4, [0\ 2\ 8\ 1\ 5\ 6] \rightarrow 1$ $f(n, l) = (f_{10}\ l\ (+\ 1\ n))$
	$3 \rightarrow [-3\ -2\ -1]$ $4 \rightarrow [-4\ -3\ -2\ -1]$ $f(n) = (f_8\ 0\ n)$	$2 \rightarrow [2\ 1]$ $4 \rightarrow [4\ 3\ 2\ 1]$ $f(n) = (f_8\ 0\ (-\ 0\ n))$
	$f_0(f, l, x) = (\text{if}\ (\text{empty?}\ l)\ x\ (\text{f}\ (\text{car}\ l)\ (f_0\ (\text{cdr}\ l))))$ $(f_0: \text{fold})$ $f_2(f, l) = (f_0\ \text{nil}\ l\ (\lambda\ (x\ a)\ (\text{cons}\ (f\ x)\ a)))$ $(f_2: \text{map})$ $f_4(f, p, n) = (f_1\ p\ f\ (+\ 1)\ n)$ $(f_4: \text{generalization of range})$ $f_6(n, p, l) = (f_1\ p\ n\ \text{car}\ l)$ $(f_6: \text{specialization of unfold})$ $f_8(n, m) = (f_4\ (\lambda\ (x)\ (-\ n\ x))\ (\text{eq?}\ 0)\ m)$ $(f_7: \text{count downwards})$ $f_{10}(l, n) = (\text{car}\ (f_0\ (\lambda\ (a\ x)\ (\text{cdr}\ a))\ (f_9\ n)\ l))$ $(f_{10}: \text{index})$	$f_1(p, f, n, x) = (\text{if}\ (p\ x)\ \text{nil}\ (\text{cons}\ (f\ x)\ (f_1\ (n\ x))))$ $(f_1: \text{unfold})$ $f_3(f, l) = (f_0\ \text{nil}\ l\ (\lambda\ (x\ a)\ (\text{if}\ (f\ x)\ a\ (\text{cons}\ x\ a))))$ $(f_3: \text{filter})$ $f_5(l) = (f_0\ (\lambda\ (a\ x)\ (+\ 1\ a))\ l\ 0)$ $(f_5: \text{length})$ $f_7(p) = (f_4\ (\lambda\ (x)\ x)\ p\ 0)$ $(f_7: \text{another generalization of range})$ $f_9(n) = (f_7\ (\text{eq?}\ n))$ $(f_9: \text{range})$

