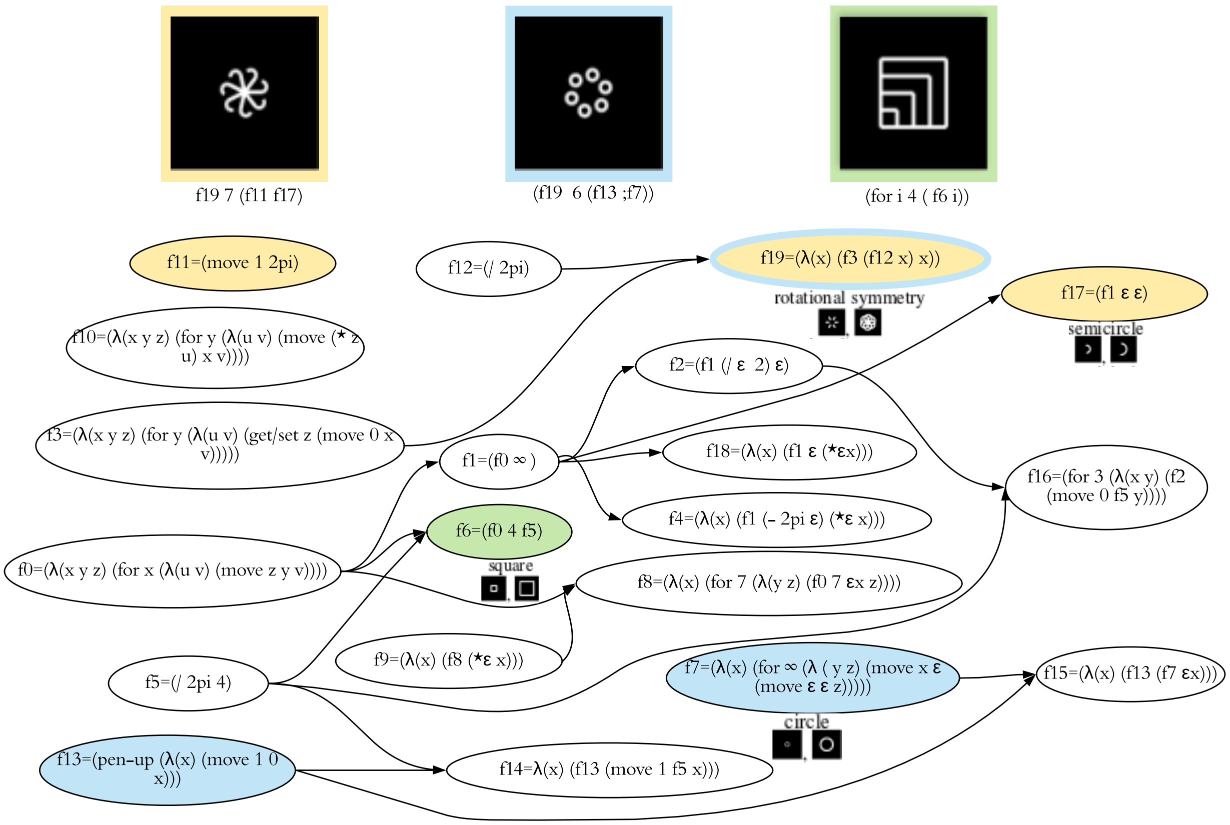
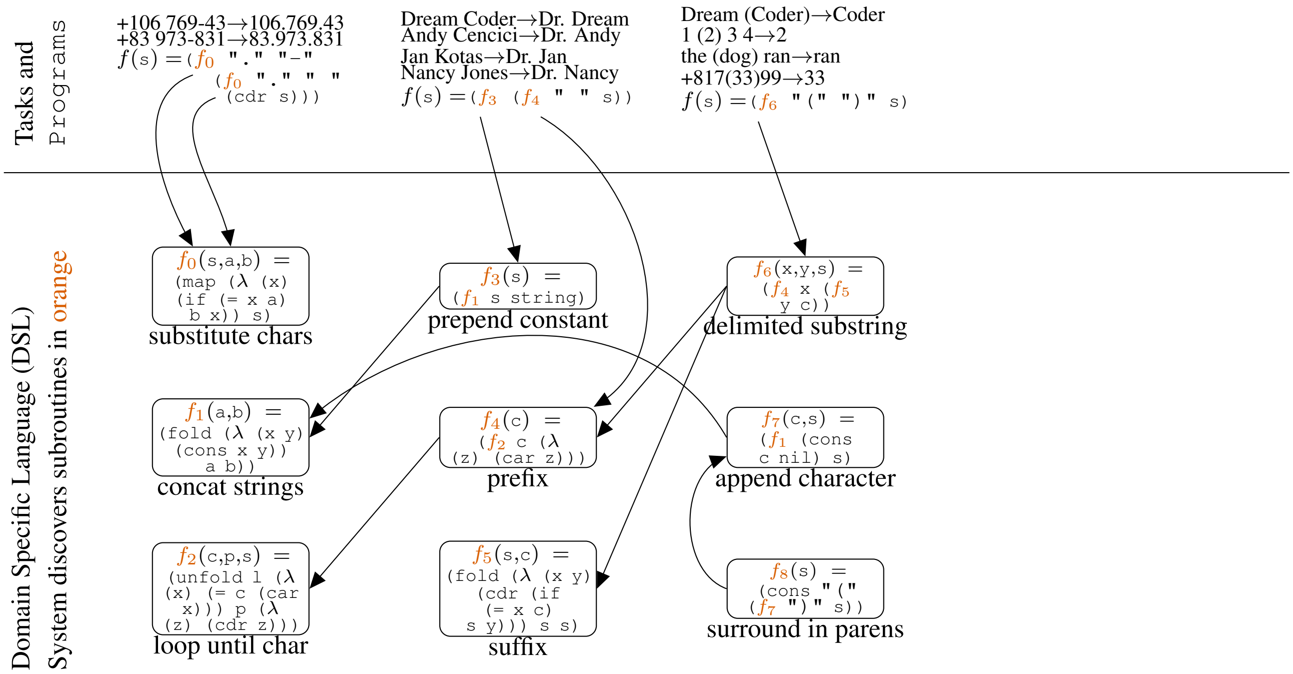
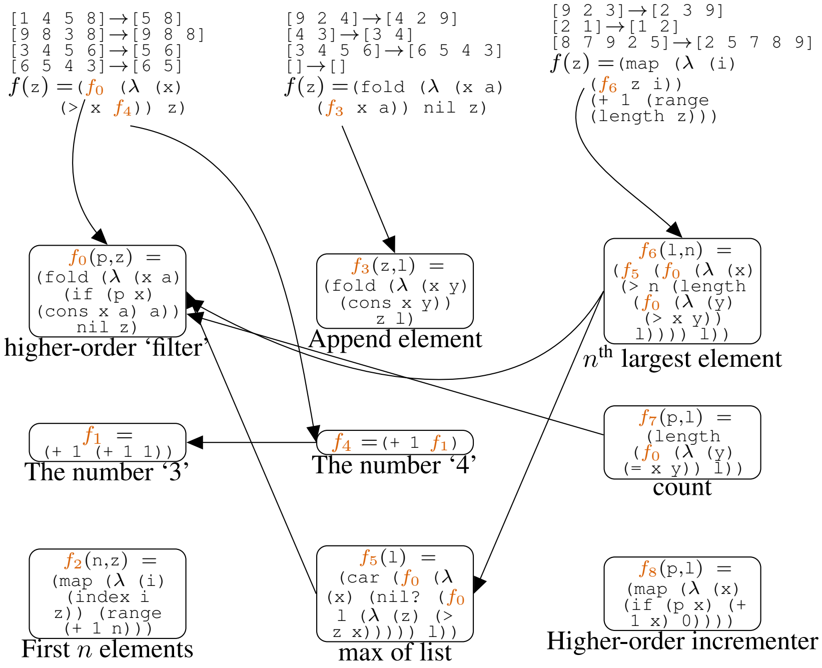

f19 7 (f11 f17)
(f19 6 (f13 ;f7))
(for i 4 (f6 i))
f11=(move 1 2pi)
f12=(/ 2pi)
f19=(λ(x) (f3 (f12 x) x))
rotational symmetry
f17=(f1 ε ε)
semicircle
f10=(λ(x y z) (for y (λ(u v) (move (⋆ z u) x v))))
f2=(f1 (/ ε 2) ε)
f3=(λ(x y z) (for y (λ(u v) (get/set z (move 0 x v)))))
f1=(f0 ∞)
f18=(λ(x) (f1 ε (⋆εx)))
f16=(for 3 (λ(x y) (f2 (move 0 f5 y))))
f4=(λ(x) (f1 (– 2pi ε) (⋆ε x)))
f6=(f0 4 f5)
square
f0=(λ(x y z) (for x (λ(u v) (move z y v))))
f8=(λ(x) (for 7 (λ(y z) (f0 7 εx z))))
f9=(λ(x) (f8 (⋆ε x)))
f5=(/ 2pi 4)
f7=(λ(x) (for ∞ (λ (y z) (move x ε (move ε ε z)))))
f15=(λ(x) (f13 (f7 εx)))
circle
f13=(pen–up (λ(x) (move 1 0 x)))
f14=λ(x) (f13 (move 1 f5 x)))

Figure 1: **Bottom**: Learned graphics programing DSL. Each circle encloses a different learned component; arrows point from a component to other components that use it. **Top**: 3 tasks, and below them the programs that solve them. Yellow/Blue/Green highlighting indicates which DSL components are used to solve which tasks.


Tasks and Programs
+106 769-43→106.769.43
+83 973-831→83.973.831
f(s) = (f0 "." "-" (f0 "." " " (cdr s)))
Dream Coder→Dr. Dream
Andy Cencici→Dr. Andy
Jan Kotas→Dr. Jan
Nancy Jones→Dr. Nancy
f(s) = (f3 (f4 " " s))
Dream (Coder)→Coder
1 (2) 3 4→2
the (dog) ran→ran
+817(33)99→33
f(s) = (f6 "(" ")" s)
Domain Specific Language (DSL)
System discovers subroutines in orange
f0(s,a,b) = (map (λ (x) (if (= x a) b x)) s)
substitute chars
f3(s) = (f1 s string)
prepend constant
f6(x,y,s) = (f4 x (f5 y c))
delimited substring
f1(a,b) = (fold (λ (x y) (cons x y)) a b))
concat strings
f4(c) = (f2 c (λ (z) (car z)))
prefix
f7(c,s) = (f1 (cons c nil) s)
append character
f2(c,p,s) = (unfold l (λ (x) (= c (car x))) p (λ (z) (cdr z)))
loop until char
f5(s,c) = (fold (λ (x y) (cdr (if (= x c) s y))) s s)
suffix
f8(s) = (cons "(" (f7 ")" s))
surround in parens


