1 Introduction: Expertise

Human learners acquire expertise in a wide range of domains: some of us become experts in calculus, or cooking, or biology, music, tennis, or software engineering, to name just a few, and every child develops expertise in natural language, intuitive physics [**?**], intuitive psychology (theory-of-mind), and motor control. This picture contrasts sharply with the current state of machine intelligence, where a machine is built to be an expert in a single domain, like boardgames [**?**], medical diagnosis [**?**, **?**], theorem proving [**?**], or visual object recognition [**?**]. Thus an outstanding challenge in the long-term program of building more humanlike machines is to develop an algorithm that, like people, autonomously acquires expertise across many different kinds of domains.

We contribute a model of the development of expertise that combines two key ingredients. First, an expert needs a sufficiently expressive knowledge representation. Following a long tradition in cognitive science and AI, we represent knowledge as programs, which prior work has used to represent expertise in recognition and generation of handwriting and speech [23], intuitive theories (of kinship, taxonomy, etc.) [44], and natural language grammar and semantics [37, 32]. Second, atop this program representation, experts possess two kinds of domain expertise. They have at their disposal a powerful, yet specialized, repertoire of concepts and abstractions: e.g., in architecture, these are concepts like 'arch' or 'foundation'; in software engineering these are libraries of code and domain-specific languages (DSLs). Here our model represents knowledge as programs, and so we identify this kind of expertise with a DSL. Experts also have knowledge of when and how to deploy these domain-specific concepts efficiently when solving new problems: e.g., mathematicians efficiently search the space of proofs, intuiting which lemmas are appropriate when; expert chefs intuit which compositions of ingredients are likely tasty, before they actually start cooking. For our model, this aspect of expertise corresponds to the ability to quickly assemble new, useful programs out of its DSL.

We integrate these ideas into a model called DreamCoder which acquires expertise through a novel kind of wake/sleep or 'dream' learning. The model iterates through wake cycles – where it solves problems by writing programs – and a pair of sleep cycles: a sleep cycle that grows its DSL by replaying experiences from waking and consolidating them into new abstractions, and a sleep cycle that improves its knowledge of how to write programs by training a neural network on replayed experiences as well as 'dreams', or samples, from its DSL.

DreamCoder builds on multiple generations of AI research, going back to the 1960's [40] when program-learning was proposed as a paradigm for general AI. Broadly speaking recent work has either developed neural approaches for learning to efficiently deploy a fixed DSL [8, 2, 20, **?**], or developed symbolic approaches for representing and searching through spaces of programs [15, 39, 21]. We were motivated by approaches that learn or grow the DSL [6, 26, 41, 18, 42]. Our goal with DreamCoder is to show that the combination of neurally-guided search and DSL learning is a uniquely powerful way of building systems that, like human learners, autonomously acquire the expertise needed to navigate a new domain of problems.

2 Introduction: AI/PL/dreaming

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 [40] and cuts to the core of 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 [23]; 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.) [44] and natural language grammar [37], 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 [**?**, 11], transforming text [15] and numerical sequences [2], and reasoning over common sense knowledge bases [29]. 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 [8, 2, 20], 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.

Our system is called DreamCoder because it acquires these two kinds of expertise through a novel kind of wake/sleep or 'dream' learning [16], iterating through a wake cycle – where it solves problems by writing programs – and a pair of sleep cycles: a sleep cycle that grows the DSL by replaying experiences from waking and consolidating them into new abstractions (modeled as fragments of code), and a sleep cycle that improves its knowledge of how to write programs by training a neural network based on replayed experiences as well as 'dreams', or samples, from the DSL.

DreamCoder builds on two lines of prior work. One line of work considers the problem of learning new concepts, abstractions, or 'options' from experience [6, 26, 41, 18, 42], while the other line of work considers the problem of learning how to deploy those concepts efficiently [8, 2, 20]. 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 solve a new class of problems.

3 Introduction: NIPS

Much of everyday human thinking and learning can be understood in terms of program induction: constructing a procedure that maps inputs to desired outputs, based on observing example input-output pairs. People can induce programs flexibly across many different domains, often from just one or a few examples. For instance, if shown that a text-editing program should map "Jane Morris Goodall" to "J. M. Goodall", we can guess it maps "Richard Erskine Leakey" to "R. E. Leakey"; if instead the first input mapped to "Dr. Jane" or "Goodall, Jane", we might guess the latter should map to "Dr. Richard" or "Leakey, Richard", respectively.

The FlashFill system [15] embedded in Microsoft Excel solves problems such as these and is probably the best known practical program-induction algorithm, but researchers in programming languages and AI have had successes in many domains, such as handwriting recognition and generation [23], procedural graphics [11, 14], question answering [19] and robot motion planning [7]. These systems work in different ways, but most hinge upon a carefully engineered **Domain Specific Language (DSL)**. This is especially true for systems like FlashFill that induce a wide range of programs very quickly, in a few seconds or less. DSLs constrain the search over programs with strong prior knowledge in the form of a restricted inventory of programming primitives finely tuned to the domain: for text editing, these are operations like appending and splitting on characters.

In this work, we consider the problem of building agents that learn to solve program induction tasks, and also the problem of acquiring the prior knowledge necessary to quickly solve these tasks in a new domain. Representative problems in three domains are shown in Table 3. Our solution is an algorithm that grows or boostraps a DSL while jointly training a neural network to help write programs in the increasingly rich DSL.

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 [39] or genetic programming [21], 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.

Our algorithm takes inspiration from several ways that skilled human programmers have learned to code: Skilled coders build libraries of reusable subroutines that are shared across related programming tasks, and can be composed to generate increasingly complex and powerful subroutines. In text editing, a good library should support routines for splitting on characters, but also specialize these routines to split on particular characters such as spaces or commas that are frequently used to delimit substrings across tasks. Skilled coders also learn to recognize what kinds of programming idioms and library routines would be useful for solving the task at hand, even if they cannot instantly work out the details. In text editing, one might learn that if outputs are consistently shorter than inputs, removing characters is likely to be part of the solution; if every output contains a constant substring (e.g., "Dr."), inserting or appending that constant string is likely to be a subroutine.

Our algorithm is called DREAMCODER because it is based on a novel kind of "wake-sleep" learning (c.f. [16]), iterating between "wake" and "sleep" phases to achieve three goals: finding programs that solve tasks; creating a

	List Functions		Text Editing	Symbolic Regression
Programs & Tasks	[7 2 3]→[7 3] [1 2 3 4]→[3 4] [4 3 2 1]→[4 3] $f(\ell)$=(f1 ℓ (λ (x) (> x 2))) [2 7 8 1]→8 [3 19 14]→19 $f(\ell)$=(f2 ℓ)	[7 3]→False [3]→False [9 0 0]→True [0]→True [0 7 3]→True $f(\ell)$=(f3 ℓ 0)	+106 769-438→106.769.438 +83 973-831→83.973.831 f(s)=(f0 "." "-" (f0 "." " " (cdr s))) Temple Anna H →TAH Lara Gregori→LG f(s)=(f2 s)	  f(x)=(f1 x) f(x)=(f6 x)   f(x)=(f4 x) f(x)=(f3 x)
DSL	f_1(ℓ,p) = (foldr ℓ nil (λ (x a) (if (p x) (cons x a) a))) (f_1: *Higher-order filter function*) f_2(ℓ) = (foldr ℓ 0 (λ (x a) (if (> a x) a x))) (f_2: *Maximum element in list ℓ*) f_3(ℓ,k) = (foldr ℓ (is-nil ℓ) (λ (x a) (if a a (= k x)))) (f_3: *Whether ℓ contains* k)	f_0(s,a,b) = (map (λ (x) (if (= x a) b x)) s) (f_0: *Performs character substitution*) f_1(s,c) = (foldr s s (λ (x a) (cdr (if (= c x) s a)))) (f_1: *Drop characters from* s *until* c *reached*) f_2(s) = (unfold s is-nil car (λ (z) (f_1 z " "))) (f_2: *Abbreviates a sequence of words*)	f_0(x) = (+ x real) f_1(x) = (f_0 (* real x)) f_2(x) = (f_1 (* x (f_0 x))) f_3(x) = (f_0 (* x (f_2 x))) f_4(x) = (f_0 (* x (f_3 x))) (*f_4: 4th order polynomial*) f_5(x) = (/ real x) f_6(x) = (f_5 (f_0 x)) (*f_6: rational function*)	

Table 1: Top: Tasks from three domains we apply our algorithm to, each followed by the programs DREAMCODER discovers for them. Bottom: Several examples from learned DSL. Notice that learned DSL primitives can call each other, and that DREAMCODER rediscovers higher-order functions like `filter` (f_1 under List Functions)

DSL by discovering and reusing domain-specific subroutines; and training a neural network that efficiently guides search for programs in the DSL. The learned DSL effectively encodes a prior on programs likely to solve tasks in the domain, while the neural net looks at the example input-output pairs for a specific task and produces a "posterior" for programs likely to solve that specific task. 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 [16]. The recognition model ensures that searching for programs remains tractable even as the DSL (and hence the search space for programs) expands.