Figure 13: Bootstrapping a standard library of functional programming routines, starting from recursion along with primitive operations found in 1959 Lisp. Complete set of tasks and learned DSL shown. Learned DSL components are numbered in the order that they are learned, *i.e.*, the agent first learns fold, then unfold, then uses fold to define map, etc.

```

#(λ (λ (#(λ (λ (#(λ (λ (map (λ (index $0 $2)) (range $0)))) $1 (fold (cdr $1) 0 (λ (λ (#(+ 1)
  ↳ (if (char-eq? $2 $1) 0 $0)))))) (#(λ (λ (cdr (fold (#(λ (λ (#(λ (λ (map (λ (index $0
  ↳ $2)) (range $0)))) $1 (fold (cdr $1) 0 (λ (λ (#(+ 1) (if (char-eq? $2 $1) 0 $0)))))) $1
  ↳ $0) $1 (λ (λ (cdr $0)))))) (#(λ (#(λ (λ (fold $1 $0 (λ (λ (cons $1 $0)))))) (cons LPAREN
  ↳ $0) (cons RPAREN empty))) $1) $0) $0)))
#(λ (λ (λ (#(λ (λ (fold $1 $0 (λ (λ (cons $1 $0)))))) $1 (cons $0 $2))))
#(λ (#(λ (λ (fold $1 $0 (λ (λ (cons $1 $0)))))) $0 STRING))
#(+ 1) #(+ 1) 1))
#(λ (#(λ (λ (#(λ (λ (map (λ (index $0 $2)) (range $0)))) $1 (fold (cdr $1) 0 (λ (λ (#(+ 1) (if
  ↳ (char-eq? $2 $1) 0 $0)))))) (#(λ (λ (fold $1 $0 (λ (λ (cons $1 $0)))))) STRING $0)
  ↳ SPACE)))
#(λ (λ (#(λ (λ (λ (#(λ (λ (fold $1 $0 (λ (λ (cons $1 $0)))))) $1 (cons $0 $2)))) (cons $0 $1)))
#(λ (λ (λ (fold $1 $0 (λ (λ (cons $1 $0)))))) STRING)
#(λ (λ (#(λ (λ (cons (car $0) (#(λ (λ (cons '.' (cons $0 $1)))) (cons '.' empty) (car $1))))
  ↳ (#(λ (λ (cdr (fold (#(λ (λ (#(λ (λ (map (λ (index $0 $2)) (range $0)))) $1 (fold (cdr $1)
  ↳ 0 (λ (λ (#(+ 1) (if (char-eq? $2 $1) 0 $0)))))) $1 $0) $1 (λ (λ (cdr $0)))))) $1 $0)
  ↳ $1)))
#(+ 1) #(+ 1) #(+ 1) 1))
#(λ (λ (λ (#(λ (λ (fold $1 $0 (λ (λ (cons $1 $0)))))) $1 (#(λ (λ (cdr (fold (#(λ (λ (#(λ (λ (map
  ↳ (λ (index $0 $2)) (range $0)))) $1 (fold (cdr $1) 0 (λ (λ (#(+ 1) (if (char-eq? $2 $1) 0
  ↳ $0)))))) $1 $0) $1 (λ (λ (cdr $0)))))) (#(λ (#(λ (λ (fold $1 $0 (λ (λ (cons $1
  ↳ $0)))))) $0 STRING)) (#(λ (λ (map (λ (index $0 $2)) (range $0)))) $2 #(+ 1) #(+ 1)
