(

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 [Lake et al.(2015)Lake, Salakhutdinov, and Tenenbaum], intuitive theories (of kinship, taxonomy, etc.) [Ullman et al.(2012)Ullman, Goodman, and Tenenbaum], and natural language grammar and semantics [Schmid & Kitzelmann(2011)Schmid and Kitzelmann, **?**]. 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 [Solomonoff(1964)] 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 [Devlin et al.(2017b)Devlin, Uesato, Bhupatiraju, Singh, Mohamed, and Kohli, Balog et al.(2016)Balog, Gaunt, Brockschmidt, Nowozin, and Tarlow, Kalyan et al.(2018)Kalyan, Mohta, Polozov, Batra, Jain, and Gul **?**], or developed symbolic approaches for representing and searching through spaces of programs [Gulwani(2011), Solar Lezama(2008), Koza()]. We were motivated by approaches that learn or grow the DSL [Dechter et al.(2013)Dechter, Malmaud, Adar Liang et al.(2010)Liang, Jordan, and Klein, Solomonoff(1989), Hwang et al.(2011)Hwang, Stuhlmüller, and Goodman, Stolle & Precup(2002)Stolle and Precup]. 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 [Solomonoff(1964)] 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 [Lake et al.(2015)Lake, Salakhutdinov, and Tenenbaum]; 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.) [Ullman et al.(2012)Ullman, Goodman, and Tenenbaum and natural language grammar [Schmid & Kitzelmann(2011)Schmid and Kitzelmann], 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 [**?**, Ellis et al.(2017)Ellis, Ritchie, Solar-Lezama, and Te transforming text [Gulwani(2011)] and numerical sequences[1] [Balog et al.(2016)Balog, Gaunt, Brockschmidt, Nowozin, and Tarlow], and reasoning over common sense knowledge bases [Muggleton et al.(2015)Muggleton, Lin, and Tamaddoni-Nezhad]. 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 ex-

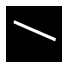
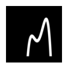
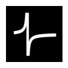
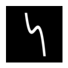
	List Functions		Text Editing	Symbolic Regression	
Programs & Tasks	[7 2 3]→[7 3] [1 2 3 4]→[3 4] [4 3 2 1]→[4 3] $f(\ell)$ = (f_1 ℓ (λ (x) (> x 2))) [2 7 8 1]→8 [3 19 14]→19 $f(\ell)$ = (f_2 ℓ)	[7 3]→False [3]→False [9 0 0]→True [0]→True [0 7 3]→True $f(\ell)$ = (f_3 ℓ 0)	+106 769-438→106.769.438 +83 973-831→83.973.831 f(s) = (f_0 "." "-" (f_0 "." " " (cdr s))) Temple Anna H →TAH Lara Gregori→LG f(s) = (f_2 s)	 f(x) = (f_1 x)  f(x) = (f_4 x)	 f(x) = (f_6 x)  f(x) = (f_3 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)

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 [Gulwani(2011)] 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 [Lake et al.(2015)Lake, Salakhutdinov, and Tenenbaum], procedural graphics [Ellis et al.(2017)Ellis, Ritchie, Solar-Lezama, and Tenenbaum, Ganin et al.(2018)Ganin, Kulkarni, Babuschkin, Esl question answering [Johnson et al.()Johnson, Hariharan, van der Maaten, Fei-Fei, Zitnick, and Girshick] and robot motion planning [Devlin et al.(2017a)Devlin, Bunel, Singh, Hausknecht, and Kohli]. 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 1. 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 [Solar Lezama(2008)] or genetic programming [Koza()], 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. [Hinton et al.(1995)Hinton, Dayan, Frey, and Neal]), iterating between "wake" and "sleep" phases to achieve three goals: finding programs that solve tasks; creating a 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 [Hinton et al.(1995)Hinton, Dayan, Frey, and Neal]. 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 [Le et al.(2017)Le, Baydin, and Wood]. 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: [Muggleton et al.(2015)Muggleton, Lin, and Tamaddoni-Nezh
FlashFill: [Gulwani(2011)]), neural approaches (e.g., RobustFill: [Devlin et al.(2017b)Devlin, Uesato, Bhupatiraju, Singh, Mohamed, and
and hybrids of neural and symbolic methods (e.g., Neural-guided deductive search: [**?**], DeepCoder: [Balog et al.(2016)Balog, Gaunt, Broc
A notable exception is the EC algorithm [Dechter et al.(2013)Dechter, Malmaud, Adams, and Tenenbaum], 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 [O'Donnell(2015)]; 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).