Concretely, our algorithm iterates through three phases. The **Wake** phase takes a given set of **tasks**, typically several hundred, and searches for compact programs that solve these tasks, guided by the current DSL and neural network. The **Sleep-G** phase grows the DSL (or **G**enerative model), which allows the agent to more compactly write programs in the domain. We modify the structure of the DSL by discovering regularities across programs, compressing them to distill out common abstractions across programs discovered during waking.[1] The **Sleep-R** phase improves the search procedure by training a neural network (the **R**ecognition model) to write programs in the current DSL, in the spirit of "amortized" or "compiled" inference [25]. We train the recognition model on two data sources: samples from the DSL (as in the Helmholtz Machine's "sleep" phase), and programs found during waking.[2]

Brisk overview of related work. Prior work on program learning has largely assumed a fixed, hand-engineered DSL, both in classic symbolic program learning approaches (e.g., Metagol: [29], FlashFill: [15]), neural approaches (e.g., RobustFill: [8]), and hybrids of neural and symbolic methods (e.g., Neural-guided deductive search: [**?**], DeepCoder: [2]). A notable exception is the EC algorithm [6], which also learns a library of subroutines. We find EC motivating, and go beyond it and other prior work through the following contributions:

Contributions. (1) We show how to learn-to-learn programs in an expressive Lisp-like programming language, including conditionals, variables, and higher-order recursive functions; (2) We give an algorithm for learning DSLs, built on a formalism known as Fragment Grammars [30]; and (3) We give a hierarchical Bayesian framing enabling joint inference of the DSL and recognition model. up

4 The DREAMCODER Algorithm

We first mathematically describe our 3-step algorithm as an inference procedure for a hierarchical Bayesian model (Section 4.1), and then describe each step algorithmically in detail (Section 4.2-4.4).

[1] This is loosely biologically inspired by the formation of abstractions during sleep memory consolidation [9]

[2] These two sources are also loosely biologically inspired by the distinct episodic replay and hallucination components of dream sleep [13]

4.1 Hierarchical Bayesian Framing

DREAMCODER takes as input a set of **tasks**, written X, each of which is a program synthesis problem. It has at its disposal a domain-specific *likelihood model*, written $\mathbb{P}[x|p]$, which scores the likelihood of a task $x \in X$ given a program p.[3] Its goal is to solve each of the tasks by writing a program, and also to infer a DSL, written $\mathcal{D}$. We equip $\mathcal{D}$ with a real-valued weight vector θ, and together $(\mathcal{D}, \theta)$ define a generative model over programs. We frame our goal as 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:

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

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

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 $\mathscr{L}$:

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

We alternate maximization of $\mathscr{L}$ w.r.t. $\{\mathcal{F}_x\}_{x \in X}$ (**Wake**) and $(\mathcal{D}, \theta)$ (**Sleep-G**):

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-G: 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 **Sleep-R** 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-R: 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 (Figure 1). 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 Sleep-G phase has more data from which to learn the DSL; and, because Sleep-R 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.

4.2 Wake: Searching for Programs

During waking, the agent's goal is to search for programs solving the tasks. We use the simple approach of enumerating programs from the DSL in decreasing order of their probability according to the recognition model, and then checking if a program p assigns positive probability to a task ($\mathbb{P}[x|p] > 0$); if so, we incorporate p into the frontier $\mathcal{F}_x$.

To make this concrete we need to define what programs actually are and what form $\mathbb{P}[p|\mathcal{D}, \theta]$ takes. We represent programs as λ-calculus expressions. λ-calculus is a formalism for expressing functional programs that closely resembles Lisp, including variables, function application, and the ability to create new functions. Throughout this paper we

[3] For example, for string editing, 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.

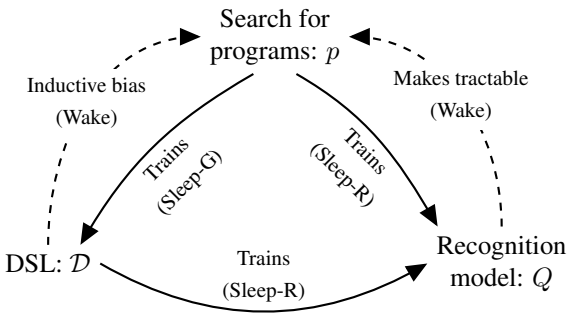


Figure 1: DREAMCODER solves for programs, the DSL, and a recognition model. Each of these steps bootstrap off of the others in a Helmholtz-machine inspired wake/sleep inference algorithm.

will write λ-calculus expressions in Lisp syntax. Our programs are all strongly typed. We use the Hindley-Milner polymorphic typing system [33] which is used in functional programming languages like OCaml and Haskell. We now define DSLs:

Definition 2. *A DSL $\mathcal{D}$ is a set of typed λ-calculus expressions. A weight vector θ for a DSL $\mathcal{D}$ is a vector of $|\mathcal{D}|+1$ real numbers: one number for each DSL element $e \in \mathcal{D}$, written θ_e and controlling the probability of e occurring in a program, and a weight controlling the probability of a variable occurring in a program, θ_{var}.*

Together with its weight vector, a DSL defines a distribution over programs, $\mathbb{P}[p|\mathcal{D},\theta]$. We define this distribution by specifying a procedure for drawing samples from $\mathbb{P}[p|\mathcal{D},\theta]$ (Algorithm 1). Care must be taken to ensure that variable scoping rules are obeyed and that programs are well-typed. We ensure well-typed programs by performing Hindley-Milner type inference [33] during sampling, and assume that each task is annotated with the type of the program that will solve it. Appendix A.1 explains how we enumerate, rather than sample, programs generated by Algorithm 1.

Why enumerate, when the program synthesis community has invented many sophisticated algorithms that search for programs? [39, 36, 12, 31, 34]. 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.

However, a drawback of enumerative search is that we have no efficient means of solving for arbitrary constants that might occur in a program. In Sec. 5.3, we will show how to find programs with real-valued constants by automatically differentiating through the program and setting the constants using gradient descent.

4.3 Sleep-R: Training a Neural Recognition Model

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

4.3.1 Training Q

How should we get the data to train Q? This is non-obvious because we are considering a weakly supervised setting (i.e., learning only from tasks and not from task/program pairs). One approach is to draw programs from the DSL, run them to get their input/outputs, and then train Q to predict the program from the input/outputs. This is like how the wake-sleep algorithm for the Helmholtz machine trains its recognition model during its sleep phase [5]. The advantage of training on samples, or "dreams," is that we can draw unlimited samples from the DSL, training on a large amount of data. Another approach is to train Q on the (program, task) pairs discovered by the Wake phase. The advantage here is that the training data is much higher quality, because we are training on real tasks. Due to these complementary advantages, we train on both these sources of data.

Formally, Q should approximate the true posteriors over programs. We can either train Q to perform full posterior inference by minimizing the expected KL-divergence, $\mathbb{E}\left[\mathrm{KL}\left(\mathbb{P}[p|x,\mathcal{D},\theta]\|Q(p|x)\right)\right]$, or we can train Q to perform

Algorithm 1 Generative model over programs

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

function sample$'(\mathcal{D}, \theta, \mathcal{E}, \tau)$:
Input: DSL $(\mathcal{D}, \theta)$, environment $\mathcal{E}$, type τ ▷ Environment $\mathcal{E}$ starts out as $\varnothing$
Output: a program whose type unifies with τ
if $\tau = \alpha \to \beta$ **then** ▷ Function type — start with a lambda
    var $\leftarrow$ 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 τ
    primitives $\leftarrow \{p | p : \tau' \in \mathcal{D} \cup \mathcal{E}$ if τ can unify with yield$(\tau')\}$ ▷ Everything in scope w/ type τ
    variables $\leftarrow \{p \mid p \in$ 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 τ with yield(τ'). ▷ Ensure well-typed program
    $\{\alpha_k\}_{k=1}^{K} \leftarrow \text{args}(\tau')$
    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:

$$\text{yield}(\tau) = \begin{cases} \text{yield}(\beta) & \text{if } \tau = \alpha \to \beta \\ \tau & \text{otherwise.} \end{cases}$$ ▷ Final return type of τ

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

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 the real data; taking it over samples from the generative model trains Q on "dreams." We define a pair of alternative objectives for the recognition model, $\mathcal{L}^{\text{posterior}}_{\text{RM}}$ and $\mathcal{L}^{\text{MAP}}_{\text{RM}}$, which either train Q to perform full posterior inference or MAP inference, respectively. These objectives combine real-data and dream data:

$$\mathcal{L}^{\text{posterior}}_{\text{RM}} = \mathcal{L}^{\text{posterior}}_{\text{Real}} + \mathcal{L}^{\text{posterior}}_{\text{Dream}} \qquad \mathcal{L}^{\text{MAP}}_{\text{RM}} = \mathcal{L}^{\text{MAP}}_{\text{Real}} + \mathcal{L}^{\text{MAP}}_{\text{Dream}}$$

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

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

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

The $\mathcal{L}^{\text{MAP}}_{\text{Dream}}$ 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 $\mathcal{L}^{\text{MAP}}_{\text{Dream}}$, we *enumerate* programs in decreasing order of their prior probability, tracking, for each dreamed task x, the set of enumerated programs maximizing $\mathbb{P}[x,p|\mathcal{D},\theta]$.