[1 4 5 8]→[5 8]
[9 8 3 8]→[9 8 8]
[3 4 5 6]→[5 6]
[6 5 4 3]→[6 5]
f(z) = (f0 (λ (x) (> x f4)) z)
[9 2 4]→[4 2 9]
[4 3]→[3 4]
[3 4 5 6]→[6 5 4 3]
[]→[]
f(z) = (fold (λ (x a) (f3 x a)) nil z)
[9 2 3]→[2 3 9]
[2 1]→[1 2]
[8 7 9 2 5]→[2 5 7 8 9]
f(z) = (map (λ (i) (f6 z i)) (+ 1 (range (length z))))
f0(p,z) = (fold (λ (x a) (if (p x) (cons x a) a)) nil z)
higher-order 'filter'
f3(z,l) = (fold (λ (x y) (cons x y)) z l)
Append element
f6(l,n) = (f5 (f0 (λ (x) (> n (length (f0 (λ (y) (> x y)) l)))) l))
nth largest element
f1 = (+ 1 (+ 1 1))
The number '3'
f4 = (+ 1 f1)
The number '4'
f7(p,l) = (length (f0 (λ (y) (= x y)) l))
count
f2(n,z) = (map (λ (i) (index i z)) (range (+ 1 n)))
First n elements
f5(l) = (car (f0 (λ (x) (nil? (f0 l (λ (z) (> z x)))))) l))
max of list
f8(p,l) = (map (λ (x) (if (p x) (+ 1 x) 0))))
Higher-order incrementer

1 Introduction

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

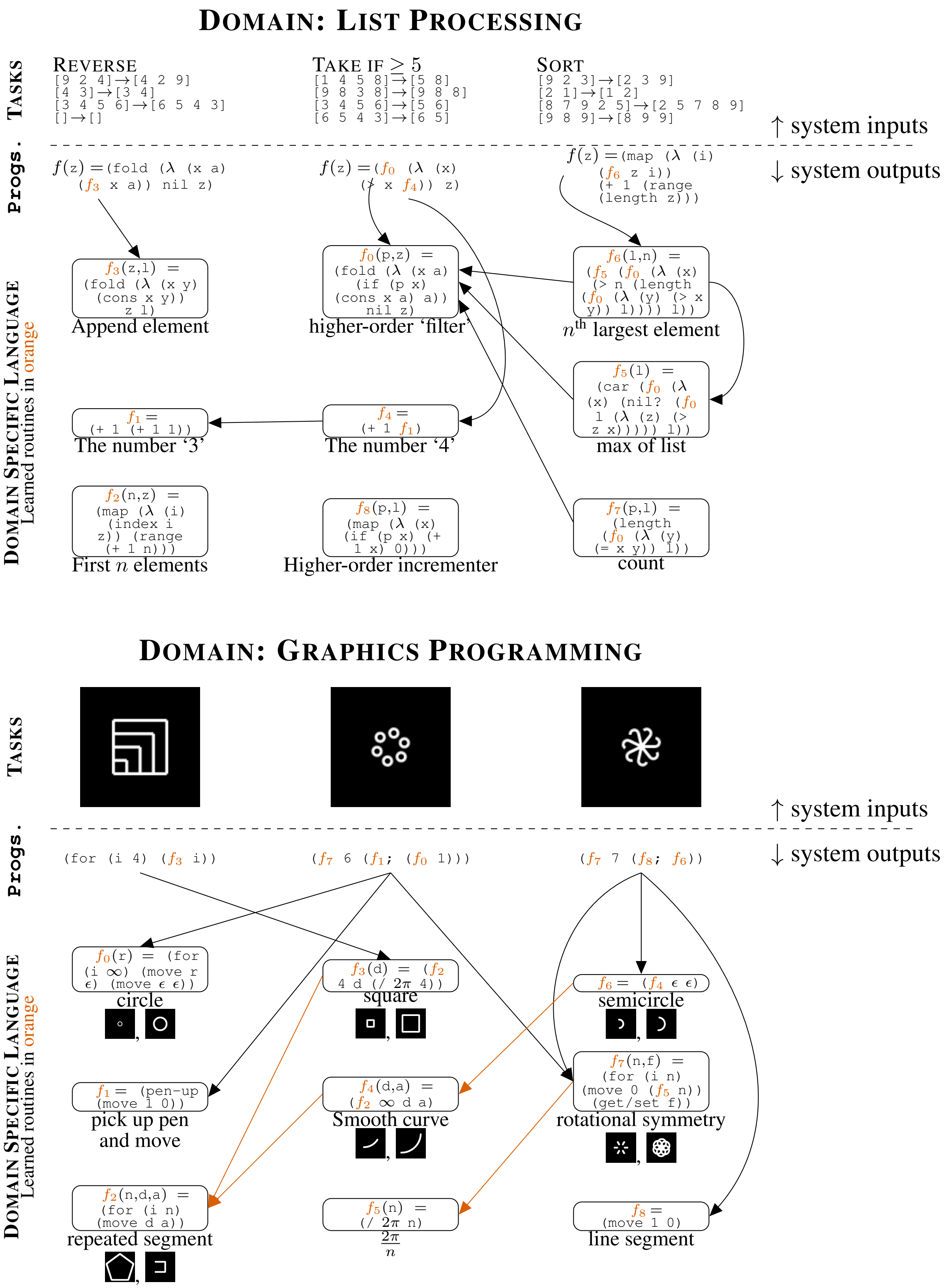
Recent program-learning systems developed within the AI and machine learning community are impressive along many dimensions, authoring programs for problem domains like drawing pictures [**?**, 10], transforming text [13] and numerical sequences [2], and reasoning over common sense knowledge bases [26]. These systems work in different ways, but typically hinge upon a carefully hand-engineered Domain Specific Language (DSL). The DSL restricts the space of programs to contain the kinds of concepts needed for one specific domain. For example, a picture-drawing DSL could include concepts like circles and spirals, and a DSL for numerical sequences could include sorting and reversing lists of numbers. Modern systems also learn how to efficiently deploy the DSL on new problems [6, 2, 17], but – unlike human learners – do not discover the underlying system of concepts needed to navigate the domain.

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

We call our system 'DreamCoder' because it acquires these two kinds of expertise through a novel kind of wake/sleep or 'dream' learning [14], iterating through a wake cycle – where it solves problems by writing programs – and a pair of sleep cycles, both of which are loosely biologically inspired by actual human sleep. The first sleep cycle, which we refer to as **consolidation**, grows the DSL by replying experiences from waking and consolidating them into new code abstractions. This cycle is inspired by the formation of abstractions during sleep memory consolidation [7]. The second sleep cycle, which we refer to as **dreaming**, improves the agents knowledge of how to write programs by training a neural network to help search for programs. The neural net is trained on replayed experiences as well as 'fantasies', or samples, from the DSL. These two kinds of dreams are inspired by the distinct episodic replay and hallucination components of dream sleep [12].

This dream-learning architecture brings together two lines of prior work, both of which have been separately influential within artificial intelligence. One line of work considers the problem of learning new concepts, abstractions, or 'options' from experience [5, 22, 36, 16, 37], while the other line of work considers the problem of learning how to deploy those concepts efficiently [6, 2, 17]. Our goal with DreamCoder is to show that the combination of these ideas is uniquely powerful, and pushes us toward program-writing systems that, like human learners, autonomously acquire the expertise needed to navigate a new domain of problems.

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



```
f(z) = (fold (λ (x a)
   (f3 x a)) nil z)

f(z) = (f0 (λ (x)
   (> x f4)) z)

f(z) = (map (λ (i)
   (f6 z i))
   (+ 1 (range
   (length z)))

f3(z,l) =
(fold (λ (x y)
(cons x y))
z l)

f0(p,z) =
(fold (λ (x a)
(if (p x)
(cons x a) a))
nil z)

f6(l,n) =
(f5 (f0 (λ (x)
(> n (length
(f0 (λ (y) (> x
y)) l))))) l))

f5(l) =
(car (f0 (λ
(x) (nil? (f0
l (λ (z) (>
z x)))))) l))

f1 =
(+ 1 (+ 1 1))

f4 =
(+ 1 f1)

f2(n,z) =
(map (λ (i)
(index i
z)) (range
(+ 1 n)))

f8(p,l) =
(map (λ (x)
(if (p x) (+
1 x) 0)))

f7(p,l) =
(length
(f0 (λ (y)
(= x y)) l))
```

```
(for (i 4) (f3 i))

(f7 6 (f1; (f0 1)))

(f7 7 (f8; f6))

f0(r) = (for
(i ∞) (move r
ϵ) (move ϵ ϵ))

f3(d) = (f2
4 d (/ 2π 4))

f6 = (f4 ϵ ϵ)

f1 = (pen-up
(move 1 0))

f4(d,a) =
(f2 ∞ d a)

f7(n,f) =
(for (i n)
(move 0 (f5 n))
(get/set f))

f2(n,d,a) =
(for (i n)
(move d a))

f5(n) =
(/ 2π n)

f8 =
(move 1 0)
```

$\frac{2\pi}{n}$

Figure 2: Two of the six domains we apply our system to. Agent observes tasks (top rows) which it solves by writing programs (middle rows) while jointly growing a library (DSL; bottom rows). Learned DSLs rediscover multiple higher-order functions (`filter` for list functions and rotational symmetry for generative graphics). Learned DSL components call each other (arrows).

2 Wake/Sleep Program Induction

DREAMCODER takes as its goal to acquire domain-specific expertise, which it learns by solving a collection of programming **tasks**. It alternatingly finds programs that solve tasks (Wake – Figure 3 top); improves its DSL by discovering and reusing domain-specific subroutines (Consolidation – Figure 3 left); and trains a neural network that efficiently guides search for programs in the DSL (Dreaming – Figure 3 right). The learned DSL acts as a a prior on programs likely to solve tasks in the domain, while the neural net looks at a specific task and produces a "posterior" for programs likely to solve that specific task (Figure 3 middle). The neural network thus functions as a **recognition model** supporting a form of approximate Bayesian program induction, jointly trained with a **generative model** for programs encoded in the DSL, in the spirit of the Helmholtz machine [14]. The recognition model ensures that searching for programs remains tractable even as the DSL (and hence the search space for programs) expands. The generative model, or DSL, distills out common abstractions across programs discovered during waking, growing a network of increasingly deep and specialized domain-specific concepts (Figure 2, bottom rows).

Viewed as an inference problem, our goal is to start with a collection of tasks, written X, and infer both a program for each task, as well as a distribution over programs encoded by a DSL, written $\mathcal{D}$. We equip $\mathcal{D}$ with a learned weight vector θ, and together $(\mathcal{D}, \theta)$ define a generative model over programs (see Appendix 2). We perform maximum a posteriori (MAP) inference of $(\mathcal{D}, \theta)$ given X. Writing J for the joint probability of $(\mathcal{D}, \theta)$ and X, we want the $\mathcal{D}^*$ and θ^* solving:

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

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

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

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

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

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

Our wake and sleep cycles correspond to alternate maximization of $\mathscr{L}$ w.r.t. $\{\mathcal{F}_x\}_{x\in X}$ (**Wake**) and $(\mathcal{D}, \theta)$ (**Consolidation**):
Wake: Maxing $\mathscr{L}$ w.r.t. the frontiers. Here $(\mathcal{D}, \theta)$ is fixed and we want to find new programs to add to the frontiers so that $\mathscr{L}$ increases the most. $\mathscr{L}$ most increases by finding programs where $\mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D},\theta] \propto \mathbb{P}[p|x,\mathcal{D},\theta]$ is large (i.e., programs with high posterior probability).
Sleep (Consolidation): Maxing $\mathscr{L}$ w.r.t. the DSL. Here $\{\mathcal{F}_x\}_{x\in X}$ is held fixed, and so we can evaluate $\mathscr{L}$. Now the problem is that of searching the discrete space of DSLs and finding one maximizing $\int \mathscr{L}\ \mathrm{d}\theta$, and then updating θ to $\arg\max_\theta \mathscr{L}(\mathcal{D}, \theta, \{\mathcal{F}_x\})$.