  ↳ #(+ 1) 1)))) $0))))

```

### A.11.3 Graphics

```

#(λ (λ (λ (logo_forLoop $2 (λ (λ (logo_FWRT $2 $3 $0))))))
#(logo_DIVA logo-UA 4)
#(λ (λ (λ (logo_forLoop $2 (λ (λ (logo_FWRT $2 $3 $0)))))) logo_IFTY)
#(logo_PT (λ (logo_FWRT logo_UL logo_ZA $0)))
#(λ (λ (λ (logo_forLoop $1 (λ (λ (logo_FWRT (logo_MULL $2 $1) $4 $0))))))
#(λ (logo_forLoop 7 (λ (λ (#(λ (λ (λ (logo_forLoop $2 (λ (λ (logo_FWRT $2 $3 $0)))))) 7
  ↳ logo_epsA $2 $0))))
#(λ (#(λ (λ (λ (logo_forLoop $2 (λ (λ (logo_FWRT $2 $3 $0)))))) logo_IFTY) (logo_SUBA logo-UA
  ↳ logo_epsA) (logo_MULL logo_epsL $0)))
#(λ (logo_forLoop logo_IFTY (λ (λ (logo_FWRT $2 logo_epsA (logo_FWRT logo_epsL logo_epsA $0))))))
#(λ (λ (λ (logo_forLoop $2 (λ (λ (logo_FWRT $2 $3 $0)))))) 4 # (logo_DIVA logo-UA 4))
#(logo_DIVA logo-UA)
#(λ (λ (λ (logo_forLoop $1 (λ (λ (logo_GETSET $2 (logo_FWRT logo_ZL $4 $0))))))
#(λ (λ (λ (logo_forLoop $2 (λ (λ (logo_FWRT $2 $3 $0)))))) logo_IFTY) (logo_DIVA logo_epsA
  ↳ 2) logo_epsL)
#(λ (λ (λ (λ (logo_forLoop $2 (λ (λ (logo_FWRT $2 $3 $0)))))) logo_IFTY) logo_epsA logo_epsL)
#(logo_forLoop 3 (λ (λ (#(λ (λ (λ (logo_forLoop $2 (λ (λ (logo_FWRT $2 $3 $0))))))
  ↳ logo_IFTY) (logo_DIVA logo_epsA 2) logo_epsL) (logo_FWRT logo_ZL # (logo_DIVA logo-UA 4)
  ↳ $0))))
#(λ (#(λ (λ (λ (logo_forLoop $1 (λ (λ (logo_GETSET $2 (logo_FWRT logo_ZL $4 $0))))))
  ↳ (#(logo_DIVA logo-UA) $0) $0))
#(λ (#(logo_PT (λ (logo_FWRT logo_UL logo_ZA $0))) (#(λ (logo_forLoop logo_IFTY (λ (λ (logo_FWRT
  ↳ $2 logo_epsA (logo_FWRT logo_epsL logo_epsA $0)))))) logo_epsL $0)))
#(λ (#(logo_PT (λ (logo_FWRT logo_UL logo_ZA $0))) (logo_FWRT logo_UL # (logo_DIVA logo-UA 4)
  ↳ $0)))
#(λ (#(λ (λ (λ (logo_forLoop $2 (λ (λ (logo_FWRT $2 $3 $0)))))) logo_IFTY) logo_epsA
  ↳ (logo_MULL logo_epsL $0)))
#(logo_FWRT logo_UL logo-UA)
#(λ (#(λ (logo_forLoop 7 (λ (λ (#(λ (λ (λ (logo_forLoop $2 (λ (λ (logo_FWRT $2 $3 $0)))))) 7
  ↳ logo_epsA $2 $0)))) (logo_MULL logo_epsL $0)))

