DREAMCODER: Bootstrapping Domain-Specific Languages for Neurally-Guided Bayesian Program Learning

Anonymous Authors[1]

1. Introduction

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), procedural graphics (Ellis et al., 2017; Ganin et al., 2018), question answering (Johnson et al., 2017) and robot motion planning (Devlin et al., 2017a). 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, 1993), 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)), 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). The recognition model ensures that searching

	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_6 x`)`
			  f(x) = `(`f_4 x`)`
			  f(x) = `(`f_3 x`)`

DSL	$f_1(\ell,p)$ = `(foldr` ℓ `nil (λ (x a) (if (p x) (cons x a) a)))` (f_1: *Higher-order filter function*) $f_2(\ell)$ = `(foldr` ℓ `0 (λ (x a) (if (> a x) a x)))` (f_2: *Maximum element in list* ℓ) $f_3(\ell,k)$ = `(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)

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). 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), FlashFill: (Gulwani, 2011)), neural approaches (e.g., RobustFill: (Devlin et al., 2017b)), and hybrids of neural and symbolic methods (e.g., Neural-guided deductive search: (Kalyan et al., 2018), DeepCoder: (Balog et al., 2016)). A notable exception is the EC algorithm (Dechter et al., 2013), 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

2. The DREAMCODER Algorithm

We first mathematically describe our 3-step algorithm as an inference procedure for a hierarchical Bayesian model (Section 2.1), and then describe each step algorithmically in detail (Section 2.2-2.4).

2.1. 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

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

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

[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.

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.

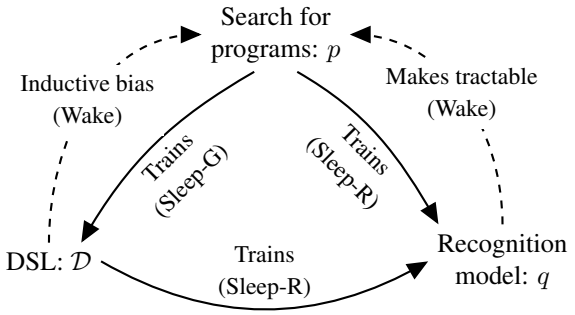


Figure 1: DREAMCODER solves for programs, the DSL, and a recognition model. Each of these steps bootstrap off of the others in a Helmholtz-machine inspired wake/sleep inference algorithm.

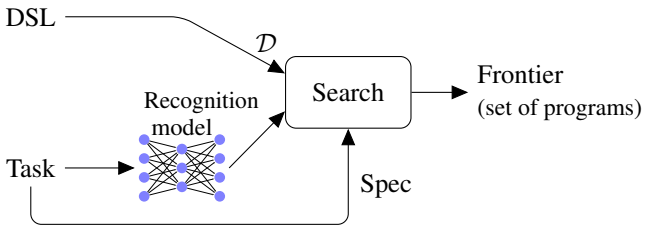
2.2. Wake: Searching for Programs

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 $\mathcal{F}_x$.

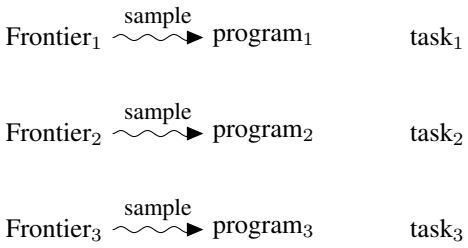
To make this concrete we need to define what programs actually are and what form $\mathbb{P}[p|\mathcal{D},\theta]$ takes. We represent programs as λ-calculus expressions. λ-calculus is a formalism for expressing functional programs that closely resembles Lisp, including variables, function application,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

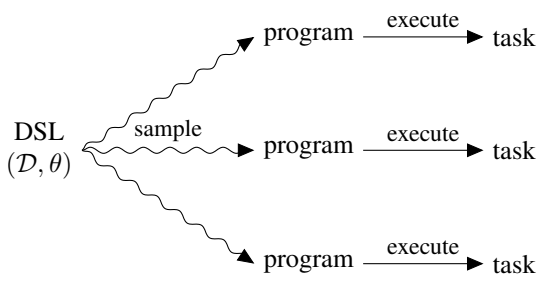
Together with its weight vector, a DSL defines a distribution over programs, $\mathbb{P}[p|\mathcal{D},\theta]$. In the supplement, we define this distribution by specifying a procedure for drawing samples from $\mathbb{P}[p|\mathcal{D},\theta]$. Care must be taken to ensure that programs