Searching for programs is hard because of the large combinatorial search space. We ease this difficulty by training a neural recognition model, $Q(\cdot|\cdot)$, during the **Dreaming** phase: Q is trained to approximate the posterior over programs, $Q(p|x) \approx \mathbb{P}[p|x,\mathcal{D}] \propto \mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D}]$. Thus training the neural network amortizes the cost of finding programs with high posterior probability.
Sleep (Dreaming): tractably maxing $\mathscr{L}$ w.r.t. the frontiers. Here we train $Q(p|x)$ to assign high probability to programs p where $\mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D},\theta]$ is large, because incorporating those programs into the frontiers will most increase $\mathscr{L}$.

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

[1] 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.

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

2.1 Wake: Solving tasks

During waking we enumerate programs from the DSL in decreasing order of their probability according to the recognition model, and then check if a program p assigns positive probability to a task ($\mathbb{P}[x|p] > 0$); if so, we incorporate p into the frontier $\mathcal{F}_x$ (Figure 4). We represent programs as polymorphicly-typed λ-calculus expressions, a representation closely resembling Lisp and functional languages like Haskell and OCaml, including variables, conditionals, higher-order recursive functions, and the ability to create new functions.

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

2.2 Consolidation-Sleep: Growing a Domain Specific Language

The DSL offers a set of abstractions that allow an agent to concisely express solutions to the tasks at hand. We automatically discover these new abstractions by combining two ideas. First, we build on techniques from the programming languages community to develop a new algorithm for automatically refractoring programs, where this refactoring exposes common reused subexpressions found 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 $\int \mathscr{L}\, \mathrm{d}\theta$ (Sec. 2). We replace this marginal with an AIC approximation and minimize the following expression, which can be interpreted as a kind of compression:

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

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

However, there is a snag with this simple approach: whenever we add a new expression e to the DSL, the programs found during waking ($\{\mathcal{F}_x\}$) are not written in terms of e. Concretely, imagine we wanted to discover a new DSL procedure for doubling numbers, after having found the programs `(cons (+ 9 9) nil)` and `(λ (x) (+ (car x) (car x)))`. As human programmers, we can look at these pieces of code and recognize that, if we define a new procedure called `double`, defined as `(λ (x) (+ x x))`, then we can rewrite the original programs as `(cons (double 9) nil)` and `(λ (x) (double (car x)))`. This process is a kind of refactoring where a new subroutine is defined (`double`) and the old programs rewritten in terms of the new subroutine. Figure 5A diagrams this refactoring process for a more complicated setting, where the agent must rediscover the higher-order function `map` starting from the basics of Lisp and the Y-combinator.

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

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

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

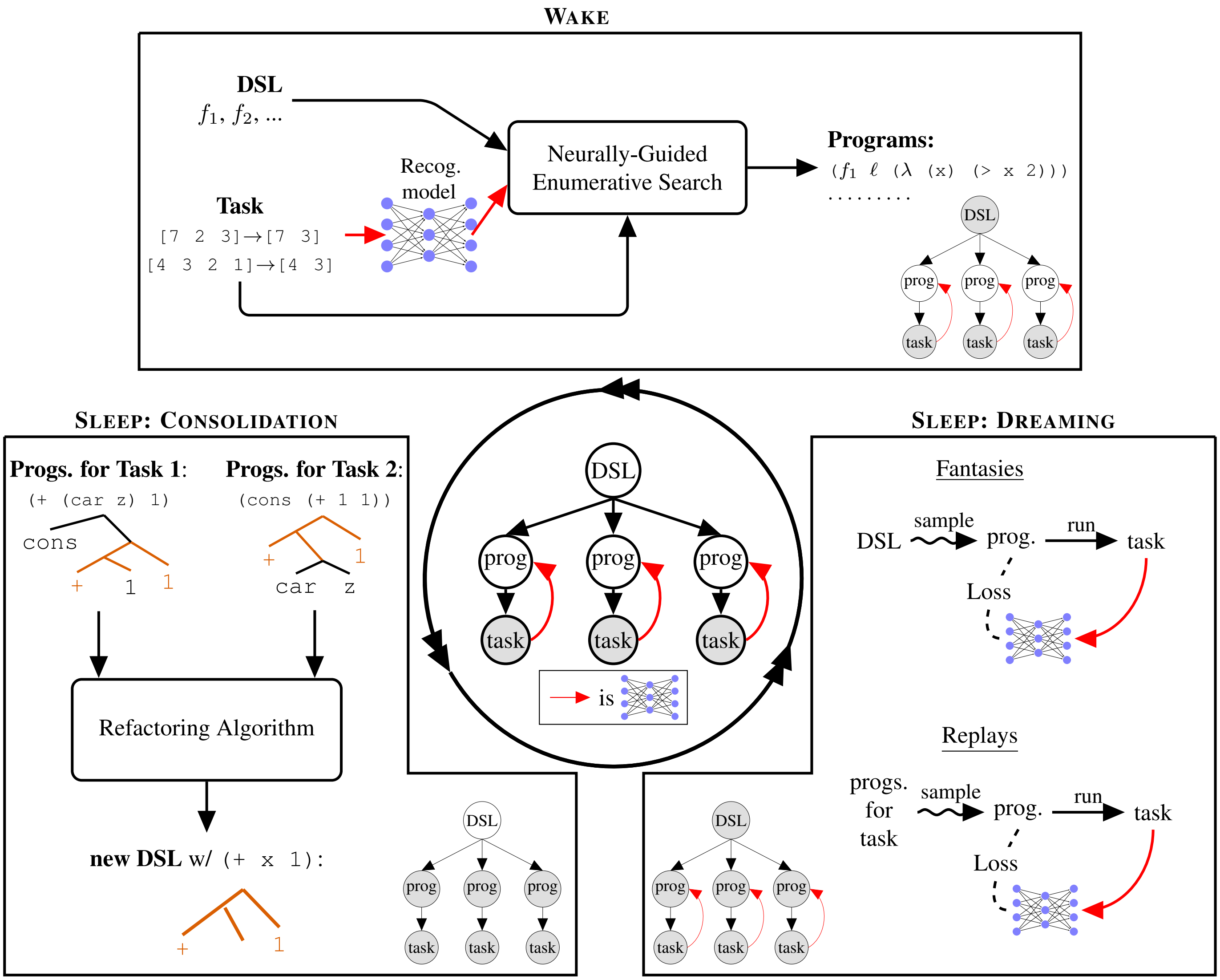


Figure 3: **Middle:** DREAMCODER as a graphical model. Agent observes programming tasks (e.g., input/outputs for list processing or images for graphics programs), which it explains with latent programs, while jointly inferring a latent Domain Specific Language (DSL) capturing cross-program regularities. 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).

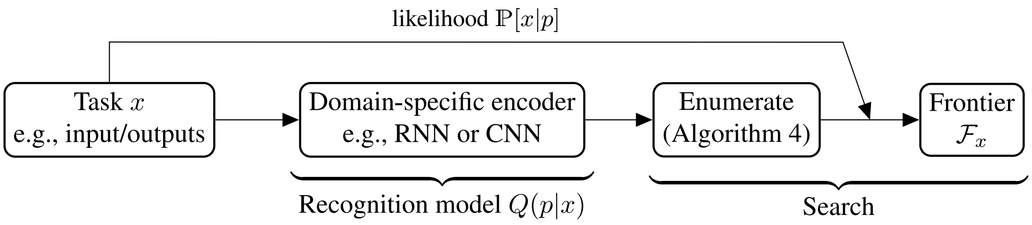


Figure 4: Neurally-guided program inference pipeline. Recognition model outputs distribution over program $Q(p|x)$. Program output by enumerative search incorporated into frontier if likelihood $\mathbb{P}[x|p] > 0$

a program. To make refactoring tractable we first limit the degree to which a piece of code can be refactored: rather than consider every refactoring, we bound the number of λ-calculus evaluation steps separating a refactoring from its original program. Now the number of refactoring this is finite but astronomically large: for the example in Figure 5 there are approximately 10^{14} possible refactorings – a quantity that grows exponentially both as a function of program size and a function of the bound on evaluation steps. How can we tame this combinatorial explosion?

A structure called a **version space algebra** [21, 25, 29] neatly resolves this exponential growth. A version space is a tree-shaped data structure that compactly represents a large set of programs and supports efficient set operations like union, intersection, and membership checking. In Appendix A.3, we give a dynamic program that takes as input a program p and then outputs a version space containing the refactorings of p. Figure 5B 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 5A, we represent the space of refactorings using a version space with 10^6 nodes, which encodes 10^{14} refactorings.

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

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

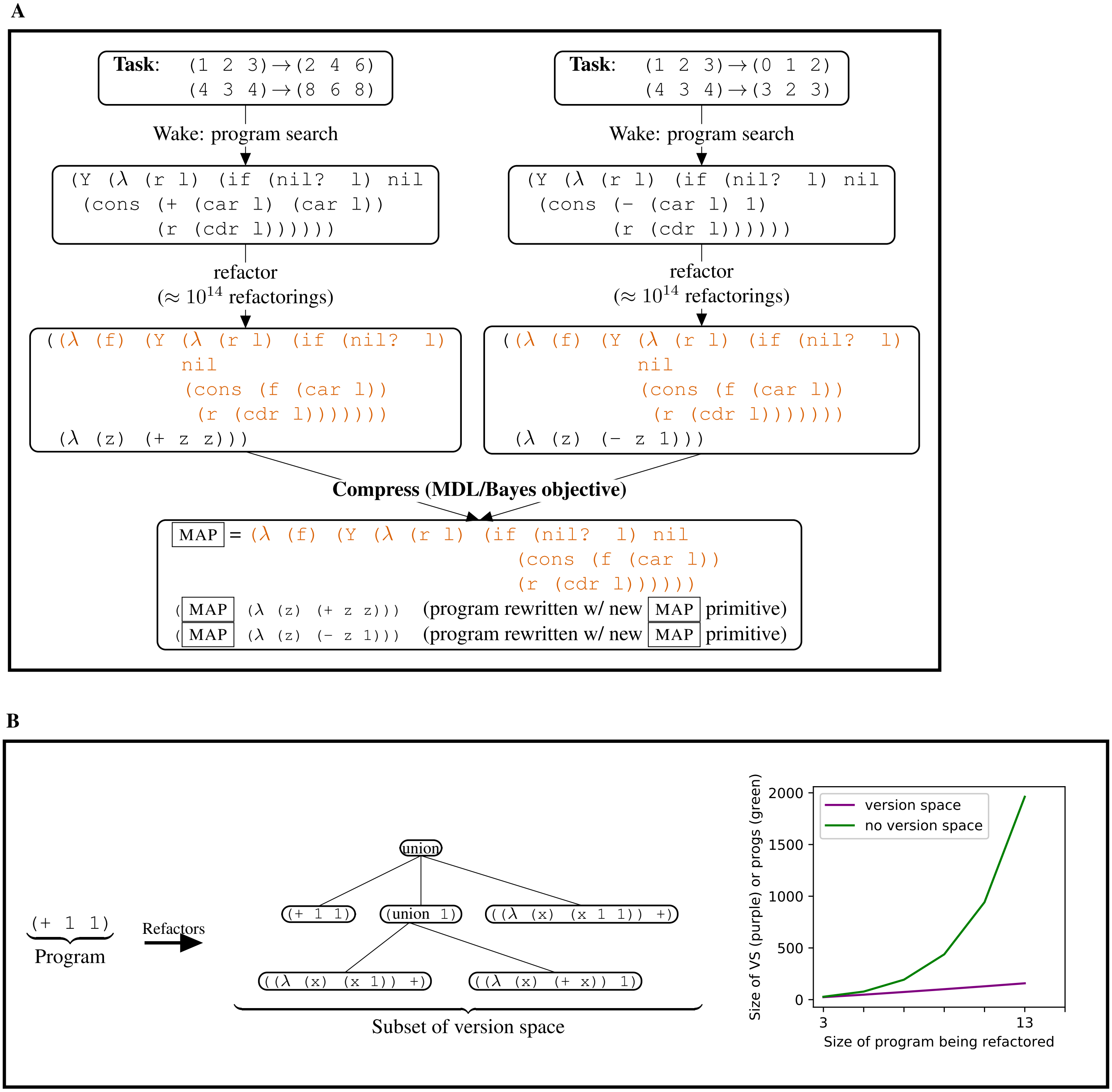
Algorithm 1 DSL Induction Algorithm

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

2.3 Dream Sleep: Training a Neural Recognition Model

During "dreaming" the system learns a recognition model that guides program search. It learns from (program, task) pairs drawn from two sources of self-supervised data: *replays* of programs discovered during waking, and *fantasies*, or programs drawn from the DSL. Replays ensure that the recognition model is trained on the 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, $\mathbb{E}\left[\text{KL}\left(\mathbb{P}[p|x, \mathcal{D}, \theta] \| Q(p|x)\right)\right]$, or we can train Q to perform MAP inference by maximizing $\mathbb{E}\left[\max_{p \text{ maxing } \mathbb{P}[\cdot|x, \mathcal{D}, \theta]} \log Q(p|x)\right]$, 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