```

### A.11.4 Towers

```

#(λ (1x3 (left 4 (1x3 (right 2 (3x1 $0))))))
#(λ (1x3 (1x3 (right 4 (1x3 (1x3 $0))))))
#(λ (tower_loopM $0 (λ (λ (left 4 (#(λ (1x3 (left 4 (1x3 (right 2 (3x1 $0)))))) $0))))))
#(λ (tower_loopM $0 (λ (λ (left 8 (#(λ (1x3 (right 4 (1x3 (right 2 (3x1 $0)))))) (left 4 (#(λ (1x3
  ↳ (1x3 (right 4 (1x3 (1x3 $0)))))) (#(λ (1x3 (left 4 (1x3 (right 2 (3x1 $0)))))) $0))))))
#(λ (λ (λ (tower_loopM $0 (λ (λ (left 3 (3x1 (right 3 (3x1 (left 4 $0)))))))))
#(λ (tower_loopM $0 (λ (λ (left 8 (#(λ (1x3 (1x3 (right 4 (1x3 (1x3 $0)))))) (left 2 (3x1
  ↳ $0))))))

```

```

#(λ (1x3 (#(λ (1x3 (1x3 (right 4 (1x3 (1x3 $0)))))) (1x3 (#(λ (1x3 (left 4 (1x3 (right 2 (3x1
  ↪ $0)))))) (right 4 $0))))))
#(λ (tower_loopM $0 (λ (λ (1x3 $0))))
#(tower_loopM 4 (λ (λ (right 6 (3x1 $0))))
#(λ (tower_loopM $0 (λ (λ (#(λ (1x3 (#(λ (1x3 (1x3 (right 4 (1x3 (1x3 $0)))))) (1x3 (#(λ (1x3
  ↪ (left 4 (1x3 (right 2 (3x1 $0)))))) (right 4 $0)))))) $0))))
#(λ (right 4 (#(λ (1x3 (1x3 (right 4 (1x3 (1x3 $0)))))) (#(λ (1x3 (left 4 (1x3 (right 2 (3x1
  ↪ $0)))))) $0))))
#(λ (right 2 (#(λ (1x3 (left 4 (1x3 (right 2 (3x1 $0)))))) $0)))
#(λ (λ (#(λ (tower_loopM $0 (λ (λ (#(λ (1x3 (#(λ (1x3 (1x3 (right 4 (1x3 (1x3 $0)))))) (1x3 (#(λ
  ↪ (1x3 (left 4 (1x3 (right 2 (3x1 $0)))))) (right 4 $0)))))) $0)))) $0 (right 2 (#(λ
  ↪ (tower_loopM $0 (λ (λ (left 4 (#(λ (1x3 (left 4 (1x3 (right 2 (3x1 $0)))))) $0)))))) $0
  ↪ $1))))))
#(tower_loopM 5 (λ (λ (#(λ (right 2 (#(λ (1x3 (left 4 (1x3 (right 2 (3x1 $0)))))) $0))) $0))))
#(λ (tower_loopM $0 (λ (λ (tower_embed (λ (#(λ (λ (λ (tower_loopM $0 (λ (λ (left $3 (3x1 (right
  ↪ 3 (3x1 (left $4 $0))))))))) 3 6 3 $0)) $0))))))
#(λ (λ (tower_loopM $0 (λ (λ (tower_embed (λ (#(λ (λ (λ (tower_loopM $0 (λ (λ (left $3 (3x1
  ↪ (right 3 (3x1 (left $4 $0))))))))) 4 5 $4 $0)) $0))))))
#(λ (λ (λ (tower_loopM $0 (λ (λ (left $3 (3x1 (right 3 (3x1 (left $4 $0))))))))) 4 5)
#(λ (λ (right 6 (tower_embed $0 $1))))
#(λ (tower_loopM $0 (λ (λ (#(λ (right 4 (#(λ (1x3 (1x3 (right 4 (1x3 (1x3 $0)))))) (#(λ (1x3
  ↪ (left 4 (1x3 (right 2 (3x1 $0)))))) $0)))) $0))))

```

### A.11.5 Symbolic regression

```

#(λ (+. $0 REAL))
#(λ (#(λ (+. $0 REAL)) (*. $0 REAL)))
#(/. REAL)
#(λ (#(λ (+. $0 REAL)) (*. $0 REAL))) (*. (#(λ (+. $0 REAL)) $0) $0)))
#(λ (/. (#(/. REAL) $0) $0))
#(λ (#(λ (#(λ (+. $0 REAL)) (*. $0 REAL))) (*. $0 (#(λ (#(λ (#(λ (+. $0 REAL)) (*. $0 REAL)))
  ↪ (*. (#(λ (+. $0 REAL)) $0) $0)))) $0))))
#(λ (#(/. REAL) (#(λ (+. $0 REAL)) $0)))
#(λ (#(λ (+. $0 REAL)) (*. (#(λ (#(λ (#(λ (+. $0 REAL)) (*. $0 REAL))) (*. (#(λ (+. $0 REAL))
  ↪ $0) $0))) $0) $0)))

```

## A.12 Hyperparameters and training details

**Neural net architecture** The recognition model for domains with sequential structure (list processing, text editing, regular expressions) is a recurrent neural network. We use a bidirectional GRU [8] with 64 hidden units that reads each input/output pair; we concatenate the input and output along with a special delimiter symbol between them. We use a 64-dimensional vectors to embed symbols in the input/output. We MaxPool the final hidden unit activations in the GRU along both passes of the bidirectional GRU.

The recognition model for domains with 2D visual structure (LOGO graphics, tower building, and symbolic regression) is a convolutional neural network. We take our convolutional architecture from [40].

We follow the RNN/CNN by an MLP with 128 hidden units and a ReLU activation which then outputs the  $Q_{ijk}$  matrix described in A.6.

**Neural net training** We train our recognition models using Adam [22] with a learning rate of 0.001.

**DREAMCODER Hyperparameters** Due to the high computational cost we performed only an informal coarse hyperparameter search. The most important parameter is the enumeration timeout during the wake phase; domains that present more challenging program synthesis problems require either longer timeouts, more CPUs, or both.

Domain	Timeout	CPUs	Batch size	$\lambda$ (A.5)	$\alpha$ (A.5.4)	Max beam size (A.2)
Lists	7m	64	10	1.5	30	5
Text	7m	64	10	1.5	30	5
Graphics	1h	96	50	1.5	30	5
Symbolic regression	2m	40	10	1	30	5
Towers	1h	64	50	1.5	30	5
Regexes	30m	64	40	1.5	30	5