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

[1]This is loosely biologically inspired by the formation of abstractions during sleep memory consolidation [Dudai et al.(2015)Dudai, Karni, and Born]

[2]These two sources are also loosely biologically inspired by the distinct episodic replay and hallucination components of dream sleep [Fosse et al.(2003)Fosse, Fosse, Hobson, and Stickgold]

[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.

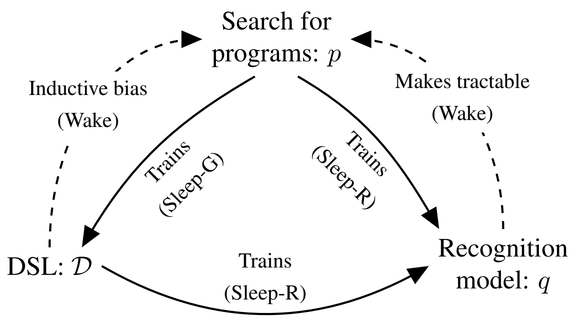


Figure 1: DREAMCODER solves for programs, the DSL, and a recognition model. Each of these steps bootstrap off of the others in a Helmholtz-machine inspired wake/sleep inference algorithm.

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. 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 $\arg\max_\theta \mathscr{L}(\mathcal{D},\theta,\{\mathcal{F}_x\})$.

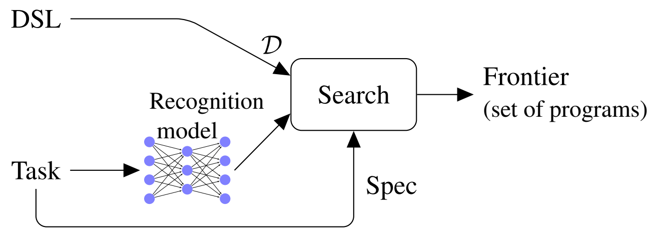
Searching for programs is hard because of the large combinatorial search space. We ease this difficulty by training a neural recognition model, $q(\cdot|\cdot)$, during the **Sleep-R** phase: q is trained to approximate the posterior over programs, $q(p|x) \approx \mathbb{P}[p|x,\mathcal{D}] \propto \mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D}]$. 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 $\mathscr{L}$.

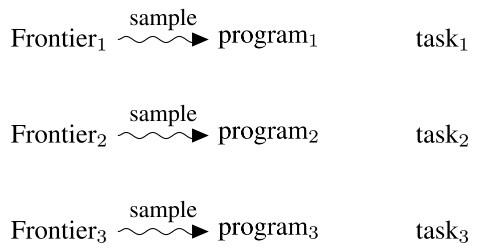
Intuitively, this 3-phase inference procedure can work in practice because each of the 3 phases bootstraps off of the others (Figure 1). As the DSL grows and as the recognition model becomes more accurate, waking becomes more effective, allowing the agent to solve more tasks; when we solve more tasks 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

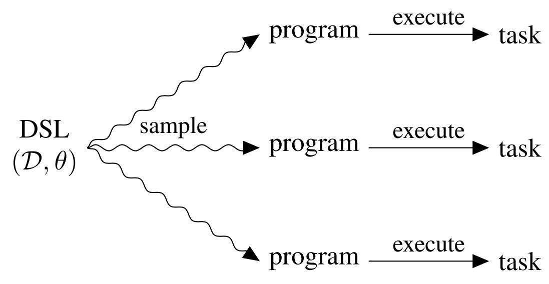
During waking, the agent's goal is to search for programs solving the tasks. We use the simple approach of enumerating programs from the DSL in decreasing order of their probability according to the recognition model, and then checking



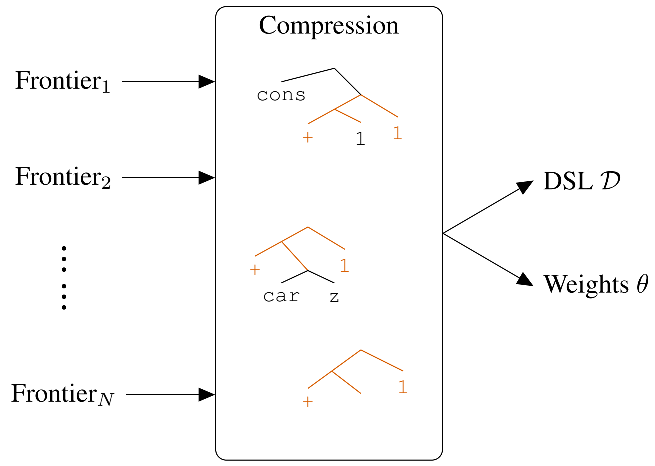
WAKE: PROBLEM SOLVING


Frontier$_1$ sample program$_1$ task$_1$
Frontier$_2$ sample program$_2$ task$_2$
Frontier$_3$ sample program$_3$ task$_3$