A
Task: (1 2 3)→(2 4 6) (4 3 4)→(8 6 8)
Task: (1 2 3)→(0 1 2) (4 3 4)→(3 2 3)
Wake: program search
Wake: program search
refactor (≈ 10^14 refactorings)
refactor (≈ 10^14 refactorings)
Compress (MDL/Bayes objective)
B
(+ 1 1)
Program
Refactors
union
(+ 1 1)
(union 1)
((λ (x) (x 1 1)) +)
((λ (x) (x 1)) +)
((λ (x) (+ x)) 1)
Subset of version space
version space
no version space
Size of VS (purple) or progs (green)
0
500
1000
1500
2000
3
13
Size of program being refactored

```
(Y (λ (r l) (if (nil?  l) nil
  (cons (+ (car l) (car l))
        (r (cdr l))))))
```

```
(Y (λ (r l) (if (nil?  l) nil
  (cons (- (car l) 1)
        (r (cdr l))))))
```

```
((λ (f) (Y (λ (r l) (if (nil?  l)
             nil
             (cons (f (car l))
              (r (cdr l)))))))
 (λ (z) (+ z z)))
```

```
((λ (f) (Y (λ (r l) (if (nil?  l)
             nil
             (cons (f (car l))
              (r (cdr l)))))))
 (λ (z) (- z 1)))
```

```
MAP = (λ (f) (Y (λ (r l) (if (nil?  l) nil
                         (cons (f (car l))
                         (r (cdr l)))))))
(MAP (λ (z) (+ z z)))  (program rewritten w/ new MAP primitive)
(MAP (λ (z) (- z 1)))  (program rewritten w/ new MAP primitive)
```

Figure 5: DSL learning as code refactoring. **Panel A:** For each task we discover programs during waking, then refactor the code from those programs to expose common subprograms (highlighted in orange). Common subprograms are incorporated into the DSL when they increase a Bayesian objective. Intuitively, these new DSL components best compress the programs found during waking. **Panel B:** # of possible refactorings grows exponentially with program size, 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 $1 + 1 + \cdots + 1$.

the recognition model, $\mathcal{L}^{\text{posterior}}$ and $\mathcal{L}^{\text{MAP}}$, which either train Q to perform full posterior inference or MAP inference, respectively. These objectives combine replays and fantasies:

$$\mathcal{L}^{\text{posterior}} = \mathcal{L}^{\text{posterior}}_{\text{Replay}} + \mathcal{L}^{\text{posterior}}_{\text{Fantasy}} \qquad \mathcal{L}^{\text{MAP}} = \mathcal{L}^{\text{MAP}}_{\text{Replay}} + \mathcal{L}^{\text{MAP}}_{\text{Fantasy}}$$

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

$$\mathcal{L}^{\text{posterior}}_{\text{Fantasy}} = \mathbb{E}_{(p,x)\sim(\mathcal{D},\theta)}\left[\log Q(p|x)\right] \qquad \mathcal{L}^{\text{MAP}}_{\text{Fantasy}} = \mathbb{E}_{x\sim(\mathcal{D},\theta)}\left[\max_{\substack{p \\ p \text{ maxing } \mathbb{P}[\cdot|x,\mathcal{D},\theta]}} \log Q(p)\right]$$

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 (Sec. 2.3.2).

2.3.1 Parameterizing Q

Broadly the literature contains two different approaches to parameterizing conditional distributions over programs. The first approach [6, 40] is to use a recurrent network to predict the entire program token-by-token, which has the advantage that, if the network is sufficiently powerful, it can completely solve the synthesis problem. The disadvantage is that these models can perform poorly at out-of-sample generalization [], which is critical for our setting, as the agent may need to solve new tasks that are qualitatively different from the tasks it has solved so far.

The second approach is to have Q predict a fixed-dimensional weight vector, which then biases a fast enumerator [2, 9] or sampler [24]. This approach can enjoy strong out-of-sample generalization, because it can fall back on enumeration or sampling when the target program is unlike the training programs. A main drawback is that the neural net is deliberately handicapped, and can only send so much information about the target program.

We adopt a middle ground between these two extremes. Our recognition model predicts a distribution over primitives in the DSL, conditioned on the local context in the syntax tree of the program. When predicting the next node to add to the syntax tree of a program, the recognition model conditions on the parent node, as well as which argument is being generated. This is a kind of 'bigram' model over trees, where the bigrams take the form of (parent, child, argument index). Figure 6 diagrams this generative process and Algorithm 4 specifies a sampling procedure for $Q(\cdot|x)$. This parameterization has three main advantages: (1) it supports fast enumeration and sampling of programs, because the recognition model only needs to run once for each task; (2) it allows the recognition model to provide fine-grained information about the structure of the target program; and (3) training this recognition model causes it to learn to break symmetries in the space of programs, described next.

2.3.2 Learning to break symmetries in program space

A good DSL not only exposes high-level building blocks, but also carefully restricts the ways in which those building blocks are allowed to compose. For example, a DSL for arithmetic should contain both addition and the number zero but disallow adding zero. These restrictions break symmetries in the space of programs. A bigram parameterization of the recognition model, combined with the $\mathcal{L}^{\text{MAP}}$ training objective, interact in a way that breaks symmetries in the program space, 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.5).

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[2] parameterization. Figure 8 shows the result of training Q in these four regimes for a DSL containing `+`, `0`, and `1` and then sampling programs. On this particular run, the combination of bigrams and $\mathcal{L}^{\text{MAP}}$ learns to avoid adding zero and associate addition to the right — different random initializations lead to either right or left association.

[2] 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 [8]

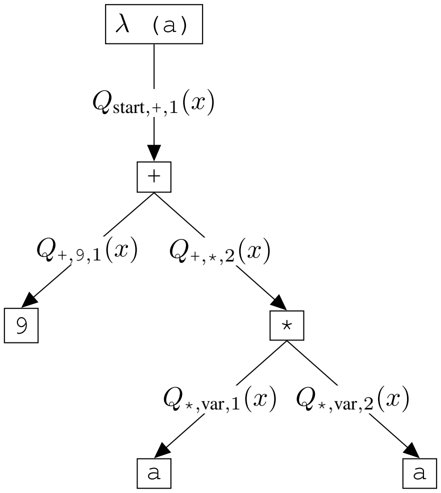
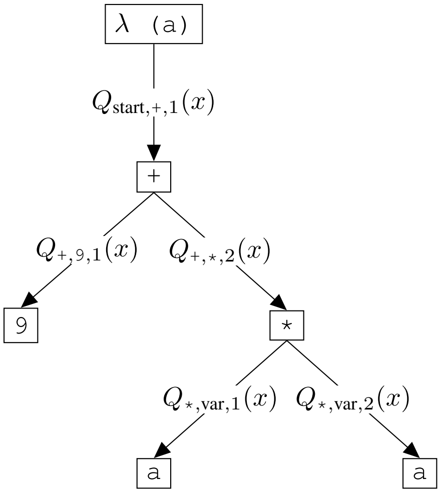

λ (a)
$Q_{\text{start},+,1}(x)$
+
$Q_{+,9,1}(x)$
$Q_{+,\star,2}(x)$
9
*
$Q_{\star,\text{var},1}(x)$
$Q_{\star,\text{var},2}(x)$
a
a

Figure 6: Parameterization of distribution over programs predicted by recognition model. Here the program (syntax tree shown above) is `(λ (a) (+ 9 (* a a )))`. Each conditional distribution predicted by the recognition model is written $Q_{\text{parent,child,argument index}}(x)$, where x is a task.


λ (a)
$Q_{\text{start},+,1}(x)$
+
$Q_{+,9,1}(x)$
$Q_{+,\star,2}(x)$
9
*
$Q_{\star,\text{var},1}(x)$
$Q_{\star,\text{var},2}(x)$
a
a

	Unigram	Bigram

$\mathcal{L}^{\text{posterior}}$	*Three samples:*
	`(+ 1 0)`
	`(+ (+ 0 0)`
	`   (+ 1 0))`
	`(+ 1 1)`
	63.0% right-associative
	37.4% `+0`’s
		Three samples:
		`0`
		`(+ (+ (+ 0 0)`
		`      (+ 0 1)) 1)`
		`1`
		55.8% right-associative
		31.9% `+0`’s

$\mathcal{L}^{\text{MAP}}$	*Three samples:*
	`1`
	`(+ 1 (+ 1 (+ (+ 1`
	`   (+ 1 1)) 1)))`
	`(+ (+ 1 1) 1)`
	48.6% right-associative
	0.5% `+0`’s
		Three Samples:
		`(+ 1 (+ 1 (+ 1`
		`      (+ 1 (+ 1 1)))))`
		`0`
		`(+ 1 (+ 1 (+ 1 1)))`
		97.9% right-associative
		2.5% `+0`’s

Figure 7: **Left:** Bigram parameterization of distribution over programs predicted by recognition model. Here the program (syntax tree shown above) is `(λ (a) (+ 9 (* a a )))`. Each conditional distribution predicted by the recognition model is written $Q_{\text{parent,child,argument index}}(x)$, where x is a task. **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`.

	Unigram	Bigram

$\mathcal{L}^{\text{posterior}}$	*Three samples:*
	`(+ 1 0)`
	`(+ (+ 0 0) (+ 1 0))`
	`(+ 1 1)`
	63.0% right-associative; 37.4% `+0`'s
		Three samples:
		`0`
		`(+ (+ (+ 0 0) (+ 0 1)) 1)`
		`1`
		55.8% right-associative; 31.9% `+0`'s

$\mathcal{L}^{\text{MAP}}$	*Three samples:*
	`1`
	`(+ 1 (+ 1 (+ (+ 1 (+ 1 1)) 1)))`
	`(+ (+ 1 1) 1)`
	48.6% right-associative; 0.5% `+0`'s
		Three Samples:
		`(+ 1 (+ 1 (+ 1 (+ 1 (+ 1 1)))))`
		`0`
		`(+ 1 (+ 1 (+ 1 1)))`
		97.9% right-associative; 2.5% `+0`'s

Figure 8: 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`.

