DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning

Kevin Ellis
Cornell, USA
kellis@cornell.edu

Catherine Wong
MIT, USA
zyzzyva@mit.edu

Maxwell Nye
MIT, USA
mnye@mit.edu

Mathias Sablé-Meyer
PSL/Collège de France & NeuroSpin, France
mathias.sable-meyer@ens-cachan.fr

Lucas Morales
MIT, USA
lucas@lucasem.com

Luke Hewitt
MIT, USA
lbh@mit.edu

Luc Cary
MIT, USA
luc.cary@gmail.com

Armando Solar-Lezama
MIT, USA
asolar@csail.mit.edu

Joshua B. Tenenbaum
MIT, USA
jbt@mit.edu

Abstract

We present a system for inductive program synthesis called DreamCoder, which inputs a corpus of synthesis problems each specified by one or a few examples, and automatically derives a library of program components and a neural search policy that can be used to efficiently solve other similar synthesis problems. The library and search policy bootstrap each other iteratively through a variant of "wake-sleep" approximate Bayesian learning. A new refactoring algorithm based on E-graph matching identifies common sub-components across synthesized programs, building a progressively deepening library of abstractions capturing the structure of the input domain. We evaluate on eight domains including classic program synthesis areas and AI tasks such as planning, inverse graphics, and equation discovery. We show that jointly learning the library and neural search policy leads to solving more problems, and solving them more quickly.

CCS Concepts: • **Software and its engineering → Software notations and tools**; • **Computing methodologies → Machine learning**.

Keywords: synthesis, neural, learning, refactoring

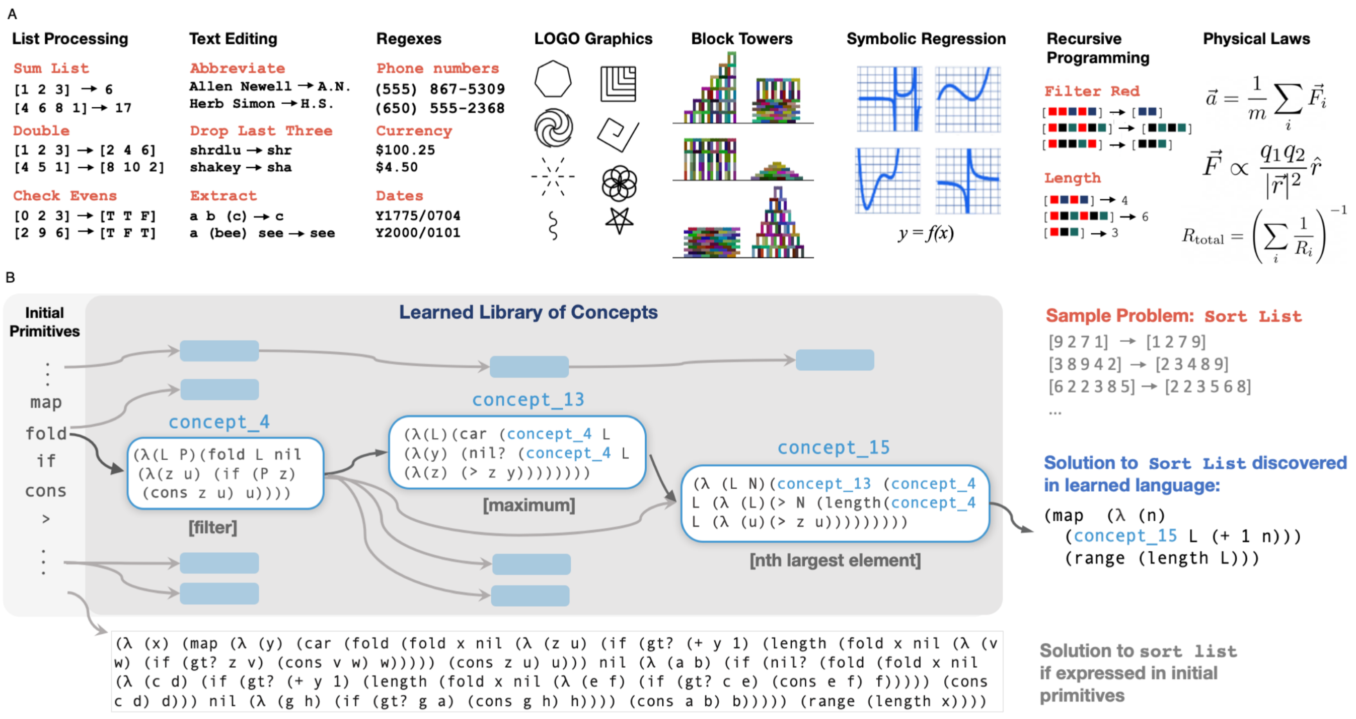
ACM Reference Format:
Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454080