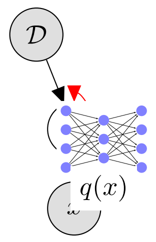
SLEEP-R: EXPERIENCE REPLAY



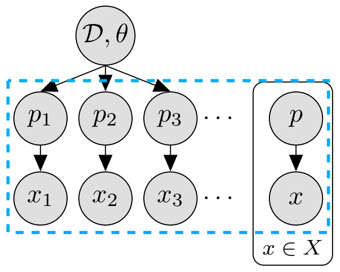
SLEEP-R: DREAMING



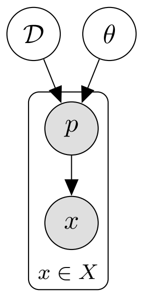
SLEEP-G: MEMORY CONSOLIDATION



Wake: Infer p



Sleep-R: Train q
training data: cyan (x, p)



Sleep-G: Induce $(\mathcal{D}, \theta)$

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 will write λ-calculus expressions in Lisp syntax. Our programs are all strongly typed. We use the Hindley-Milner polymorphic typing system [Pierce(2002)] which is used in functional programming languages like OCaml and Haskell. We now define DSLs:

Definition: $(\mathcal{D}, \theta)$**.** 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, $\mathbb{P}[p|\mathcal{D}, \theta]$. In the supplement, we define this distribution by specifying a procedure for drawing samples from $\mathbb{P}[p|\mathcal{D}, \theta]$. Care must be taken to ensure that programs are well-typed and that variable scoping rules are obeyed.

Why enumerate, when the program synthesis community has invented many sophisticated algorithms that search for programs? [Solar Lezama(2008), Schkufza et al.(2013)Schkufza, Sharma, and Aiken, Feser et al.(2015)Feser, Chaudhuri, and Dillig, Osera & Zdancewic(2015)Osera and Zdancewic, Polozov & Gulwani(2015)Polozov and Gulwani]. 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. 5.2, 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 $(\mathcal{D}, \theta)$. Concretely, the recognition model Q predicts, for each task $x \in X$, a distribution over programs, $Q(\cdot|x)$. We want $Q(\cdot|x)$ to approximate 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 nonobvious because we are considering a weakly supervised setting (i.e., learning only from tasks and not from (program, task) pairs). One approach is to sample 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 a Helmholtz machine trains its recognition model during its "sleep" phase [Dayan et al.(1995)Dayan, Hinton, Neal, and Zemel]. The advantage of "Helmholtz machine" training is that we can draw unlimited samples from the DSL, training on a large amount of data. Another approach is self-supervised learning, training q on the (program, task) pairs discovered during waking. The advantage of self-supervised learning is that the training data is much higher quality, because we are training on the actual tasks. Due to these complementary advantages, we train on both these sources of data.

Formally, q should approximate the true posteriors over programs: minimizing the expected KL-divergence, $\mathbb{E}\left[\text{KL}\left(\mathbb{P}[p|x,\mathcal{D},\theta]\|q(p|x)\right)\right]$, equivalently maximizing $\mathbb{E}[\sum_p \mathbb{P}[p|x,\mathcal{D},\theta]\log q(p|x)]$, where the expectation is taken over tasks. Taking this expectation over the empirical distribution of tasks gives self-supervised training; taking it over samples from the generative model gives Helmholtz-machine style training. The objective for a recognition model ($\mathcal{L}_{\text{RM}}$) combines the Helmholtz machine ($\mathcal{L}_{\text{HM}}$) and self supervised ($\mathcal{L}_{\text{SS}}$) objectives:

$$\begin{aligned}
\mathcal{L}_{\text{RM}} &= \mathcal{L}_{\text{SS}} + \mathcal{L}_{\text{HM}} \\
\mathcal{L}_{\text{HM}} &= \mathbb{E}_{(p,x)\sim(\mathcal{D},\theta)}\left[\log q(p|x)\right] \\
\mathcal{L}_{\text{SS}} &= \mathbb{E}_{x\sim X}\left[\sum_{p\in\mathcal{F}_x}\frac{\mathbb{P}\left[x,p|\mathcal{D},\theta\right]}{\sum_{p'\in\mathcal{F}_x}\mathbb{P}\left[x,p'|\mathcal{D},\theta\right]}\log q(p|x)\right]
\end{aligned} \tag{3}$$

The $\mathcal{L}_{\text{HM}}$ objective is 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 q. Once we bootstrap a $(\mathcal{D},\theta)$, we can draw unlimited samples from $(\mathcal{D},\theta)$ and train q on those samples.