3 Experiments

3.1 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 few input/output examples, using a recurrent neural network for the recognition model and starting with a generic functional programming basis: `foldr`, `unfold`, `if`, `map`, `length`, `index`, `=`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`.

3.1.1 List Processing

We took 218 list manipulation tasks from our previous work [9], each with 15 input/output examples. In solving these tasks, the system composed 16 new DSL components, and discovered multiple higher-order functions, such as `filter` (f_0 in Figure **??**). Each round of memory consolidation built on components discovered in earlier sleep cycles — for example the agent first learns `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^{th} largest element of a list, which it finally uses to learn how to sort a list of numbers (Figure **??**). This incremental, modular learning of deep hierarchies of DSL components occurs because of the alternation between code writing and code refactoring.

3.1.2 Text Editing

Synthesizing programs that edit text is a classic problem in the programming languages and AI literatures [21], 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 [1] 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 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.

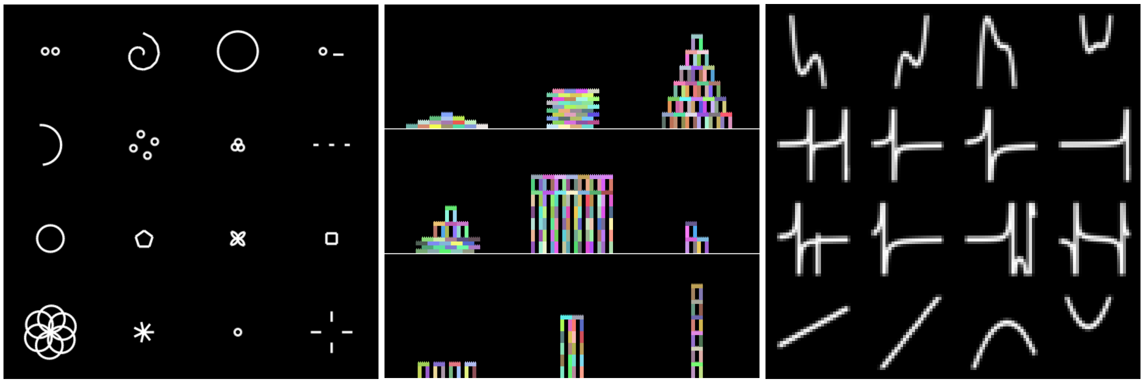


Figure 9: Three domains where the agent infers a program from visual input. **Left**: 16 (out of 145) LOGO graphics tasks. Agent writes a program controlling a 'pen' that draws the target picture. **Middle**: 9 (out of 112) tower building tasks. Agent writes a program controlling a 'hand' that builds 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.

3.2 Programs from visual input

We consider three domains where the agent must infer a program from an image (Figure 9).

3.2.1 Programs that make plans and take actions

We apply DreamCoder to two domains where the agent plans a series of actions in order to draw a picture (Sec. 3.2.2) or build a tower out of blocks (Sec. 3.2.3). For drawing pictures, the agent controls a 'pen' that it can pick up or place down while moving across a canvas. For building towers, the agent controls a 'hand' that it moves through a simulated world while placing down differently sized blocks. For each of these domains, the agent observes a target picture or block tower, and must either draw the picture or build the tower. The recognition model is a CNN that observes an image of the target picture or tower. The system is initially provided with two control flow operators: `loop` (a 'for' loop) and `get/set`, which saves and then restores the current state of the pen or hand.

3.2.2 Programs that draw pictures

Procedural visual concepts are studied across AI and cognitive science — Bongard problems [?], Raven's progressive matrices [?], and Lake et al.'s BPL model of omniglot [20] are prominent examples. Here we take inspiration from LOGO Turtle graphics [38], and give our agent the ability to control a pen, along with arithmetic operations on angles and distances and simple control flow primitives. We task it with drawing a small corpus of images starting from this basis.

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 10, left). It additionally learns more abstract visual relationships, like rotational symmetry, which it models by incorporating a higher-order function into its DSL (Figure 10, right). This abstraction comes from jointly compressing programs for all of the training images into its DSL.

What does DreamCoder dream of? Prior to learning samples from the DSL are simple and largely unstructured (Figure 11, left). After training the samples become richly structured (Figure 11, right), compositionally recombining latent building blocks and motifs acquired from the training data. This offers a 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.

3.2.3 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

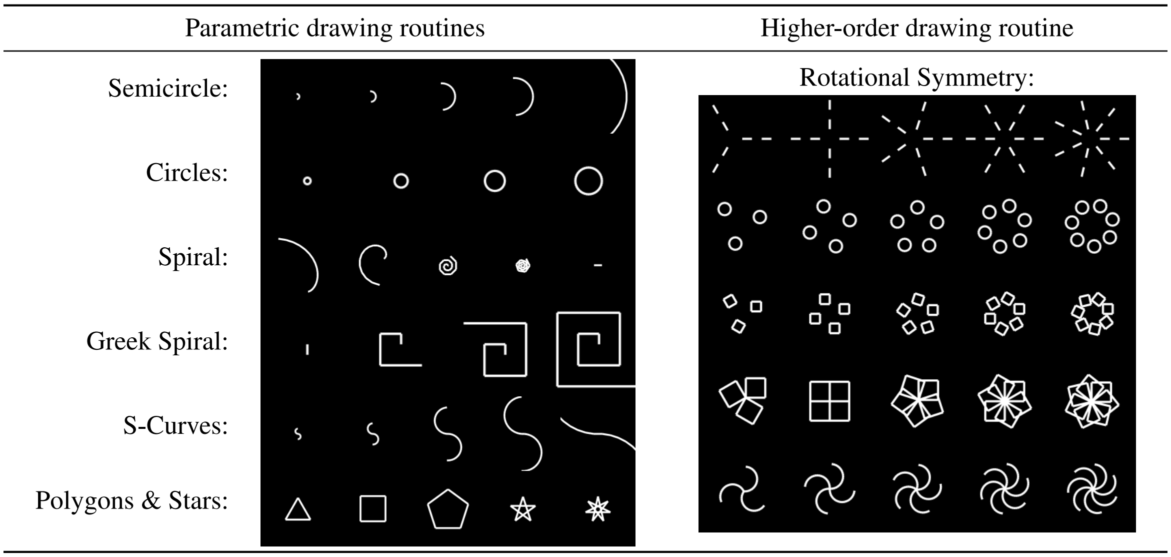


Figure 10: Example primitives learned by DreamCoder when trained on tasks in Figure **??**. Agent learns parametric routines for drawing families of curves (left) as well as subroutines that take entire programs as input (right). Each row of images on the left is the same code executed with different parameters. Each image on the right is the same code executed with different parameters and with a different subprogram provided as input.

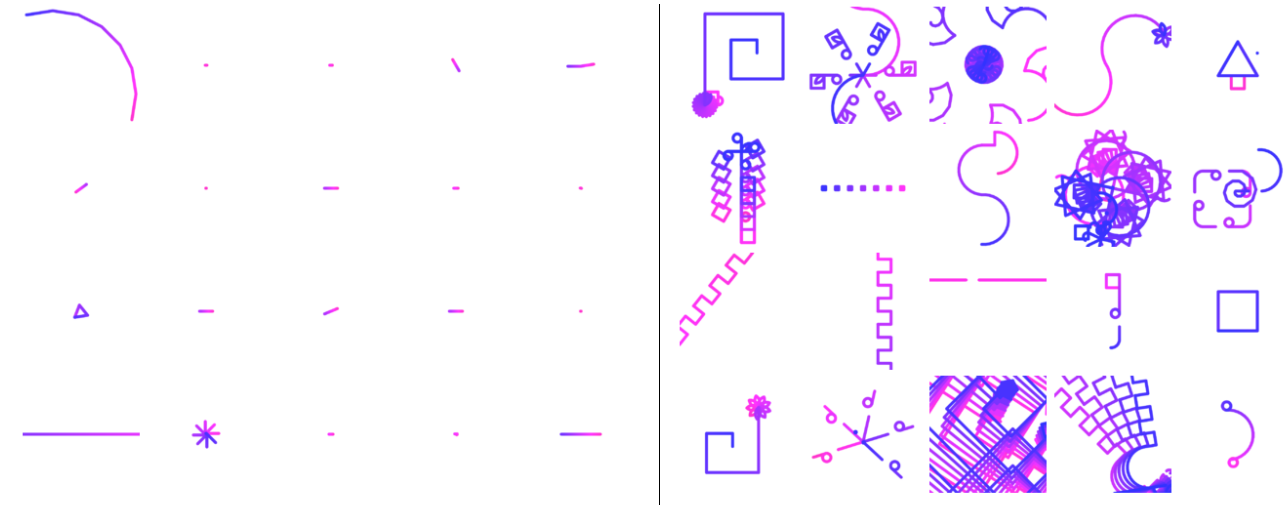


Figure 11: Dreams, or samples, from the DSL before (left) and after (right) training on tasks in Figure **??**. Blue: where the agent started drawing. Pink: where the agent ended drawing.

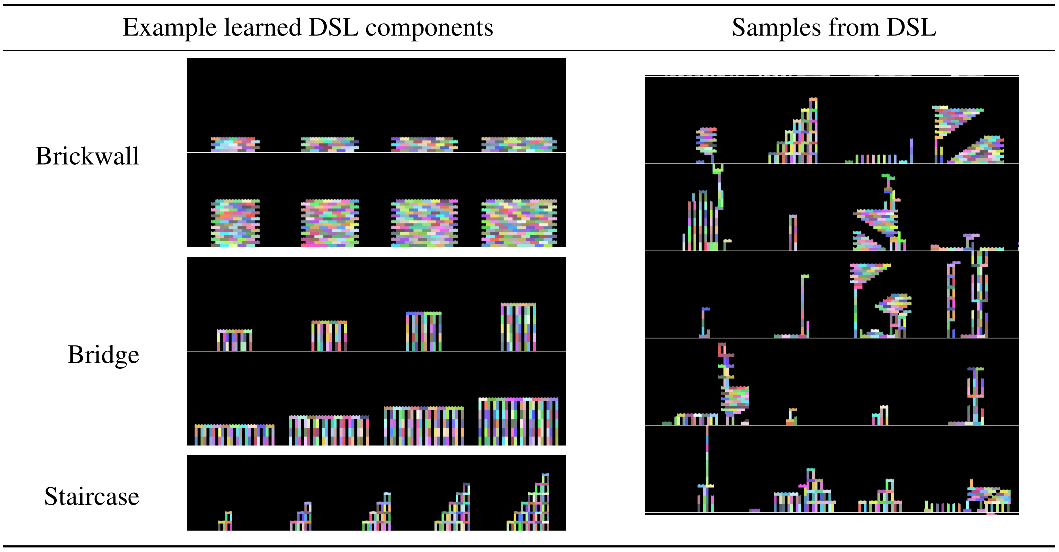


Figure 12: **Left**: Three (out of 19) learned DSL components for building towers out of Lego-style blocks. These components act like parametric options [**?**], giving higher-level building blocks that the agent can use to plan. **Right**: 16 random samples, or 'dreams', from learned DSL.

starts with the same control flow primitives as with LOGO graphics, and learns parametric 'options' for building blocks towers (Figure 12), inferring concepts like arches, staircases, bridges and brick walls.

3.2.4 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. 9, 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 [3].

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.