4.3.2 Parameterizing Q

Broadly the literature contains two different approaches to parameterizing conditional distributions over programs. The first approach, seen in systems like RobustFill [8] and [45], is to use a recurrent neural network to predict the program token-by-token as in Seq2Seq or Seq2Tree. The advantage of this approach is that the neural net can predict the entire program, so if the network is sufficiently powerful, it can completely solve the synthesis problem. There are two disadvantages to this approach. First, these models can perform poorly at out-of-sample generalization [], which is critical for our setting, as the agent may need to solve new tasks that are qualitatively different from the tasks it has solved so far. Second, a powerful deep recurrent network may be costly to sample or enumerate from — so if the network cannot easily solve a task, we cannot compensate with rapid sampling or enumeration. In contrast, state-of-the-art enumerative program synthesizers evaluate millions of programs per second [12].

The second approach is to have Q predict a fixed-dimensional weight vector, which then biases a fast enumerator [2] or sampler [28]. This approach can enjoy strong out-of-sample generalization, because it can fall back on enumeration or sampling when the target program is unlike the training programs. A main drawback is that the neural net is deliberately handicapped, and can only send so much information about the target program.

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

Formally, Q predicts a $(|\mathcal{D}|+2)\times(|\mathcal{D}|+1)\times A$-dimensional tensor, where A is the maximum 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. Figure 2 diagrams this generative process, Algorithm 2 specifies a sampling procedure for $Q(\cdot|x)$, and Figure 3 diagrams the neurally guided program inference procedure. This parameterization has three main advantages: (1) it supports fast enumeration and sampling of programs, because the recognition model only needs to run once for each task; (2) it allows the recognition model to provide 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.

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

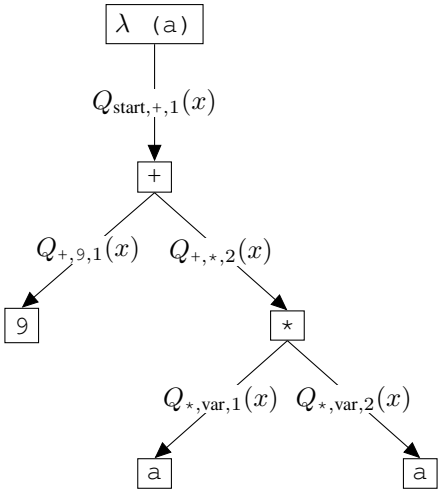


Figure 2: Parameterization of distribution over programs predicted by recognition model. Here the program (syntax tree shown above) is `(λ (a) (+ 9 (* a a )))`. Each conditional distribution predicted by the recognition model is written $Q_{\text{parent,child,argument index}}(x)$, where x is a task.

Algorithm 2 Drawing from distribution over programs predicted by recognition model. Compare w/ Algorithm 1

function recognitionSample$(Q, x, \mathcal{D}, \tau)$:
Input: recognition model Q, task x, DSL $\mathcal{D}$, type τ
Output: a program whose type unifies with τ
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 τ
Output: a program whose type unifies with τ
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 τ
    primitives $\leftarrow \{p | p : \tau' \in \mathcal{D} \cup \mathcal{E}$ if τ can unify with yield$(\tau')\}$ ▷ Everything in scope w/ type τ
    variables $\leftarrow \{p \mid p \in$ primitives and p a variable$\}$
    Draw $e \sim$ primitives, w.p. $\propto \begin{cases} Q_{\text{parent},e,\text{argumentIndex}}(x) & \text{if } e \in \mathcal{D} \\ Q_{\text{parent,var,argumentIndex}}(x)/|\text{variables}| & \text{if } e \in \mathcal{E} \end{cases}$
    Unify τ with yield(τ'). ▷ Ensure well-typed program
    newParent$\leftarrow \begin{cases} e & \text{if } e \in \mathcal{D} \\ \text{var} & \text{if } e \in \mathcal{E} \end{cases}$
    $\{\alpha_k\}_{k=1}^{K} \leftarrow \text{args}(\tau')$
    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

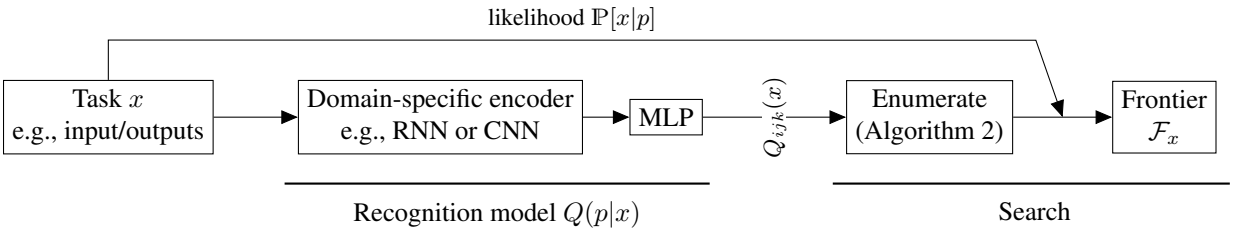


Figure 3: Neurally-guided program inference pipeline. Recognition model outputs distribution over program $Q(p|x)$ parameterized by tensor Q_{ijk}. Program output by enumerative search incorporated into frontier if likelihood $\mathbb{P}[x|p] > 0$

4.3.3 Learning to break symmetries in program space

A good DSL not only exposes high-level building blocks, but also carefully restricts the ways in which those building blocks are allowed to compose. For example, a DSL for list manipulation should contain both the empty list and a routine for appending lists, but should not allow appending the empty list. Similarly a DSL for arithmetic should contain both addition and the number zero but disallow adding zero. These restrictions break symmetries in the space of programs. Here we show that the combination of the $\mathcal{L}^{\text{MAP}}$ training objective (Section 4.3.1) and bigram parameterization (Section 4.3.2) interact in a way that spontaneously breaks symmetries in the program space, allowing the agent to more efficiently explore the space of programs.

Consider an agent tasked with writing programs built from addition and the constants zero and one. A bigram parameterization of Q allows it to represent the fact that it should never add zero ($Q_{+,0,0} = Q_{+,0,1} = 0$) or that addition should always associate to the right ($Q_{+,+,0} = 0$). The $\mathcal{L}^{\text{MAP}}$ 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

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

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 1), we have

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

So $\mathcal{L}^{\text{MAP}}$ prefers Q_2 (the symmetry breaking recognition model), while $\mathcal{L}^{\text{posterior}}$ reverses this preference. 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[5] parameterization. Figure 4 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.

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 rest of the paper.

[5]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 [10]

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

4.4 Sleep-G: Learning a Generative Model (a DSL)

The purpose of the DSL is to offer a set of abstractions that allow an agent to easily express solutions to the tasks at hand. Intuitively, we want the algorithm to look at the programs found during waking and generalize beyond them, both so the DSL can better express the current solutions, and also so that the DSL might expose new abstractions which will later be used to discover more programs. Formally, we want the DSL maximizing $\int \mathscr{L}\ \mathrm{d}\theta$ (Sec. 4.1). We replace this marginal with an AIC approximation, giving the following objective for DSL induction:

$$\log \mathbb{P}[\mathcal{D}] + \arg\max_{\theta} \left(\log \mathbb{P}[\theta|\mathcal{D}] - \|\theta\|_0 + \sum_{x \in X} \log \sum_{p \in \mathcal{F}_x} \mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D},\theta] \right) \tag{3}$$

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 increase. 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 5 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.