Evaluating $\mathcal{L}_{\text{HM}}$ involves sampling 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.

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 [Devlin et al.(2017b)Devlin, Uesato, Bhupatiraju, Singh, Mohamed, and Kohli] and [**?**], is to use a recurrent neural network to predict the program token-by-token ala 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 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 [Feser et al.(2015)Feser, Chaudhuri, and Dillig].

The second approach is to have Q predict a fixed-dimensional weight vector, which then biases a fast enumerator [Balog et al.(2016)Balog, Gaunt, Brockschmidt, Nowozin, and Tarlow] or sampler [Menon et al.(2013)Menon, Tamuz, Gulwani, Lan
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). See Figure **??**. 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.

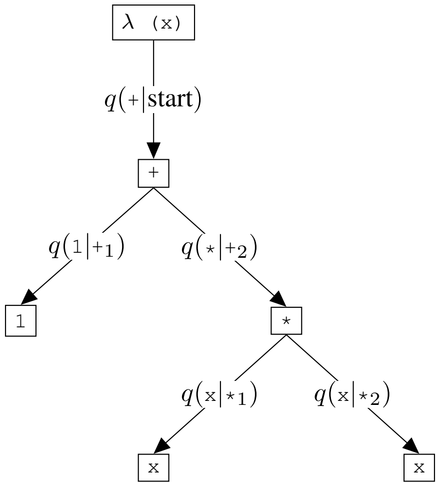


Figure 2: Parameterization of distribution over programs predicted by recognition model. Here the program (syntax tree shown above) is `(λ (x) (+ 1 (* x x)))`. Each conditional distribution predicted by the recognition model is written $q(\text{child}|\text{parent}_{\text{argument index}})$.

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:

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

At a high level, our approach is to search locally through the space of DSLs, proposing small changes to $\mathcal{D}$ until Eq. 4 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 3 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:

$$\log \mathbb{P}[\mathcal{D}] + \arg\max_\theta \Bigg(\log \mathbb{P}[\theta|\mathcal{D}] - \|\theta\|_0 + \sum_{x \in X} \log \sum_{p \in \mathcal{F}_x} \mathbb{P}[x|p] \max_{p' \longrightarrow^* p} \mathbb{P}[p'|\mathcal{D},\theta] \Bigg) \tag{5}$$

where $p' \longrightarrow^* p$ is the standard notation for "expression p' evaluates to p by the rules of λ-calculus" [Pierce(2002)].

Equation 5 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:

$$R_n(p) = \left\{ p' \;:\; p' \underbrace{\longrightarrow p'' \longrightarrow \cdots \longrightarrow}_{\leq n \text{ times}} p \right\} \tag{6}$$

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" [Pierce(2002)]. For example,

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

Returning to Equation 5, this approximation gives the following objective:

$$\log \mathbb{P}[\mathcal{D}] + \arg\max_{\theta} \Bigg(\log \mathbb{P}[\theta|\mathcal{D}] - \|\theta\|_0 + \sum_{x \in X} \log \sum_{p \in \mathcal{F}_x} \mathbb{P}[x|p] \max_{p' \in R_n(p)} \mathbb{P}[p'|\mathcal{D}, \theta] \Bigg) \tag{7}$$

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 3, 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* [Lau(2001), **?**, Polozov & Gulwani(2015)Polozov and Gulwani]. 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 3, 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[4] 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:

[4]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 [Pierce(2002)] for more details.

Definition 1. *The **extension** of a version space v is written $[\![v]\!]$ and is defined recursively as:*

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

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 $\textsc{refactor}(v|\mathcal{D})$ inductively as:

$$\textsc{refactor}(v|\mathcal{D}) = \begin{cases} e, \text{ if } e \in \mathcal{D} \text{ and } e \in v. \text{ Exploits the fact that } e \in v \text{ can be efficiently computed.} \\ \textsc{refactor}'(v|\mathcal{D}), \text{ otherwise.} \end{cases} \tag{8}$$