3.3 Quantitative Results on Held-Out Tasks

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

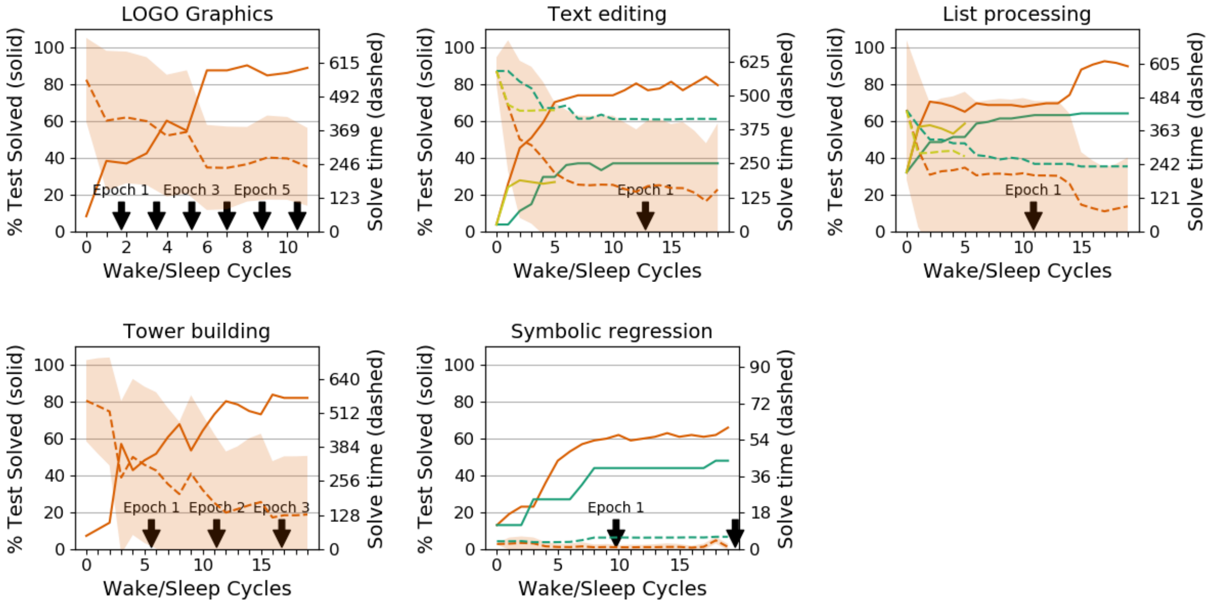


Figure 13

4 Discussion

4.1 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 [35, 33, 15, 36]. All existing systems, including ours, fall far short of this dream, and it is unclear (and we believe unlikely) that this dream could ever be fully realized. How far can we push in this direction? “Learning from scratch” is subjective, but a reasonable starting point is the set of primitives provided in 1959 Lisp [23]: these include conditionals, recursion, arithmetic, and the list operators `cons`, `car`, `cdr`, and `nil`. A basic first goal is to start with these primitives, and then recover a DSL that more closely resembles modern functional languages like Haskell and OCaml. Recall (Sec. 3.1) that we initially provided our system with functional programming routines like `map` and `fold`.

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

After running for 93 hours on 64 CPUs, our algorithm solves these 22 exercises, along the way assembling a DSL with a modern repertoire of functional programming idioms and subroutines, including `map`, `fold`, `zip`, `unfold`, `index`, `length`, and arithmetic operations like building lists of natural numbers between an interval (see Appendix A.8).

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.

4.2 Program Induction as part of the generic AI toolkit

Our aim with DREAMCODER is both to show how learning can enable better program synthesis, and to show how better program synthesis can be more broadly useful for AI. How could a program induction approach to AI become broadly useful across a diverse spectrum of AI problems — planning, natural language understanding, machine vision, causal inference, and more?

References

[1] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.

[2] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *ICLR*, 2016.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. 2006.

[4] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[5] Eyal Dechter, Jon Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. Bootstrap learning via modular concept discovery. In *IJCAI*, 2013.

[6] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. *ICML*, 2017.

[7] Yadin Dudai, Avi Karni, and Jan Born. The consolidation and transformation of memory. *Neuron*, 88(1):20 – 32, 2015.

[8] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Library learning for neurally-guided bayesian program induction. In *NIPS*, 2017.

[9] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. In *NeurIPS*, 2018.

[10] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B Tenenbaum. Learning to infer graphics programs from hand-drawn images. *NIPS*, 2018.

[11] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, 2015.

[12] Magdalena J Fosse, Roar Fosse, J Allan Hobson, and Robert J Stickgold. Dreaming and episodic memory: a functional dissociation? *Journal of cognitive neuroscience*, 15(1):1–9, 2003.

[13] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

[14] Geoffrey E Hinton, Peter Dayan, Brendan J Frey, and Radford M Neal. The ”wake-sleep” algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995.

[15] Marcus Hutter. *Universal artificial intelligence: Sequential decisions based on algorithmic probability*. Springer Science & Business Media, 2004.

[16] Irvin Hwang, Andreas Stuhlmüller, and Noah D. Goodman. Inducing probabilistic programs by bayesian program merging. *CoRR*, abs/1110.5667, 2011.

[17] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *ICLR*, 2018.

[18] John R. Koza. *Genetic programming - on the programming of computers by means of natural selection*. MIT Press, 1993.

[19] J.D. Lafferty. *A Derivation of the Inside-outside Algorithm from the EM Algorithm*. Research report.

[20] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[21] Tessa Lau. *Programming by demonstration: a machine learning approach*. PhD thesis, 2001.

[22] Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *ICML*, 2010.

[23] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[24] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *ICML*, pages 187–195, 2013.

[25] Tom M Mitchell. Version spaces: A candidate elimination approach to rule learning. In *Proceedings of the 5th international joint conference on Artificial intelligence-Volume 1*, pages 305–310. Morgan Kaufmann Publishers Inc., 1977.

[26] Stephen H Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.

[27] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, volume 50, pages 619–630. ACM, 2015.

[28] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[29] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.

[30] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[31] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 305–316. ACM, 2013.

[32] Ute Schmid and Emanuel Kitzelmann. Inductive rule learning on the knowledge level. *Cognitive Systems Research*, 12(3-4):237–248, 2011.

[33] Jürgen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.

[34] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, 2008.

[35] Ray J Solomonoff. A formal theory of inductive inference. *Information and control*, 7(1):1–22, 1964.

[36] Ray J Solomonoff. A system for incremental learning based on algorithmic probability. Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition, 1989.

[37] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pages 212–223. Springer, 2002.

[38] David D. Thornburg. Friends of the turtle. *Compute!*, March 1983.

[39] T. Ullman, N. D. Goodman, and J. B. Tenenbaum. Theory learning as stochastic search in the language of thought. *Cognitive Development*, 27(4):455–480, 2012.

[40] Maksym Zavershynskyi, Alex Skidanov, and Illia Polosukhin. Naps: Natural program synthesis dataset. In *ICML*, 2018.

A Appendix

A.1 Generative model over programs

Algorithm 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.

Algorithm 2 Generative model over programs

```
function sample($\mathcal{D}, \theta, \tau$):
Input: DSL $(\mathcal{D}, \theta)$, type $\tau$
Output: a program whose type unifies with $\tau$
return sample$'(\mathcal{D}, \theta, \varnothing, \tau)$

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

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

A.2 Enumerative program search

Our current implementation of DREAMCODER takes the simple and generic strategy of enumerating programs in descending order of their probability under either $(\mathcal{D}, \theta)$ or $Q(p|x)$. Algorithm 2 and 4 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 4, 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 Δ, then programs with description length between Δ and 2Δ, then 2Δ to 3Δ, etc., until a timeout is reached. This is similar in spirit to iterative deepening depth first search [30].

A.3 Refactoring code with version spaces

Formally, a version space is either:

- A deBuijn[3] index: written $\$i$, where i is a natural number
- An abstraction: written λv, where v is a version space
- An application: written $(f\ x)$, where both f and x are version spaces
- A union: $\uplus V$, where V is a set of version spaces
- The empty set, $\varnothing$
- The set of all λ-calculus expressions, Λ

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

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

$$
\begin{array}{lll}
[\![\$i]\!] = \{\$i\} & [\![\lambda v]\!] = \{\lambda e : e \in [\![v]\!]\} & [\![(v_1\ v_2)]\!] = \{(e_1\ e_2) : e_1 \in [\![v_1]\!],\ e_2 \in [\![v_2]\!]\} \\
[\![\uplus V]\!] = \{e : v \in V,\ e \in [\![v]\!]\} & [\![\varnothing]\!] = \varnothing & [\![\Lambda]\!] = \Lambda
\end{array}
$$

Version spaces also support efficient membership checking, which we write as $e \in [\![v]\!]$. 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:

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

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

Algorithm 3 Parallel enumerative program search algorithm

```
function enumerate($\mu, T$, CPUs):
Input: Distribution over programs $\mu$, timeout $T$, CPU count
Output: stream of programs in approximately descending order of probability under $\mu$
Hyperparameter: nat increase rate $\Delta$    ▷ We set $\Delta = 1.5$
lowerBound← 0
while total elapsed time $< T$ do
    heap←newMaxHeap()    ▷ Heap for best-first search
    heap.insert(priority $= 0$, value $=$ empty syntax tree)    ▷ Initialize heap with start state of search space
    while $0 < |\text{heap}| \leq 10 \times$ CPUs do    ▷ Each CPU will get approximately 10 jobs (a partial program)
        priority, partialProgram ← heap.popMaximum()
        if partialProgram is finished then    ▷ Nothing more to fill in in the syntax tree
            if lowerBound $\leq -$priority $<$ lowerBound $+ \Delta$ then
                yield partialProgram
            end if
        else
            for child∈children(partialProgram) do    ▷ children($\cdot$) fills in next random choice in syntax tree.
                if $-\log \mu$(child) $<$ lowerBound $+ \Delta$ then    ▷ Child's description length small enough
                    heap.insert(priority $= \log \mu$(child), value $=$ child)
                end if
            end for
        end if
    end while
    yield from ParallelMap$_{\text{CPUs}}$(depthFirst($\mu, T -$ elapsed time, lowerBound, $\cdot$), heap.values())
    lowerBound ← lowerBound $+ \Delta$    ▷ Push up lower bound on MDL by $\Delta$
end while