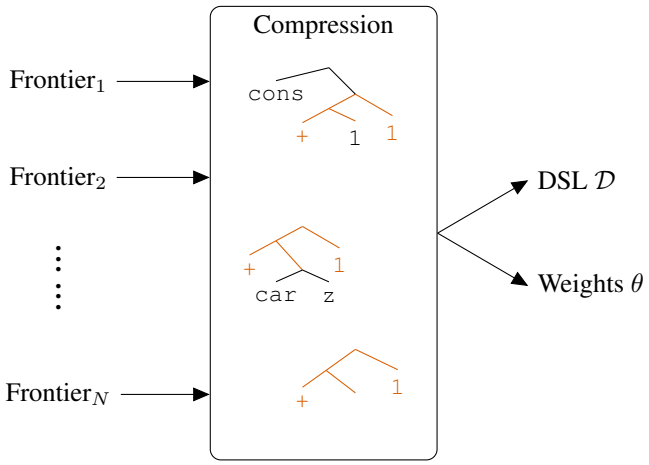
WAKE: PROBLEM SOLVING



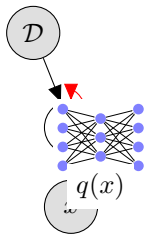
SLEEP-R: EXPERIENCE REPLAY



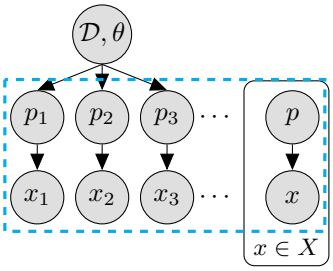
SLEEP-R: DREAMING



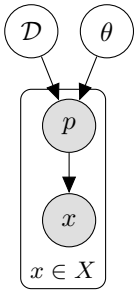
SLEEP-G: MEMORY CONSOLIDATION



Wake: Infer p



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



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

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; Feser et al., 2015; Osera & Zdancewic, 2015; Polozov & Gulwani, 2015). 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. 3.2, we will show how to find programs with real-valued constants by automatically differentiating through the program and setting the constants using gradient descent.

2.3. 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 weight vector $q(x) = \theta^{(x)} \in \mathbb{R}^{|\mathcal{D}|+1}$. Together with the DSL, this defines a distribution over programs, $\mathbb{P}[p|\mathcal{D}, \theta = q(x)]$. We abbreviate this distribution as $q(p|x)$. The crucial aspect of this framing is that the neural network leverages the structure of the learned DSL,

so it is *not* responsible for generating programs wholesale. We share this aspect with DeepCoder (Balog et al., 2016) and (Menon et al., 2013).

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). 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, $\mathbb{E}\left[\mathrm{KL}\left(\mathbb{P}[p|x,\mathcal{D},\theta]\|q(p|x)\right)\right]$, 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:

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

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.

2.4. Sleep-G: Learning a Generative Model (a DSL)

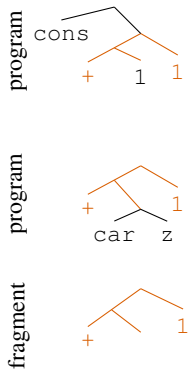
The purpose of the DSL is to offer a set of abstractions that allow an agent to easily express solutions to the tasks at hand. Intuitively, we want the algorithm to look at the frontiers and generalize beyond them, both so the DSL can better express the current solutions, and also so that the DSL might 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. 2.1). We replace this marginal with an AIC approximation, giving the following objective for DSL induction:

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

We induce a DSL by searching locally through the space of DSLs, proposing small changes to $\mathcal{D}$ until Eq. 4 fails to increase. The search moves work by introducing new λ-expressions into the DSL. We propose these new expressions by extracting fragments of programs already in the frontiers (Tbl. 2). An important point here is that we are *not* simply adding subexpressions of programs to $\mathcal{D}$, as done in the EC algorithm (Dechter et al., 2013) and other prior work (Lin et al., 2014). Instead, we are extracting fragments that unify with programs in the frontiers. This idea of storing and reusing fragments of expressions comes from Fragment Grammars (O'Donnell, 2015) and Tree-Substitution Grammars (Cohn et al.), and is closely related to the idea of antiunification (Henderson).

To define the prior distribution over $(\mathcal{D},\theta)$, we penalize the syntactic complexity of the λ-calculus expressions in the DSL, defining $\mathbb{P}[\mathcal{D}] \propto \exp(-\lambda\sum_{p\in\mathcal{D}}\mathrm{size}(p))$ where $\mathrm{size}(p)$ 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, defining $\mathbb{P}[\theta|\mathcal{D}] = \mathrm{Dir}(\theta|\alpha)$, where α is a concentration parameter controlling the smoothness of the prior over θ.