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

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

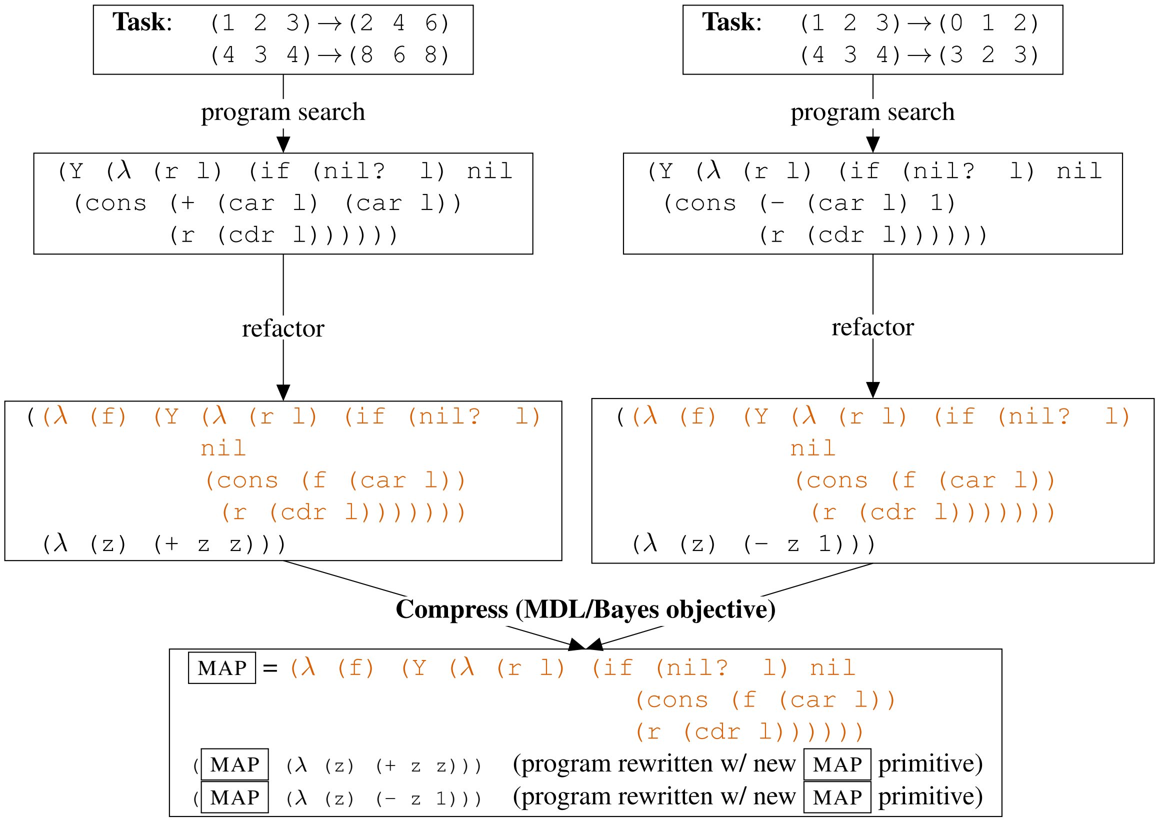
In Appendix A.1, we describe a dynamic program for efficiently constructing a version space containing every n-step refactoring. This dynamic program, which we call $I\beta^n(p)$, satisfies $[\![I\beta^n(p)]\!] = R_n(p)$. In other words, this dynamic program builds a data structure that represents the 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 1). To define the prior distribution over $(\mathcal{D}, \theta)$ (Algorithm 1, lines 7-8), we penalize the syntactic complexity of the λ-calculus expressions in the DSL, defining $\mathbb{P}[\mathcal{D}] \propto \exp(-\lambda \sum_{p \in \mathcal{D}} \text{size}(p))$ where size(p) measures the size of the syntax tree of program p, and λ controls how strongly we regularize the size of the DSL. We place a symmetric Dirichlet prior over the weight vector θ.

To appropriately score each proposed $\mathcal{D}$ we must reestimate the weight vector θ (Algorithm 1, line 7). Although this may seem very similar to estimating the parameters of a probabilistic context free grammar, for which we have effective approaches like the Inside/Outside algorithm [Lafferty()], our DSLs are context-sensitive due to the presence of variables in the programs and also due to the polymorphic typing system. Appendix A.2 derives a tractable MAP estimator for θ.