function depthFirst($\mu, T$, lowerBound, partialProgram):    ▷ Each worker does a depth first search. Enumerates completions of partialProgram whose MDL is between lowerBound and lowerBound $+ \Delta$
stack←[partialProgram]
while total elapsed time $< T$ and stack is not empty do
    partialProgram←stack.pop()
    if partialProgram is finished then
        if lowerBound $\leq -\log \mu$(partialProgram) $<$ lowerBound $+ \Delta$ then
            yield partialProgram
        end if
    else
        for child $\in$ children(partialProgram) do
            if $-\log \mu$(child) $<$ lowerBound $+ \Delta$ then    ▷ Child's description length small enough
                stack.push(child)
            end if
        end for
    end if
end while
```

$$\textsc{refactor}'(e|\mathcal{D}) = e, \text{ if } e \text{ is a leaf} \qquad \textsc{refactor}'(\lambda b|\mathcal{D}) = \lambda\textsc{refactor}(b|\mathcal{D})$$

$$\textsc{refactor}'(f\ x|\mathcal{D}) = \textsc{refactor}(f|\mathcal{D})\ \textsc{refactor}(x|\mathcal{D}) \qquad \textsc{refactor}'(\uplus V|\mathcal{D}) = \operatorname*{arg\,min}_{e\in\{\textsc{refactor}(v|\mathcal{D})\ :\ v\in V\}} \text{size}(e|\mathcal{D})$$

where 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, $\textsc{refactor}(v|\mathcal{D})$ calculates $\arg\min_{p\in[\![v]\!]} \text{size}(p|\mathcal{D})$.

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

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

where

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

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

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

$$\downarrow_c^k \$i = \$i, \text{ when } i < c$$
$$\downarrow_c^k \$i = \$(i-k), \text{ when } i \ge c+k$$
$$\downarrow_c^k \$i = \varnothing, \text{ when } c \le i < c+k$$
$$\downarrow_c^k \lambda b = \lambda \downarrow_{c+1}^k b$$
$$\downarrow_c^k (f\ x) = (\downarrow_c^k f\ \downarrow_c^k x)$$
$$\downarrow_c^k \uplus V = \uplus\left\{\downarrow_c^k v \mid v \in V\right\}$$
$$\downarrow_c^k v = v, \text{ when } v \text{ is a primitive or } \varnothing \text{ or } \Lambda$$

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

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

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

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

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

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

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

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

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

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

- If u is of the form λa, then $S_n(u) \subset \{v \mapsto \lambda b \mid (v \mapsto b) \in S_{n+1}(a)\}$. Let $v \mapsto \lambda b \in S_n(u)$. By induction for every $v' \in v$ and $b' \in b$ we have $\uparrow_{n+1}^{-1}\ [\$n \mapsto \uparrow^{2+n}\ v']b' \in a$, which we can rewrite to $\uparrow_n^{-1}\ [\$n \mapsto \uparrow^{1+n}\ v']\lambda b' \in \lambda a = u$.

- If u is of the form $(f\ x)$ then then $S_n(u) \subset \{v_f \cap v_x \mapsto (b_f\ b_x) \mid (v_f \mapsto b_f) \in S_n(f),\ (v_x \mapsto b_x) \in S_n(x)\}$. Pick $v' \in v_f \cap v_x$ arbitrarily. By induction for every $v'_f \in v_f$, $v'_x \in v_x$, $b'_f \in b_f$, $b'_x \in b_x$ we have $\uparrow_n^{-1}\ [\$n \mapsto \uparrow^{1+n}\ v'_f]b'_f \in f$ and $\uparrow_n^{-1}\ [\$n \mapsto \uparrow^{1+n}\ v'_x]b'_x \in x$. Combining these facts gives $\uparrow_n^{-1}\ [\$n \mapsto \uparrow^{1+n}\ v'](b'_f\ b'_x) \in (f\ x) = u$.

- If u is of the form $\uplus U$ then pick $(v \mapsto b) \in S_n(u)$ arbitrarily. By the definition of S_n there is a z such that $(v \mapsto b) \in S_n(z)$, and the theorem holds immediately by induction.

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

□

Theorem 2. ***Completeness of** S_n.*
If there exists programs v' and b', and a version space u, such that $\uparrow_n^{-1}\ [\$n \mapsto \uparrow^{1+n}\ v']b' \in u$, then there also exists $(v \mapsto b) \in S_n(u)$ such that $v' \in v$ and $b' \in b$.

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

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

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

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

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

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

- If u is a primitive then, because b' is not $\$n$, we have $\uparrow_n^{-1}\ b' = u$, and so $b' = u$. By the definition of S_n we have $(\Lambda \mapsto u) \in S_n(u)$ completing this inductive step because $v' \in \Lambda$ and $b' = u$.

- If u is of the form λa then, because of the assumption that $b' \neq \$n$, we know that b' is of the form $\lambda c'$ and that $\lambda \uparrow_{n+1}^{-1} [\$(n+1) \mapsto \uparrow^{2+n} v']c' \in \lambda a$. By induction this means that there is a $(v \mapsto c) \in S_{n+1}(a)$ satisfying $v' \in v$ and $c' \in c$. By the definition of S_n we also know that $(v \mapsto \lambda c) \in S_n(u)$, completing this inductive step because $b' = \lambda c' \in \lambda c$.

- If u is of the form $(f\ x)$ then, because of the assumption that $b' \neq \$n$, we know that b' is of the form $(b'_f\ b'_x)$ and that both $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']b'_f \in f$ and $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']b'_x \in x$. Invoking the inductive hypothesis twice gives a $(v_f \mapsto b_f) \in S_n(f)$ satisfying $v' \in v_f$, $b'_f \in b_f$ and a $(v_x \mapsto b_x) \in S_n(x)$ satisfying $v' \in v_x$, $b'_x \in b_x$. By the definition of S_n we know that $(v_f \cap v_x \mapsto b_f\ b_x) \in S_n(u)$ completing the inductive step because v' is guaranteed to be in both v_f and v_x and we know that $b' = (b'_f\ b'_x) \in (b_f\ b_x)$.

- If u is of the form $\uplus U$ then there must be a $z \in U$ such that $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']b' \in z$. By induction there is a $(v \mapsto b) \in S_n(z)$ such that $v' \in v$ and $b' \in v$. 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 $\varnothing$ or Λ then the theorem holds vacuously.

□

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

Theorem 3. ***Consistency of*** $I\beta'$**.**
If $p \in \llbracket I\beta'(u) \rrbracket$ then there exists $p' \in \llbracket u \rrbracket$ such that $p \longrightarrow p'$.

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

- Case $p = (\lambda b')v'$ where $v' \in v$, $b' \in b$, and $v \mapsto b \in S_0(u)$: From the definition of β-reduction we know that $p \longrightarrow \uparrow^{-1} [\$0 \mapsto \uparrow^1 v']b'$. From the consistency of S_n we know that $\uparrow^{-1} [\$0 \mapsto \uparrow^1 v']b' \in u$. Identify $p' = \uparrow^{-1} [\$0 \mapsto \uparrow^1 v']b'$.

- Case $u = \lambda b$ and $p = \lambda b'$ where $b' \in \llbracket I\beta'(b) \rrbracket$: By induction there exists $b'' \in \llbracket b \rrbracket$ such that $b' \longrightarrow b''$. So $p \longrightarrow \lambda b''$. But $\lambda b'' \in \llbracket \lambda b \rrbracket = \llbracket u \rrbracket$, so identify $p' = \lambda b''$.

- Case $u = (f\ x)$ and $p = (f'\ x')$ where $f' \in \llbracket I\beta'(f) \rrbracket$ and $x' \in \llbracket x \rrbracket$: By induction there exists $f'' \in \llbracket f \rrbracket$ such that $f' \longrightarrow f''$. So $(f'\ x') \longrightarrow (f''\ x')$. But $(f''\ x') \in \llbracket (f\ x) \rrbracket = \llbracket u \rrbracket$, so identify $p' = (f''\ x')$.

- Case $u = (f\ x)$ and $p = (f'\ x')$ where $x' \in \llbracket I\beta'(x) \rrbracket$ and $f' \in \llbracket f \rrbracket$: Symmetric to the previous case.

- Case $u = \uplus U$ and $p \in \llbracket I\beta'(u') \rrbracket$ where $u' \in U$: By induction there is a $p' \in \llbracket u' \rrbracket$ satisfying $p' \longrightarrow p$. But $\llbracket u' \rrbracket \subseteq \llbracket u \rrbracket$, so also $p' \in \llbracket u \rrbracket$.

- Case u is an index, primitive, $\varnothing$, or Λ: The theorem holds vacuously.

□

Theorem 4. ***Completeness of*** $I\beta'$**.**
Let $p \longrightarrow p'$ and $p' \in \llbracket u \rrbracket$. Then $p \in \llbracket I\beta'(u) \rrbracket$.