Formally, we want a DSL which puts high probability on any refactoring of the programs found during waking, maximizing:

$$\log \mathbb{P}[\mathcal{D}] + \arg\max_{\theta} \left(\log \mathbb{P}[\theta|\mathcal{D}] - \|\theta\|_0 + \sum_{x \in X} \log \sum_{p \in \mathcal{F}_x} \mathbb{P}[x|p] \max_{p' \longrightarrow^* p} \mathbb{P}[p'|\mathcal{D},\theta] \right) \tag{4}$$

where $p' \longrightarrow^* p$ is the standard notation for "expression p' evaluates to p by the rules of λ-calculus" [33].

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 inference problem is intractable, because there are infinitely many ways of refactoring a program. We combine two ideas to make this refactoring process tractable:

Idea 1: Limit the degree to which a piece of code can be refactored. Instead of considering every refactoring, bound the number of λ-calculus evaluation steps separating a refactoring from its original program. Formally, we define the

set of n-step refactorings as:

$$R_n(p) = \left\{ p' \ : \ p' \underbrace{\longrightarrow p'' \longrightarrow \cdots \longrightarrow}_{\leq n \text{ times}} p \right\} \tag{5}$$

where $p_1 \longrightarrow p_2$ is the standard notation for "p_1 rewrites to p_2 in one step according to the rules of λ-calculus" [33]. For example,

```
((lambda (x) (x x)) (lambda (y) y)) ⟶
    ((lambda (y) y) (lambda (y) y)) ⟶
                        (lambda (y) y)
```

Returning to Equation 4, this approximation gives the following objective:

$$\log \mathbb{P}[\mathcal{D}] + \arg\max_{\theta} \left(\log \mathbb{P}[\theta|\mathcal{D}] - \|\theta\|_0 + \sum_{x \in X} \log \sum_{p \in \mathcal{F}_x} \mathbb{P}[x|p] \max_{p' \in R_n(p)} \mathbb{P}[p'|\mathcal{D}, \theta] \right) \tag{6}$$

In practice, setting the number of refactoring steps n to 3 suffices to give a competent DSL learning algorithm. Although the number of refactorings is now finite, it is still prohibitively large, and in fact grows exponentially quickly both as a function of n and as a function of the size of the program being refactored. For example, for the programs in Figure 5, there are approximately 10^{14} possible refactorings. Next, we show how to tame this exponential explosion.

Idea 2: Rather than explicitly enumerate every refactoring, we developed a technique for compactly representing and efficiently calculating the set of all refactorings. The key idea is to represent the set of refactorings using a *version space* [24, **?**, 34]. A version space is a tree-shaped data structure that compactly represents a large set of programs. We first define version spaces over λ-calculus expressions, and then give a dynamic program for constructing a version space that represents the set of possible refactorings (i.e., $R_n(p)$). This technique is astronomically more efficient than explicitly representing the space of possible refactorings: for the example in Figure 5, we represent the space of refactorings using a version space with 10^6 nodes, which encodes 10^{14} refactorings. Formally, a version space is either:

- A deBuijn[6] 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 3. *The **extension** of a version space v is written $[\![v]\!]$ and is defined recursively as:*

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

[6]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 [33] for more details.

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

$$\begin{aligned}
\text{REFACTOR}(v|\mathcal{D}) &= \begin{cases} e, \text{ if } e \in \mathcal{D} \text{ and } e \in v. \text{ Exploits the fact that } e \in v \text{ can be efficiently computed.} \\ \text{REFACTOR}'(v|\mathcal{D}), \text{ otherwise.} \end{cases} \\
\text{REFACTOR}'(e|\mathcal{D}) &= e, \text{ if } e \text{ is a leaf} \\
\text{REFACTOR}'(\lambda b|\mathcal{D}) &= \lambda \text{REFACTOR}(b|\mathcal{D}) \\
\text{REFACTOR}'(f\ x|\mathcal{D}) &= \text{REFACTOR}(f|\mathcal{D})\ \text{REFACTOR}(x|\mathcal{D}) \\
\text{REFACTOR}'(\uplus V|\mathcal{D}) &= \underset{e \in \{\text{REFACTOR}(v|\mathcal{D})\ :\ v \in V\}}{\arg\min} \text{size}(e|\mathcal{D})
\end{aligned} \tag{7}$$

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

In Appendix A.2, we describe a dynamic program for efficiently constructing a version space containing every n-step refactoring. This dynamic program, which we call $I\beta_n(p)$, satisfies $[\![I\beta_n(p)]\!] = R_n(p)$. In other words, this dynamic program builds a data structure that represents the entire set of refactorings – but without having to explicitly enumerate all of the refactorings.

With this machinery in hand, we now have all the pieces needed to learn a DSL (Algorithm 3). To define the prior distribution over $(\mathcal{D}, \theta)$ (Algorithm 3, lines 7-8), we penalize the syntactic complexity of the λ-calculus expressions in the DSL, defining $\mathbb{P}[\mathcal{D}] \propto \exp(-\lambda \sum_{p \in \mathcal{D}} \text{size}(p))$ where $\text{size}(p)$ measures the size of the syntax tree of program p, and λ controls how strongly we regularize the size of the DSL. We place a symmetric Dirichlet prior over the weight vector θ.

To appropriately score each proposed $\mathcal{D}$ we must reestimate the weight vector θ (Algorithm 3, 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 [22], our DSLs are context-sensitive due to the presence of variables in the programs and also due to the polymorphic typing system. Appendix A.3 derives a tractable MAP estimator for θ.

Algorithm 3 DSL Induction Algorithm

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

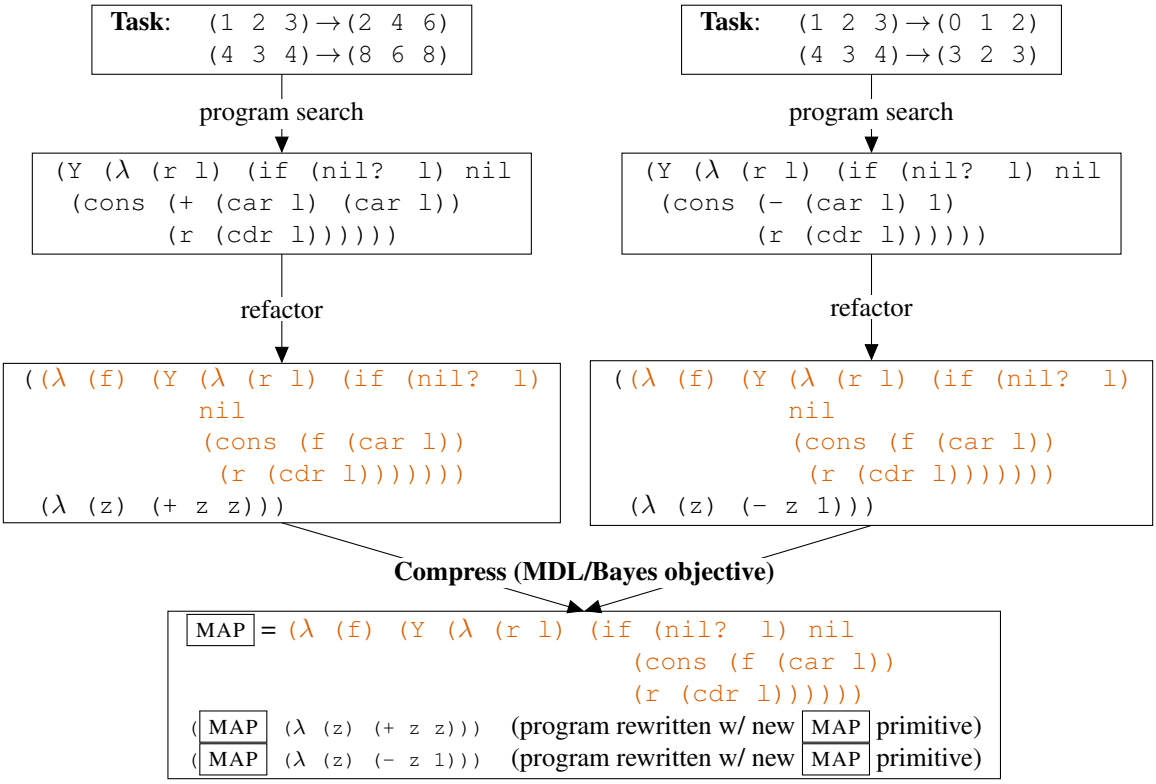


Figure 5: DSL learning as code refactoring. For each task we discover programs during waking, then refactor the code from those programs to expose common subprograms (highlighted in orange). Common subprograms are incorporated into the DSL when they increase a Bayesian objective. Intuitively, these new DSL components best compress the programs found during waking.