Algorithm 1 DSL Induction Algorithm

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



Task: (1 2 3)→(2 4 6) (4 3 4)→(8 6 8)
Task: (1 2 3)→(0 1 2) (4 3 4)→(3 2 3)
program search
program search
(Y (λ (r l) (if (nil? l) nil (cons (+ (car l) (car l)) (r (cdr l))))))
(Y (λ (r l) (if (nil? l) nil (cons (- (car l) 1) (r (cdr l))))))
refactor
refactor
((λ (f) (Y (λ (r l) (if (nil? l) nil (cons (f (car l)) (r (cdr l))))))) (λ (z) (+ z z)))
((λ (f) (Y (λ (r l) (if (nil? l) nil (cons (f (car l)) (r (cdr l))))))) (λ (z) (- z 1)))
Compress (MDL/Bayes objective)
MAP = (λ (f) (Y (λ (r l) (if (nil? l) nil (cons (f (car l)) (r (cdr l)))))))
(MAP (λ (z) (+ z z))) (program rewritten w/ new MAP primitive)
(MAP (λ (z) (- z 1))) (program rewritten w/ new MAP primitive)

Figure 3: 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 [Cho et al.(2014)Cho, Van Merriënboer, Gulcehre, Bahdanau, B
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 [Feser et al.(2015)Feser, Chaudhuri, and Dillig]. 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 1, left).

Text Editing: Synthesizing programs that edit text is a classic problem in the programming languages and AI literatures [Gulwani(2011), Lau(2001)]. 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. 1, 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 [Alur et al.(2016)Alur, Fisman, Singh, and Solar-Lezama] 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 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. 4) — 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 [Bishop(2006)]. 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. 1 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.

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.

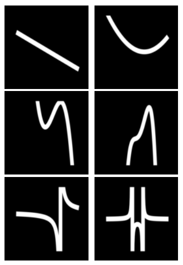


Figure 4: Recognition model input for symbolic regression. DSL learns subroutines for polynomials (top rows) and rational functions (bottom) while the recognition model jointly learns to look at a graph of the function (left) and predict which learned subroutines best explain the observation.

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*) f_2(f,l,x) = (if (empty? l) x (f (car l) (f_2 (cdr l)))) (f_2: *fold*)
[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_3(f,l) = (f_2 nil l (λ (x a) (cons (f x) a))) (f_3: *map*)
[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)	$f_4(\ell)$ = (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 5: Bootstrapping a standard library of functional programming routines, starting from recursion along with primitive operations found in 1959 Lisp.

5.3 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 [Solomonoff(1964), Schmidhuber(2004), Hutter(2004), Solomonoff(1989)]. 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 [McCarthy(1960)]: 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 5). We did not use the recognition model for this experiment: a bottom-up pattern recognizer is of little use for acquiring this abstract knowledge from less than two dozen problems.