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

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

- Case $p = (\lambda b')v'$ and $p' = \uparrow^{-1} [\$0 \mapsto \uparrow^1 v']b'$: Using the fact that $\uparrow^{-1} [\$0 \mapsto \uparrow^1 v']b' \in \llbracket u \rrbracket$, we can invoke the completeness of S_n to construct a $(v \mapsto b) \in S_0(u)$ such that $v' \in \llbracket v \rrbracket$ and $b' \in \llbracket b \rrbracket$. Combine these facts with the definition of $I\beta'$ to get $p = (\lambda b')v' \in \llbracket (\lambda b)v \rrbracket \subseteq I\beta'(u)$.

- 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 $p' = (f'\ x) \in [\![u]\!]$ and by assumption $u \neq \uplus V$ we know that $u = (a\ b)$ where $f' \in [\![a]\!]$ and $x \in [\![b]\!]$. By induction on a we know $f \in [\![I\beta'(a)]\!]$. Therefore $p = (f\ x) \in [\![(I\beta'(a)\ b)]\!] \subseteq [\![I\beta'((a\ b))]\!] \subseteq [\![I\beta'(u)]\!]$.
- 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 $p \underbrace{\longrightarrow q \longrightarrow \cdots \longrightarrow}_{\leq n \text{ times}} p'$ if and only if $p \in [\![I\beta_n(p')]\!]$.*

Proof. Induction on n.

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

If $p \underbrace{\longrightarrow q \longrightarrow \cdots \longrightarrow}_{\leq n \text{ times}} p'$ then $q \underbrace{\longrightarrow \cdots \longrightarrow}_{\leq n-1 \text{ times}} p'$; induction on n gives $q \in [\![I\beta_{n-1}(p')]\!]$. Combined with $p \longrightarrow q$ we can invoke the completeness of $I\beta'$ to get $p \in [\![I\beta'(I\beta_{n-1}(p'))]\!] \subset [\![I\beta_n(p')]\!]$.

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

A.3.1 Computational complexity of DSL learning

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

For each version space v we calculate a *beam*, which is a function from a DSL $\mathcal{D}$ to a shortest program in $[\![v]\!]$ 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 $B \to \infty$ we recover the exact behavior of REFACTOR. The following recursive equations define how we calculate these beams; the set 'proposals' is defined in line 10 of Algorithm 1, and $\mathcal{D}$ is the current DSL:

$$\text{beam}_v(\mathcal{D}') = \begin{cases} \text{if } \mathcal{D}' \in \text{dom}(b_v): & b_v(\mathcal{D}') \\ \text{if } \mathcal{D}' \not\in \text{dom}(b_v): & \text{REFACTOR}(v, \mathcal{D}) \end{cases}$$

$$b_v = \text{the } B \text{ pairs } (\mathcal{D}' \mapsto p) \text{ in } b'_v \text{ where the syntax tree of } p \text{ is smallest}$$

$$b'_v(\mathcal{D}') = \begin{cases} \text{if } \mathcal{D}' \in \text{proposals and } e \in \mathcal{D}' \text{ and } e \in v\text{: } e \\ \text{otherwise if } v \text{ is a primitive or index:} v \text{otherwise if } v = \lambda b\text{: } \lambda \text{beam}_b(\mathcal{D}') \\ \text{otherwise if } v = (f\ x)\text{: } (\text{beam}_f(\mathcal{D}')\ \text{beam}_x \mathcal{D}')) \\ \text{otherwise if } v = \uplus V\text{: } \arg\min_{e \in \left\{b'_{v'}(\mathcal{D}')\ :\ v' \in V\right\}} \text{size}(e|\mathcal{D}') \end{cases}$$

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

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

A.4 Estimating the continuous parameters θ of a DSL

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

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

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

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

We write $c(e,p)$ to mean the number of times that primitive e was used in program p; $c(p) = \sum_{e\in\mathcal{D}} c(e,p)$ to mean the total number of primitives used in program p; $c(\tau,p)$ to mean the number of times that type τ was the input to sample in Algorithm 2 while sampling program p. Jensen's inequality gives a lower bound on the likelihood:

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

where we have defined

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

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

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

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

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

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

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

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

A.5 Recognition model training

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

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

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 $\mathcal{L}^{\text{MAP}}_{\text{Fantasy}}$ 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 $\mathcal{L}^{\text{MAP}}_{\text{Fantasy}}$, 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[4] 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 $k\in\{1,2,\cdots,A\}$. The output $Q_{ijk}(x)$ controls the probability of sampling primitive j given that i is the parent node in the syntax tree and we are sampling the k^{th} argument. Algorithm 4 specifies a procedure for drawing samples from $Q(\cdot|X)$.

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 $\mathcal{L}^{\text{posterior}}$.

[4] The arity of a function is the number of arguments that it takes as input.

Algorithm 4 Drawing from distribution over programs predicted by recognition model. Compare w/ Algorithm 2

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

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

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

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

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

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

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

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

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

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

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

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

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 Λ 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 $p^* \in \mathcal{B}_x$. □

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

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

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

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

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.

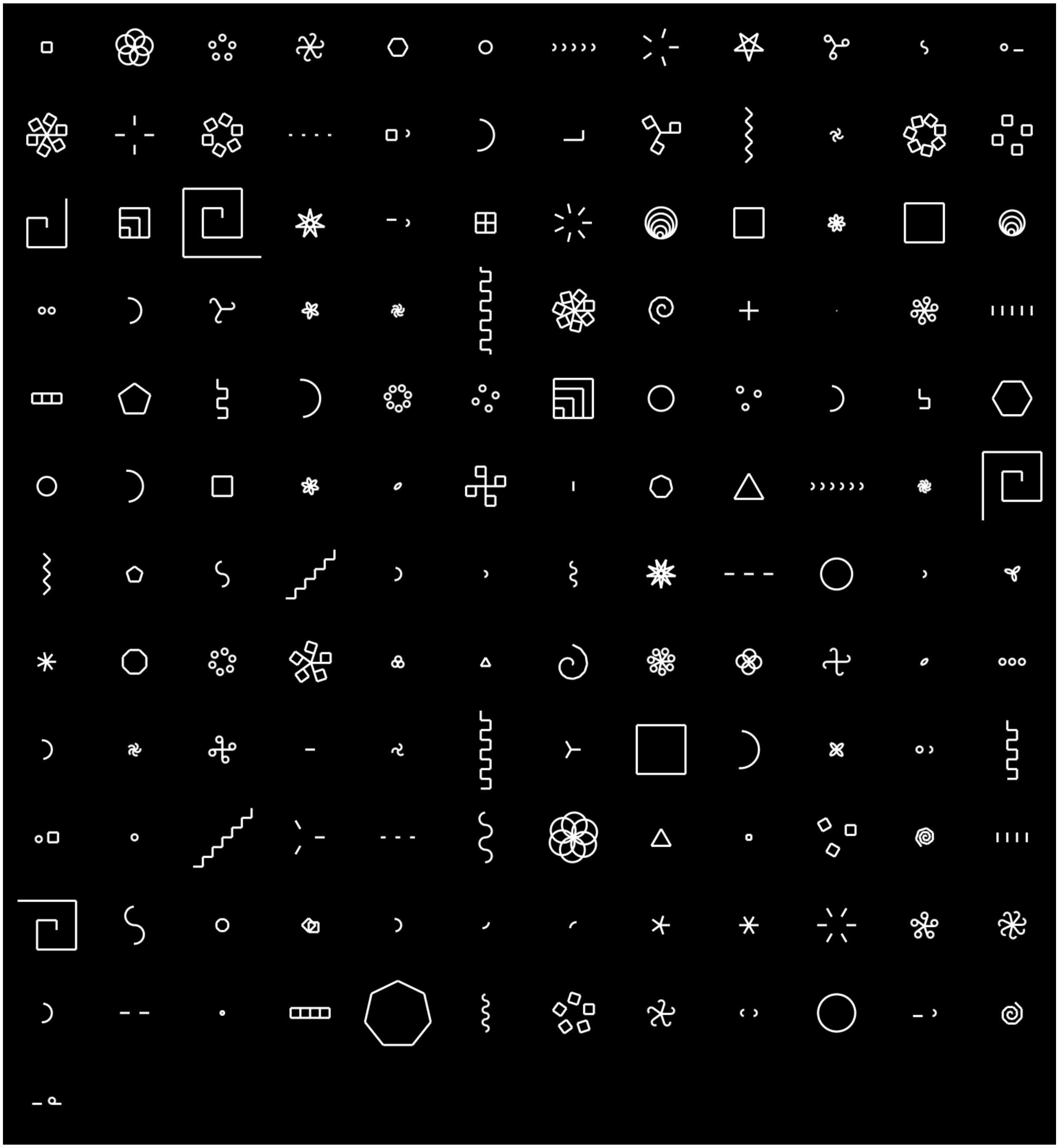


Figure 14: Full set of LOGO graphics tasks that we apply our system to

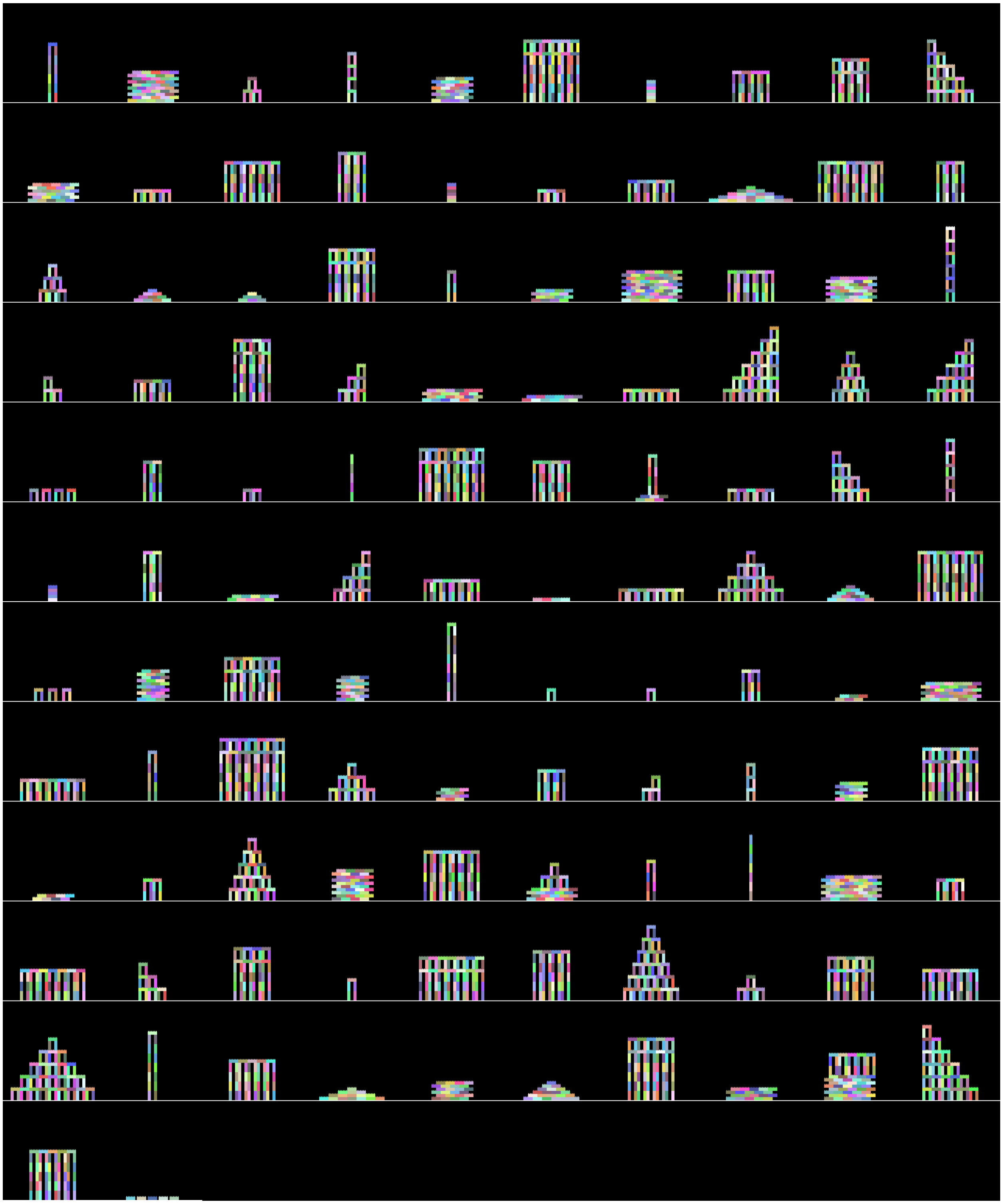


Figure 15: Full set of tower building tasks that we apply our system to

A.6 Full set of LOGO tasks

A.7 Full set of tower tasks

A.8 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 [23].[5] 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 16 shows the full set of tasks and the learned DSL.

[5] 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	DSL
[1 9]→2 [5 3 8]→3 $f(\ell) = (f_4\ \ell)$ [true false]→2 [false false false]→3 $f(\ell) = (f_4\ \ell)$ [2 1 4]→[2 1 4 0] [9 8]→[9 8 0] $f(\ell) = (f_2$ cons ℓ (cons 0 nil)) [2 1 4]→[2 1] [9 8]→[9] $f(\ell) = (f_0$ (λ (z) (empty? (cdr z))) car cdr ℓ) [2 5 6 0 6]→19 [9 2 7 6 3]→27 $f(\ell) = (f_2$ + ℓ 0) [4 2 6 4]→[8 4 12 8] [2 3 0 7]→[4 6 0 14] $f(\ell) = (f_3$ (λ (x) (+ x x)) ℓ) [4 2 6 4]→[-4 -2 -6 -4] [2 3 0 7]→[-2 -3 -0 -7] $f(\ell) = (f_3$ (- 0) ℓ) [4 2 6 4]→[5 3 7 5] [2 3 0 7]→[3 4 1 8] $f(\ell) = (f_3$ (+ 1) ℓ) [1 5 2 9]→[1 2] [3 8 1 3 1 2]→[3 1 1] $f(\ell) = (f_0$ empty? car (λ (l) (cdr (cdr l))) ℓ) 3→[0 1 2] 2→[0 1] $f(n) = (f_5$ (λ (x) x) 0 n) 3→[0 1 2 3] 2→[0 1 2] $f(n) = (f_5$ (λ (x) x) 0 (+ 1 n))	f_0(p,f,n,x) = (if (p x) nil (cons (f x) (f_0 (n x)))) (f_0: *unfold*) f_1(i,l) = (if (= i 0) (car l) (f_1 (- i 1) (cdr l)))) (f_1: *index*) f_2(f,l,x) = (if (empty? l) x (f (car l) (f_2 (cdr l)))) (f_2: *fold*) f_3(f,l) = (f_2 nil l (λ (x a) (cons (f x) a))) (f_3: *map*) $f_4(\ell)$ = (if (empty? ℓ) 0 (+ 1 (f_4 (cdr ℓ))))) (f_4: *length*) f_5(f,m,n) = (f_0 (= m) f (+ 1) n) (f_5: *generalization of range*)

Figure 16: Bootstrapping a standard library of functional programming routines, starting from recursion along with primitive operations found in 1959 Lisp.