5 Experiments

5.1 Programs that manipulate sequences

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

List Processing: Synthesizing programs that manipulate data structures is a widely studied problem in the programming languages community [12]. We consider this problem within the context of learning functions that manipulate lists, and also perform arithmetic operations upon lists (Table 2). We created 236 human-interpretable list manipulation tasks, each with 15 input/output examples. In solving these tasks, the system composed 38 new subroutines, and rediscovered the higher-order function `filter` (f_1 in Table 3, left).

Text Editing: Synthesizing programs that edit text is a classic problem in the programming languages and AI literatures [15, 24]. This prior work uses hand-engineered DSLs. Here, we instead start out with generic sequence manipulation primitives and recover many of the higher-level building blocks that have made these other systems successful.

We trained our system on 109 automatically-generated text editing tasks, with 4 input/output examples each. After three iterations, it assembles a DSL containing a dozen new functions (Fig. 3, center) solving all the training tasks. But, how well does the learned DSL generalized to real text-editing scenarios? We tested, but did not train, on the 108 text editing problems from the SyGuS [1] program synthesis competition. Before any learning, DREAMCODER solves 3.7% of the problems with an average search time of 235 seconds. After learning, it solves 74.1%, and does so much faster, solving them in an average of 29 seconds. As of the 2017 SyGuS competition, the best-performing algorithm solves 79.6% of the problems. But, SyGuS comes with a different hand-engineered DSL *for each text editing problem*. Here we learned a single DSL that applied generically to all of the tasks, and perform comparably to the best prior work.

5.2 Programs that make plans and take actions

We apply DREAMCODER to two domains where the agent plans a series of actions in order to draw a picture (Sec. 5.2.1) or build a tower out of blocks (Sec. 5.2.2). For drawing pictures, the agent controls a 'pen' that it can pick up or place down while moving across a canvas. For building towers, the agent controls a 'hand' that it moves through a simulated world while placing down differently sized blocks. For each of these domains, the agent observes a target picture or block tower, and must either draw the picture or build the tower. The recognition model is a CNN that observes an image of the target picture or tower. The system is initially provided with two control flow operators: `loop` (a 'for' loop) and `get/set`, which saves and then restores the current state of the pen or hand.

5.2.1 Programs that draw pictures

We take inspiration from LOGO Turtle graphics [43], and task our agent with drawing 90 different pictures (Figure 6). The agent has primitives for moving the pen forward and rotating it (`move`), picking the pen up and then putting it down (`pen-up`), arithmetic operations on angles and distances, and constants (2π, 0 through 9, and ∞[7]).

Over the course of 5 wake/sleep iterations, the system authors a DSL containing parametric and, sometimes, higher-order drawing routines, allowing it to write programs for objects like polygons, circles, spirals, and snowflakes. Figure 7 illustrates several learned DSL subroutines. Once it has synthesized these new subroutines, the agent generates a rich and highly varied data set of dreams on which to train the recognition model. Figure 8 illustrates dreams before

[7] To ensure that programs terminate, we set $\infty = 20$

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

Table 2: Some tasks in our list function domain.

and after learning — one sees that the system has learned to recombine the building blocks and motifs in the training data.

5.2.2 Building towers out of 'Lego' blocks

Inspired by the classic AI 'copy task' — where an agent must look at an image of a tower made of toy blocks and re-create the tower [**?**] — we give DREAMCODER 112 tower 'copy tasks' (Figure 9). 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 (e.g., we do not model gravity and discretize the coordinates of the blocks).

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

5.3 Symbolic Regression: Programs from visual input

We apply DREAMCODER to symbolic regression problems. Here, the agent observes points along the curve of a function, and must write a program that fits those points. We initially equip our learner with addition, multiplication, and division, and task it with solving 100 problems, each either a polynomial or rational function. The recognition model is a convnet that observes an image of the target function's graph (Fig. 11) — visually, different kinds of polynomials and rational functions produce different kinds of graphs, and so the convnet can look at a graph and predict what kind of function best explains it. A key difficulty, however, is that these problems are best solved with programs containing real numbers. Our solution to this difficulty is to enumerate programs with real-valued parameters, and then fit those parameters by automatically differentiating through the programs the system writes and use gradient descent to fit the parameters. We define the likelihood model, $\mathbb{P}[x|p]$, 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 polynomials of different orders or ratios of polynomials. The model also learns to find programs minimizing the number of continuous parameters — learning to represent linear functions with `(* real (+ x real))`, which has two continuous parameters, and represents quartic functions using f_4 in the rightmost column of Fig. 3 which has five continuous parameters. This phenomenon arises from our Bayesian framing: both the generative model's bias toward shorter programs, and the likelihood model's BIC penalty.

5.4 Learning from Scratch

A long-standing dream within the program induction community is "learning from scratch": starting with a *minimal* Turing-complete programming language, and then learning to solve a wide swath of induction problems [40, 38, 17, 41]. 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 [27]: 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. 5.1) that we initially provided our system with functional programming routines like `map` and `fold`.

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

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

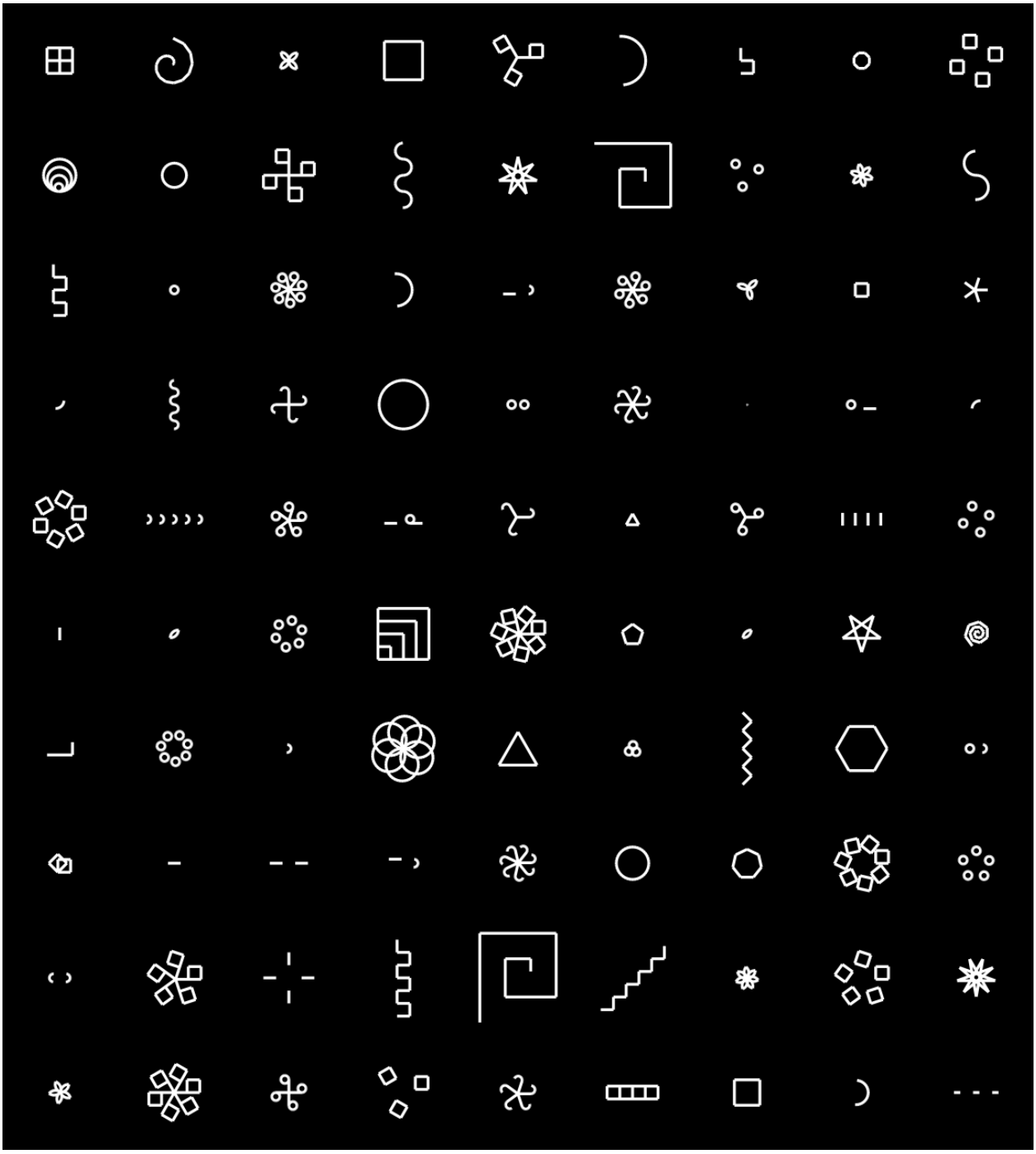


Figure 6: LOGO graphics tasks. The agent must learn to write programs that draw these 90 pictures.