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

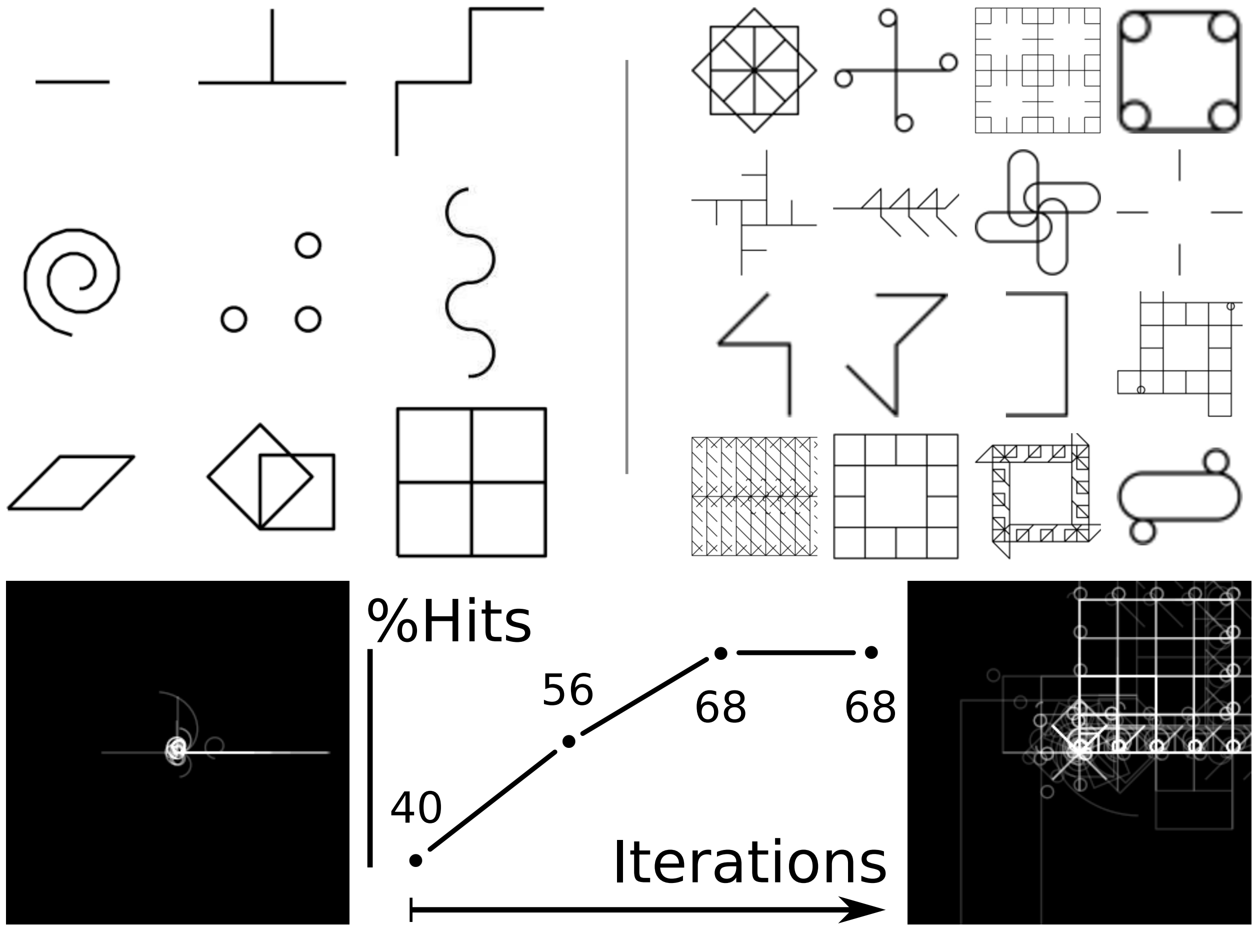


Figure 6: Top left: Example training tasks. Top right: samples from the learned DSL. Bottom: % holdout testing tasks solved (middle); on the sides are averaged samples from the DSL before any training (left) and after last iteration (right).

programming systems, provided it is given enough computational power and a suitable spectrum of tasks.

5.4 Learning Generative Models

We apply DREAMCODER to learning generative models for images and text (Fig. 6-**??**). 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.

5.5 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. 8). 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.

Learned DSL:

$f_1() =$ `\u\w*`
$f_2(x) =$ `(`x`|`f_1`)*` $=$ `(`x`|\u\w*)*`
$f_3(x) = f_2($`space`$) =$ `( |\u\w*)*`

$f_4(x) =$ `(`x`*`x`)`
(equivalent to regex 'plus')
$f_5() = f_4($`\l`$) =$ `\l*\l`

Tasks:

1.14531	F	110.9
?	CL	163.2
1.29857	F	207.3
?	PCFL	143.3

Learned generative models:

`\?|(1\.\d+)`	`((\u\u)*)|F`	`\d\d\d\.\d`

Samples from synthesized generative models:

1.61	DQDF	343.8
?	F	241.2
?	F	647.5
1.2	KI	246.8
1.5080987	F	728.6
?	GL	029.3
1.453	F	289.1

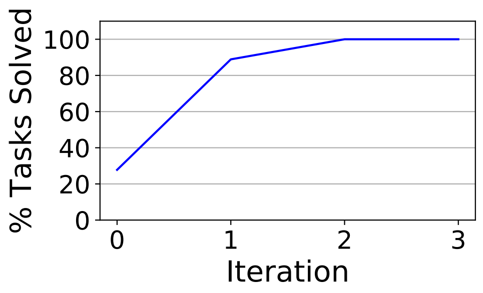


Figure 7: regex stuff

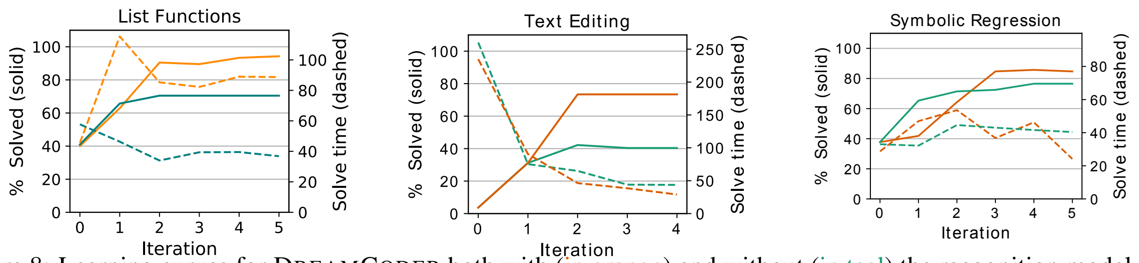


Figure 8: Learning curves for DREAMCODER both with (in orange) and without (in teal) the recognition model. Solid lines: % holdout testing tasks solved w/ 10m timeout. Dashed lines: Average solve time, averaged only over tasks that are solved.