To appropriately score each proposed $\mathcal{D}$ we must reestimate the weight vector θ. 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. In the Supplement we derive a tractable MAP estimator for θ.



Example programs in frontiers	Proposed λ-expression
`(λ (ℓ) (map (λ (x) (index x ℓ)) (range (- (length ℓ) 1))))`	`(map (λ (x) (index x ℓ)) (range α))`
`(λ (ℓ) (map (λ (x) (index x ℓ)) (range (+ 1 1))))`	
`(λ (s) (map (λ (x) (if (= x '.') '-' x))) s)`	`(λ (s) (map (λ (x) (if (= x α) β x))) s)`
`(λ (s) (map (λ (x) (if (= x '-') ',' x))) s)`	

Figure 2: **Left:** syntax trees of two programs sharing common structure, highlighted in orange, from which we extract a fragment and add it to the DSL (bottom). **Right:** actual programs, from which we extract fragments that (top) slice from the beginning of a list or (bottom) perform character substitutions.

3. Experiments

3.1. Programs that manipulate sequences

We apply DREAMCODER to list processing and text editing, using a GRU (Cho et al., 2014) 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). 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.

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.

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) program synthesis competition. Before any learning, DREAMCODER solves 3.7% of the problems with an average search time of 235 seconds. After learning, it solves 74.1%, and does so much faster, solving them in an average of 29 seconds. As of the 2017 SyGuS competition, the best-performing algorithm solves 79.6% of the problems. But, SyGuS comes with a different hand-engineered DSL *for each text editing problem*. Here we learned a single DSL that applied generically to all of the tasks, and perform comparably to the best prior work.

3.2. 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. 3) — 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, $\mathbb{P}[x|p]$,

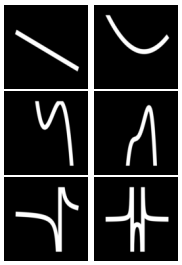


Figure 3: Recognition model input for symbolic regression. DSL learns subroutines for polynomials (top rows) and rational functions (bottom) while the recognition model jointly learns to look at a graph of the function (left) and predict which learned subroutines best explain the observation.

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.

3.3. 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. 3.1) that we initially provided our system with functional programming routines like `map` and `fold`.

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

After running for 93 hours on 64 CPUs, our algorithm solves these 22 exercises, along the way assembling a DSL with a modern repertoire of functional programming idioms and subroutines, including `map`, `fold`, `zip`, `unfold`, `index`, `length`, and arithmetic operations like building lists of natural numbers between an interval (see Figure 4). We did not use the recognition model for this experiment: a bottom-up pattern recognizer is of little use for acquiring this abstract knowledge from less than two dozen problems.

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

3.4. Learning Generative Models

We apply DREAMCODER to learning generative models for images and text (Fig. 5-**??**). For images, we learn programs controlling a simulated "pen," and the task is to look at an image and explain it in terms of a graphics program. For text, we learn probabilistic regular expressions – a simple probabilistic program for which inference is always tractable – and the task is to infer a regex from a collection of strings.

3.5. 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. 7). Prior to any learning, the system cannot find solutions for most of the tasks, and those it does solve take a long time; with more wake/sleep iterations, we converge upon DSLs and recognition models more closely matching the domain.

Programs & Tasks

```
[4 7 6 9 0]→5
[3 8 2 1 4]→4
[2 2 9 4 8]→3
[0 9 8 9 9 8]→1
f(ℓ)=(f0 (car ℓ))

[2 5 6 0 6]→19
[9 2 7 6 3]→27
f(ℓ)=(f4 + ℓ 0)

[4 2 6 4]→[8 4 12 8]
[0 3 5 0 3 2]→[0 6 10 0 6 4]
[2 3 0 7]→[4 6 0 14]
f(ℓ)=(f5 (λ (x) (+ x x)) ℓ)

[5 2 9]→[[2 9] [9] []]
[3 8 1 3]→[[8 1 3] [1 3] [3] []]
f(ℓ)=(f2 empty? cdr cdr ℓ)
```

DSL

```
f0(x) = (+ x 1)
  (f0: increment)
f1(x) = (- x 1)
  (f1: decrement)
f2(p,f,n,x) = (if (p x) nil
                  (cons (f x) (f2 (n x))))
  (f2: unfold)
f3(i,l) = (if (= i 0) (car l)
              (f3 (f1 i) (cdr l))))
  (f3: index)
f4(f,l,x) = (if (empty? l) x
                (f (car l) (f4 (cdr l))))
  (f4: fold)
f5(f,l) = (if (empty? l) nil)
              (cons (f (car l)) (f5 (cdr l)))
  (f5: map)
```

Figure 4: Bootstrapping a standard library of functional programming routines, starting from recursion along with primitive operations found in 1959 Lisp.