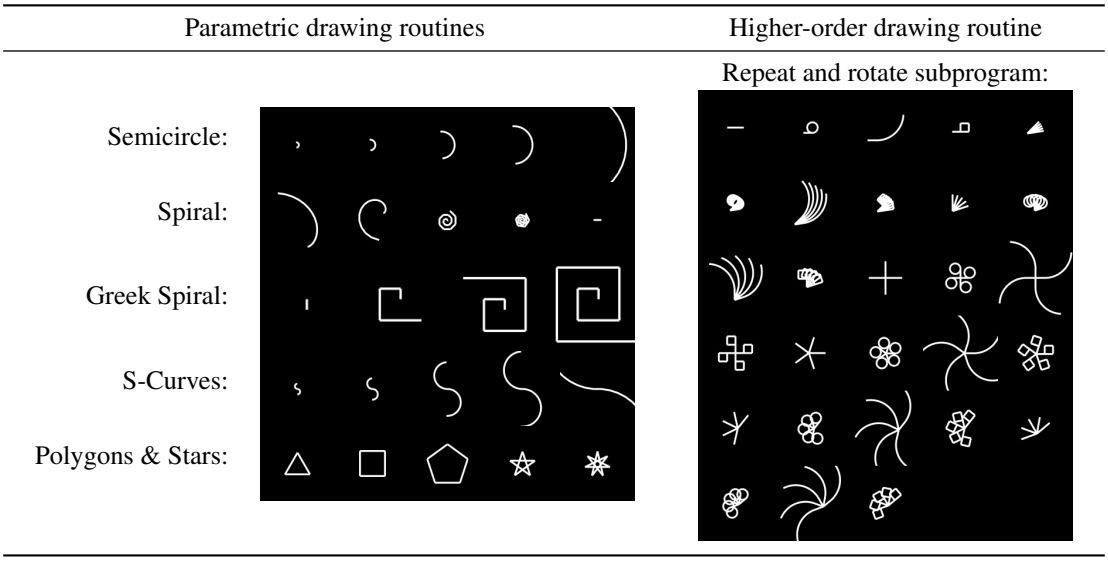


Figure 7: Example primitives learned by DREAMCODER when trained on tasks in Figure 6. Agent learns parametric routines for drawing families of curves (left) as well as subroutines that take entire programs as input (right). Each row of images on the left is the same code executed with different parameters. Each image on the right is the same code executed with different parameters and with a different subprogram provided as input.

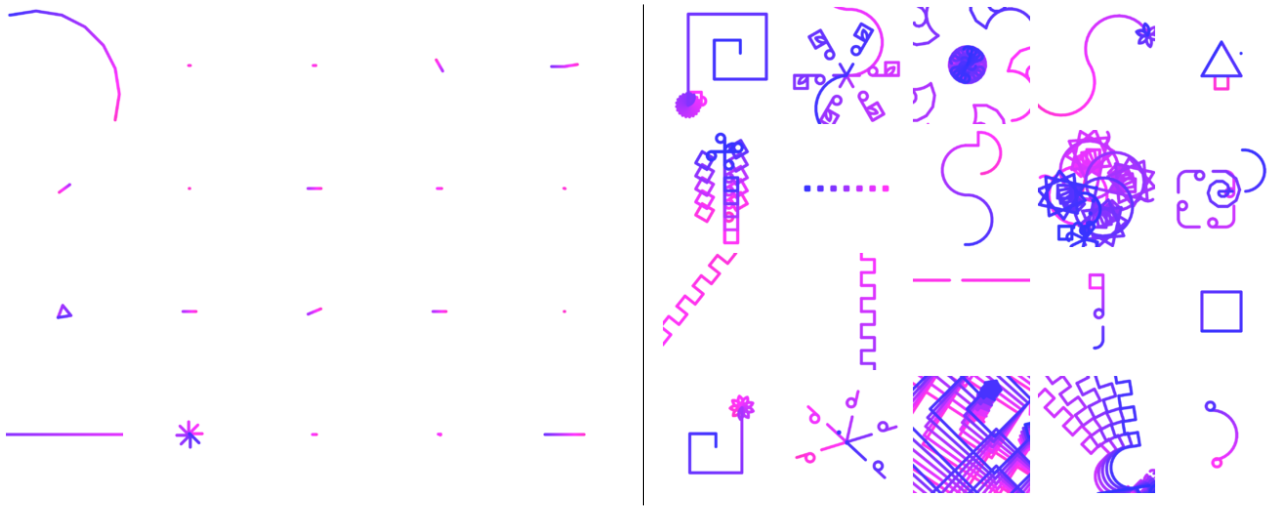


Figure 8: Dreams, or samples, from the DSL before (left) and after (right) training on tasks in Figure 6. Blue: where the agent started drawing. Pink: where the agent ended drawing.

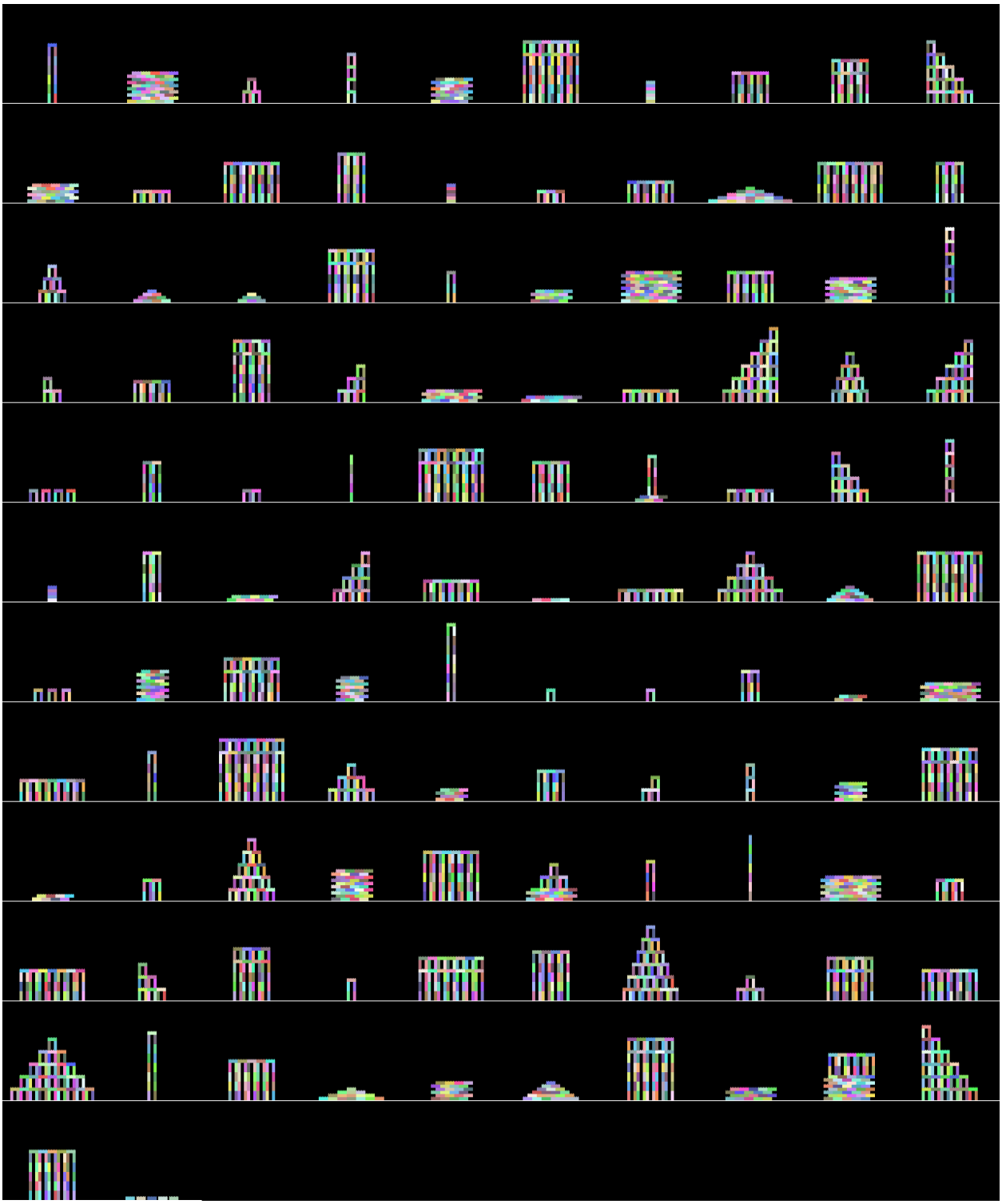


Figure 9

Figure 10: Learned subroutines for building towers out of Lego-style blocks when trained on tasks in Figure 9. These subroutines act like parametric options [**?**], giving higher-level building blocks that the agent can use to plan.