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 [Devlin et al.(2017b)Devlin, Uesato, Bhupatiraju, Singh, Mohamed, and Kohli] and program synthesizers [Solar Lezama(2008)], 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

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

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

[Bishop(2006)] Bishop, Christopher M. *Pattern Recognition and Machine Learning*. 2006.

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

[Cohn et al.()Cohn, Blunsom, and Goldwater] Cohn, Trevor, Blunsom, Phil, and Goldwater, Sharon. Inducing tree-substitution grammars. *JMLR*.

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

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

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

[Devlin et al.(2017b)Devlin, Uesato, Bhupatiraju, Singh, Mohamed, and Kohli] Devlin, Jacob, Uesato, Jonathan, Bhupatiraju, Surya, Singh, Rishabh, Mohamed, Abdel-rahman, and Kohli, Pushmeet. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*, 2017b.

[Dudai et al.(2015)Dudai, Karni, and Born] Dudai, Yadin, Karni, Avi, and Born, Jan. The consolidation and transformation of memory. *Neuron*, 88(1):20 – 32, 2015. ISSN 0896-6273. doi: https://doi.org/10.1016/j.neuron.2015.09.004. URL `http://www.sciencedirect.com/science/article/pii/S0896627315007618`.

[Ellis et al.(2017)Ellis, Ritchie, Solar-Lezama, and Tenenbaum] Ellis, Kevin, Ritchie, Daniel, Solar-Lezama, Armando, and Tenenbaum, Joshua B. Learning to infer graphics programs from hand-drawn images. *arXiv preprint arXiv:1707.09627*, 2017.

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

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

[Ganin et al.(2018)Ganin, Kulkarni, Babuschkin, Eslami, and Vinyals] Ganin, Yaroslav, Kulkarni, Tejas, Babuschkin, Igor, Eslami, S. M. Ali, and Vinyals, Oriol. Synthesizing programs for images using reinforced adversarial learning. *CoRR*, abs/1804.01118, 2018. URL `http://arxiv.org/abs/1804.01118`.

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

[Henderson()] Henderson, Robert John. Cumulative learning in the lambda calculus.

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

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

[Hwang et al.(2011)Hwang, Stuhlmüller, and Goodman] Hwang, Irvin, Stuhlmüller, Andreas, and Goodman, Noah D. Inducing probabilistic programs by bayesian program merging. *CoRR*, abs/1110.5667, 2011. URL `http://arxiv.org/abs/1110.5667`.

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

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

[Koza()] Koza, John R.

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

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

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

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

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

[Lin et al.(2014)Lin, Dechter, Ellis, Tenenbaum, and Muggleton] Lin, Dianhuan, Dechter, Eyal, Ellis, Kevin, Tenenbaum, Joshua B., and Muggleton, Stephen. Bias reformulation for one-shot function induction. In *ECAI 2014*, 2014.

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

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

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

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

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

[Pierce(2002)] Pierce, Benjamin C. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

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

[Schkufza et al.(2013)Schkufza, Sharma, and Aiken] Schkufza, Eric, Sharma, Rahul, and Aiken, Alex. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 41, pp. 305–316. ACM, 2013.

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

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

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

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

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

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

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

A Appendix

A.1 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:

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

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

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

where $\uparrow^k$ is the shifting operator [Pierce(2002)], 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 8 for the recursive definition of REFACTOR).

A.1.1 Computational complexity of DSL learning

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

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

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

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

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

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.2 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

$$\theta = \arg\max_\theta \log P(\theta) + \sum_x \mathbb{E}_{Q_x}\left[\log \mathbb{P}\left[p|\theta\right]\right] \tag{9}$$

$$Q_x(p) \propto \mathbb{P}[x|p]\mathbb{P}[p|\theta]\mathbb{1}\left[p\in\mathcal{F}_x\right] \tag{10}$$

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

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

where we have defined

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

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

$$\sum_e (C(e)+\alpha)\log\theta_e - \beta\log\sum_\tau \frac{R(\tau)}{\beta} \sum_{\substack{e:\tau'\in\mathcal{D}\\ \text{unify}(\tau,\tau')}} \theta_e \tag{11}$$

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

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

$$\theta_e \propto \frac{C(e)+\alpha}{\sum_{\tau'} \mathbb{1}\left[\text{unify}(\tau,\tau')\right] R(\tau')} \tag{13}$$

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.