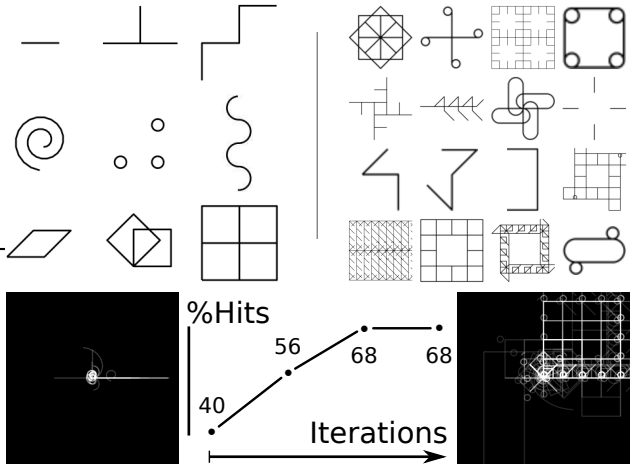


Figure 5: Top left: Example training tasks. Top right: samples from the learned DSL. Bottom: % holdout testing tasks solved (middle); on the sides are averaged samples from the DSL before any training (left) and after last iteration (right).

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

Learned DSL:

$f_1() = \u\w*$
$f_2(x) = (x|f_1)* = (x|\u\w*)*$
$f_3(x) = f_2(\text{space}) = (\ |\u\w*)*$
$f_4(x) = (x*x)$
(equivalent to regex 'plus')
$f_5() = f_4(\l) = \l*\l$

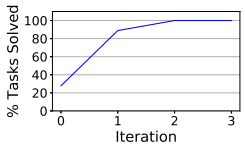


Figure 6: regex stuff

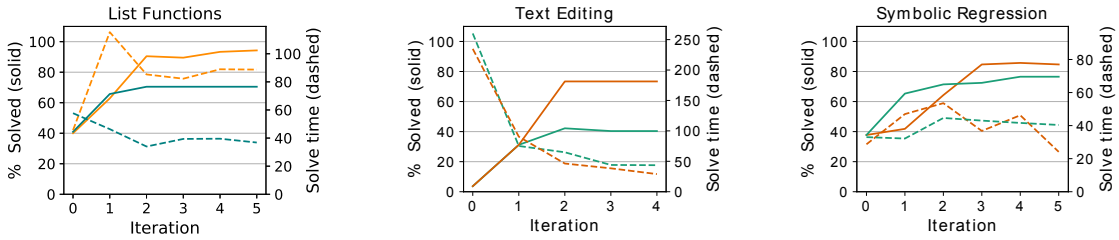


Figure 7: Learning curves for DREAMCODER both with (in orange) and without (in teal) the recognition model. Solid lines: % holdout testing tasks solved w/ 10m timeout. Dashed lines: Average solve time, averaged only over tasks that are solved.

4. Discussion

We contribute an algorithm, DREAMCODER, that learns to program by bootstrapping a DSL with new domain-specific primitives that the algorithm itself discovers, together with a neural recognition model that learns to efficiently deploy the DSL on new tasks. Two immediate future goals are to integrate more sophisticated neural recognition models (Devlin et al., 2017b) 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, Rajeev, Fisman, Dana, Singh, Rishabh, and Solar-Lezama, Armando. Sygus-comp 2016: results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.

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

Bishop, Christopher M. *Pattern Recognition and Machine Learning*. 2006.

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, Trevor, Blunsom, Phil, and Goldwater, Sharon. Inducing tree-substitution grammars. *JMLR*.

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

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

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

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

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

Henderson, Robert John. Cumulative learning in the lambda calculus.

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

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 *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pp. 1988–1997. IEEE, 2017.

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, John R. *Genetic programming - on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, 1993. ISBN 978-0-262-11170-6.

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

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

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

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

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

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

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, 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, Timothy J. *Productivity and Reuse in Language: A Theory of Linguistic Computation and Storage*. The MIT Press, 2015.

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

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

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

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

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

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

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

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