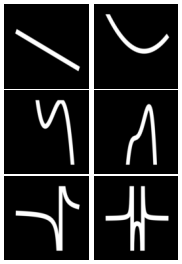


Figure 11: Recognition model input for symbolic regression. DSL learns subroutines for polynomials (top rows) and rational functions (bottom) while the recognition model jointly learns to look at a graph of the function (left) and predict which learned subroutines best explain the observation.

Programs & Tasks	DSL

`[2 1 4]→[2 1 4 0]`
`[9 8]→[9 8 0]`
$f(\ell)$ = `(`f_2 `cons` ℓ `(cons 0 nil))`
	f_0`(p,f,n,x) = (if (p x) nil (cons (f x) (`f_0 `(n x))))`
	(f_0: *unfold*)

`[2 5 6 0 6]→19`
`[9 2 7 6 3]→27`
$f(\ell)$ = `(`f_2 `+` ℓ `0)`
	f_1`(i,l) = (if (= i 0) (car l) (`f_1 `(- i 1) (cdr l))))`
	(f_1: *index*)

`[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`)`
	f_2`(f,l,x) = (if (empty? l) x (f (car l) (`f_2 `(cdr l))))`
	(f_2: *fold*)

`[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`)`
	f_3`(f,l) = (`f_2 `nil l (λ (x a) (cons (f x) a)))`
	(f_3: *map*)
	f_4`(`ℓ`) = (if (empty?` ℓ`) 0 (+ 1 (`f_4 `(cdr` ℓ`)))))`
	(f_4: *length*)
	f_5`(`n`) = (`f_0 `(=` n`) (λ (x) x) (+ 1) 0)`
	(f_5: *range*)

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

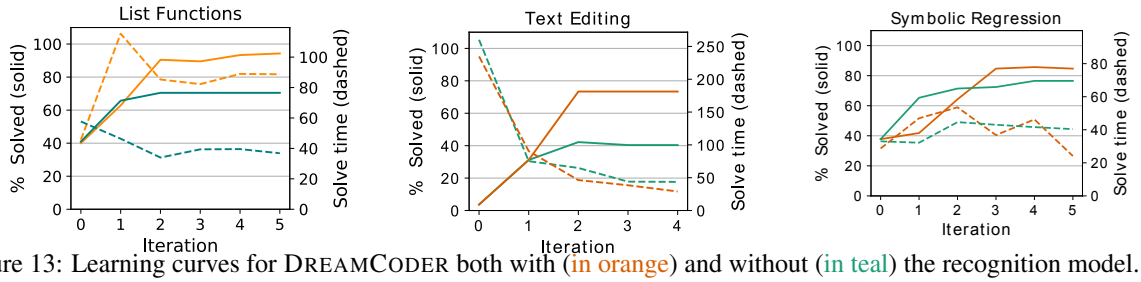


Figure 13: Learning curves for DREAMCODER both with (in orange) and without (in teal) the recognition model. Solid lines: % holdout testing tasks solved w/ 10m timeout. Dashed lines: Average solve time, averaged only over tasks that are solved.

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

5.5 Learning Generative Models

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

5.6 Quantitative Results on Held-Out Tasks

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

6 Discussion

We contribute an algorithm, DREAMCODER, that learns to program by bootstrapping a DSL with new domain-specific primitives that the algorithm itself discovers, together with a neural recognition model that learns to efficiently deploy the DSL on new tasks. Two immediate future goals are to integrate more sophisticated neural recognition models [8] and program synthesizers [39], which may improve performance in some domains over the generic methods used here. Another direction is DSL meta-learning: can we find a *single* universal primitive set that could bootstrap DSLs for new domains, including the domains considered here, but also many others?

References

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

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

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

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

[5] Peter Dayan, Geoffrey E Hinton, Radford M Neal, and Richard S Zemel. The helmholtz machine. *Neural computation*, 7(5):889–904, 1995.

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

[7] Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural program meta-induction. In *NIPS*, 2017.

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

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

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

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

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

[13] 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.

[14] Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S. M. Ali Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. *CoRR*, abs/1804.01118, 2018.

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

[16] 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.

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

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

[19] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *CVPR*.

[20] 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.

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

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

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

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

[25] Tuan Anh Le, Atlm Gne Baydin, and Frank Wood. Inference Compilation and Universal Probabilistic Programming. In *AISTATS*, 2017.

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

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

[28] 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.

[29] 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.

[30] Timothy J. O'Donnell. *Productivity and Reuse in Language: A Theory of Linguistic Computation and Storage*. The MIT Press, 2015.

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

[32] Steven Thomas Piantadosi. *Learning and the language of thought*. PhD thesis, Massachusetts Institute of Technology, 2011.

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

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

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

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

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

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

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

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

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

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

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

[44] 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.

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

A Appendix

A.1 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 1 and 2 specify procedures for sampling from these distributions, but not for enumerating from them. We combine two different enumeration strategies, which allowed us to build a massively parallel program enumerator:

- **Best-first search:** Best-first search maintains a heap of partial programs ordered by their probability — here a partial program means a program whose syntax tree may contain unspecified 'holes'. Best-first search is guaranteed to enumerate programs in decreasing order of their probability, and has memory requirements that in general grow exponentially as a function of the description length of programs in the heap (thus linearly as a function of run time).
- **Depth-first search:** Depth first search recursively explores the space of execution traces through Algorithm 1 and 2, equivalently maintaining a stack of partial programs. In general it does not enumerate programs in decreasing order of probability, but has memory requirements that grow linearly as a function of the description length of the programs in the stack (thus logarithmically as a function of run time).

Our parallel enumeration algorithm (Algorithm 4) first performs a best-first search until the best-first heap is much larger than the number of CPUs. At this point, it switches to performing many depth-first searches in parallel, initializing a depth first search with one of the entries in the best-first heap. Because depth-first search does not produce programs in decreasing order of their probability, we wrap this entire procedure up into an outer loop that first enumerates programs whose description length is between 0 to Δ, then programs with description length between Δ and 2Δ, then 2Δ to 3Δ, etc., until a timeout is reached. This is similar in spirit to iterative deepening depth first search [35].

Algorithm 4 Parallel enumerative program search algorithm

function enumerate(μ, T, CPUs):
Input: Distribution over programs μ, timeout T, CPU count
Output: stream of programs in approximately descending order of probability under μ
Hyperparameter: nat increase rate Δ ▷ We set $\Delta = 1.5$
lowerBound$\leftarrow 0$
while total elapsed time $< T$ **do**
    heap$\leftarrow$newMaxHeap() ▷ Heap for best-first search
    heap.insert(priority $= 0$, value $=$ empty syntax tree) ▷ Initialize heap with start state of search space
    while $0 <$ |heap| $\leq 10 \times$ CPUs **do** ▷ Each CPU will get approximately 10 jobs (a partial program)
        priority, partialProgram $\leftarrow$ 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$\in$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 $\leftarrow$ lowerBound $+ \Delta$ ▷ Push up lower bound on MDL by Δ
end while

function depthFirst(μ, T, lowerBound, partialProgram): ▷ Each worker does a depth first search. Enumerates completions of partialProgram whose MDL is between lowerBound and lowerBound $+ \Delta$
stack$\leftarrow$[partialProgram]
while total elapsed time $< T$ and stack is not empty **do**
    partialProgram$\leftarrow$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

A.2 Refactoring code with version spaces

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 recursively as follows:

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

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

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

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

where $\uparrow^k$ is the shifting operator [33], which adds k to all of the free variables in a λ-expression or version space. We have written the definition of $I\beta_n$ recursively, but implement it 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. Refactoring is similarly done more quickly with dynamic programming (see Equation 7 for the recursive definition of REFACTOR).

We now formally proved 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.

□

A.2.1 Computational complexity of DSL learning

How long does each update to the DSL in Algorithm 3 take? Constructing the version spaces takes time linear in the number of programs (written P) in the frontiers (Algorithm 3, 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 3 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 3, and $\mathcal{D}$ is the current DSL:

$$\text{beam}_v(\mathcal{D}') = \begin{cases} \text{if } \mathcal{D}' \in \text{dom}(b_v): & b_v(\mathcal{D}') \\ \text{if } \mathcal{D}' \notin \text{dom}(b_v): & \text{REFACTOR}(v, \mathcal{D}) \end{cases}$$

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

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

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.3 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

$$\theta = \arg\max_\theta \log P(\theta) + \sum_x \mathbb{E}_{q_x} \left[\log \mathbb{P}\left[p|\theta\right]\right] \tag{8}$$

$$q_x(p) \propto \mathbb{P}[x|p]\mathbb{P}[p|\theta]\mathbb{1}\left[p \in \mathcal{F}_x\right] \tag{9}$$

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 1 while sampling program p. Jensen's inequality gives a lower bound on the likelihood:

$$\sum_x \mathbb{E}_{q_x}\left[\log \mathbb{P}[p|\theta]\right] =$$

$$\sum_{e\in\mathcal{D}} \log\theta_e \sum_x \mathbb{E}_{q_x}\left[c(e,p_x)\right] - \sum_\tau \mathbb{E}_{q_x}\left[\sum_x c(\tau,p_x)\right] \log \sum_{\substack{e:\tau'\in\mathcal{D}\\ \text{unify}(\tau,\tau')}} \theta_e$$

$$= \sum_e C(e)\log\theta_e - \beta \sum_\tau \frac{\mathbb{E}_{q_x}\left[\sum_x c(\tau,p_x)\right]}{\beta} \log \sum_{\substack{e:\tau'\in\mathcal{D}\\ \text{unify}(\tau,\tau')}} \theta_e$$

$$\geq \sum_e C(e)\log\theta_e - \beta \log \sum_\tau \frac{\mathbb{E}_{q_x}\left[\sum_x c(\tau,p_x)\right]}{\beta} \sum_{\substack{e:\tau'\in\mathcal{D}\\ \text{unify}(\tau,\tau')}} \theta_e$$

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

where we have defined

$$C(e) \triangleq \sum_x \mathbb{E}_{q_x}\left[c(e,p_x)\right]$$

$$R(\tau) \triangleq \mathbb{E}_{q_x}\left[\sum_x c(\tau,p_x)\right]$$

$$\beta \triangleq \sum_\tau \mathbb{E}_{q_x}\left[\sum_x c(\tau,p_x)\right]$$

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:

$$\sum_e (C(e)+\alpha)\log\theta_e - \beta\log\sum_\tau \frac{R(\tau)}{\beta} \sum_{\substack{e:\tau'\in\mathcal{D}\\ \text{unify}(\tau,\tau')}} \theta_e \tag{10}$$

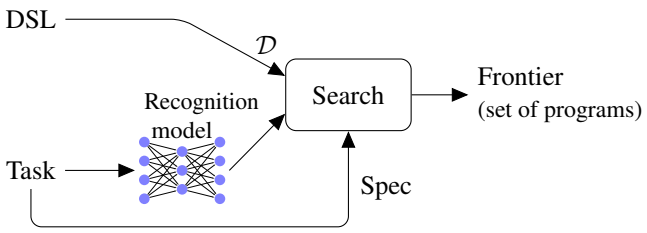
Differentiate with respect to θ_e, where $e:\tau$, and set to zero to obtain:

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

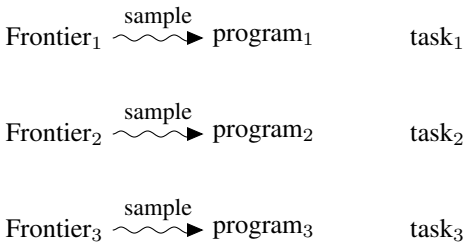
$$\theta_e \propto \frac{C(e)+\alpha}{\sum_{\tau'} \mathbb{1}\left[\text{unify}(\tau,\tau')\right] R(\tau')} \tag{12}$$

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.

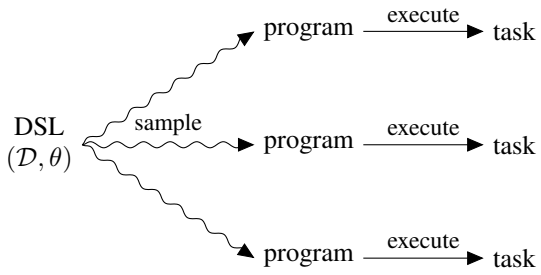
We are only maximizing a lower bound on the log posterior; when is this lower bound tight? This lower bound is tight whenever all of the types of the expressions in the DSL are not polymorphic, in which case our DSL is equivalent to a PCFG and this estimator is equivalent to the inside/outside algorithm. 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.



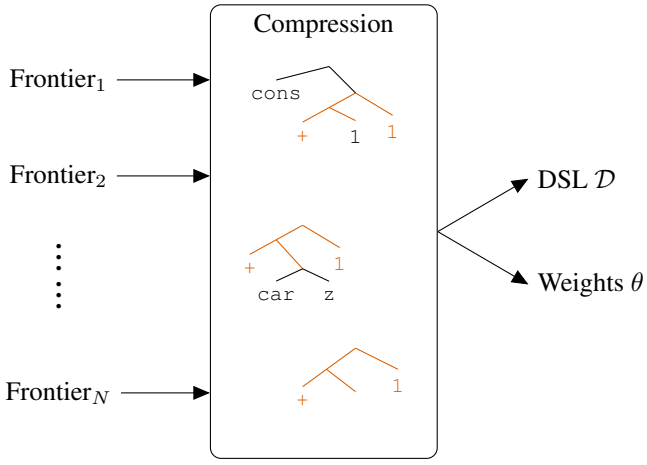
WAKE: PROBLEM SOLVING



SLEEP-R: EXPERIENCE REPLAY



SLEEP-R: DREAMING



SLEEP-G: MEMORY CONSOLIDATION

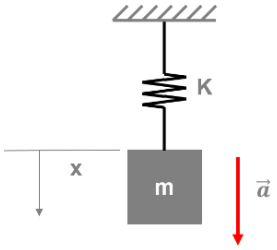
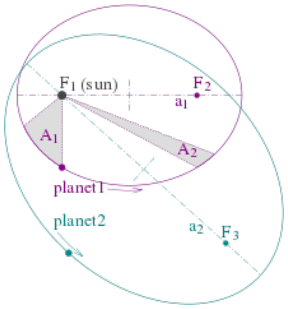
	Symbolic regression		Laws of motion	
Programs & Tasks	 $f(\mathtt{x}) = (f_1\ \mathtt{x})$	 $f(\mathtt{x}) = (f_6\ \mathtt{x})$	 $f(\mathtt{o}, \Delta) = (f_3\ \mathtt{o}\ \Delta\ (-\ (*\ k\ (\mathtt{pos\ o}))\ (*\ -9.8\ \hat{x})))$	 $f(\mathtt{a}, \mathtt{b}, \Delta) = (f_3\ \mathtt{a}\ \Delta\ (/\ (*\ G\ (\mathtt{mass\ a})\ (\mathtt{mass\ b}))\ (\mathtt{square}\ (-\ (\mathtt{pos\ a})\ (\mathtt{pos\ b})))))$
DSL	$f_0(\mathtt{x}) = (+\ \mathtt{x\ real})$ $f_1(\mathtt{x}) = (f_0\ (*\ \mathtt{real\ x}))$ $f_2(\mathtt{x}) = (f_1\ (*\ \mathtt{x}\ (f_0\ \mathtt{x})))$ $f_3(\mathtt{x}) = (f_0\ (*\ \mathtt{x}\ (f_2\ \mathtt{x})))$ $f_4(\mathtt{x}) = (f_0\ (*\ \mathtt{x}\ (f_3\ \mathtt{x})))$ *(f_4: 4th order polynomial)* $f_5(\mathtt{x}) = (/\ \mathtt{real\ x})$ $f_6(\mathtt{x}) = (f_5\ (f_0\ \mathtt{x}))$ *(f_6: rational function)*		$f_0(\mathtt{o}, \Delta) = (\mathtt{set\text{-}pos\ o}\ (+\ (\mathtt{pos\ o})\ (*\ \Delta\ (\mathtt{vel\ o}))))$ *(f_0: integrates position)* $f_1(\mathtt{o}, \mathtt{a}, \Delta) = (\mathtt{set\text{-}vel\ o}\ (+\ (\mathtt{vel\ o})\ (*\ \Delta\ \mathtt{a})))$ *(f_1: integrates velocity)* $f_2(\mathtt{o}, \mathtt{F}) = (/\ \mathtt{F}\ (\mathtt{mass\ o}))$ *(f_2: Newton's second law)* $f_3(\mathtt{o}, \Delta, \mathtt{F}) = (f_0\ \mathtt{o}\ \Delta\ (f_1\ \mathtt{o}\ (f_2\ \mathtt{o}\ \mathtt{F})\ \Delta))$ *(f_3: applies Newton's second law and integrates)*	

Table 3: Top: Tasks from three domains we apply our algorithm to, each followed by the programs DREAMCODER discovers for them. Bottom: Several examples from learned DSL. Notice that learned DSL primitives can call each other, and that DREAMCODER rediscovers higher-order functions like `filter` (f_1 under List Functions)