1 Introduction

Inductive program synthesis – synthesizing programs given one or more examples of their behavior – has emerged as both a practical tool for automating end-user programming tasks [20, 43, 51], and a route to building more robust, interpretable, generalizable and sample-efficient learning systems in artificial intelligence (AI) [27, 29, 37]. A key insight behind many of these advances is to simplify the synthesis problem significantly by providing a suitable *structure hypothesis* [46]. This structure hypothesis may take the form of a program sketch, or a restricted domain specific language (DSL), but the general idea is the same: making program search tractable by restricting it to a limited space of possible expressions.

Unfortunately, the need for a strong structure hypothesis also limits the value of synthesis systems. Many success stories in inductive synthesis rely on specialized DSLs designed specifically for one task domain (*e.g.* FlashFill [20]). Even general-purpose synthesis tools such as Synquid [41] require the user to provide a carefully targeted library of components in order to restrict search. The expertise required to design such libraries and DSLs limits the applicability of inductive synthesis in novel real-world domains, and also limits its scalability as a means of general-purpose learning in AI.

Here we present DreamCoder, a program synthesizer which learns its own structure hypothesis in the form of a library of components. DreamCoder follows the learning-to-synthesize paradigm [2, 13], where a corpus of synthesis



```
concept_4:
(λ(L P)(fold L nil
(λ(z u) (if (P z)
(cons z u) u))))

concept_13:
(λ(L)(car (concept_4 L
(λ(y) (nil? (concept_4 L
(λ(z) (> z y))))))))

concept_15:
(λ (L N)(concept_13 (concept_4
L (λ (L)(> N (length(concept_4
L (λ (u)(> z u)))))))))
```

```
(map  (λ (n)
  (concept_15 L (+ 1 n)))
  (range (length L)))
```

```
(λ (x) (map (λ (y) (car (fold (fold x nil (λ (z u) (if (gt? (+ y 1) (length (fold x nil (λ (v
w) (if (gt? z v) (cons v w) w))))) (cons z u) u))) nil (λ (a b) (if (nil? (fold (fold x nil
(λ (c d) (if (gt? (+ y 1) (length (fold x nil (λ (e f) (if (gt? c e) (cons e f) f))))) (cons
c d) d))) nil (λ (g h) (if (gt? g a) (cons g h) h)))) (cons a b) b))))) (range (length x))))
```

Figure 1. (A): Eight problem-solving domains DreamCoder is applied to. **(B)**: Given initial primitives (left), DreamCoder iteratively builds a library of more advanced functions (middle) and uses this library to solve problems too complex to be solved initially. Each learned function can call functions learned earlier (arrows), forming hierarchically organized layers of functions. A typical solution to 'Sort List' (right), discovered after six iterations of learning is found in less than 10 minutes of search. At bottom the model's solution is re-expressed in terms of only the initial primitives, yielding a long and cryptic program with 32 function calls, which would take in excess of 10^{72} years of brute-force search to discover.

problems is used to train a neural search policy that can then be used to efficiently search for solutions to similar problems. We go further by learning the library of components that defines the search space in tandem with training the search policy. The learned library is embodied in a probabilistic grammar for programs: a generative model that is used both to score candidate programs (to find the most probable solution to each task, in a Bayesian sense), and also to sample random tasks for training the neural search policy. Yet a DSL is more than a library: it also carefully constrains the ways in which library components can combine. Our search policy, trained alongside the learned library and scaling with it, approximates these constraints by learning to break syntactic symmetries in the search over programs.

DreamCoder draws on prior work in library learning [10, 14, 21, 31]. A key novelty in the DreamCoder system is an automatic refactoring algorithm that is able to take a collection of synthesized programs and extract from them a set of components that can be used to more compactly represent each of these programs. The algorithm leverages E-graph matching [11] to identify rewrites to the original programs that may expose more common patterns. As the library grows, the learned neural search policy is trained to guide synthesis while breaking new symmetries in search introduced by the new library components.

The resulting system has wide applicability. We describe applications to eight domains including classic program synthesis challenges, more creative visual drawing and building problems, and finally, library learning that captures the basic languages of recursive programming, vector algebra, and physics (Fig. 1A). All of our tasks involve inducing programs from very minimal data, e.g., 5-10 examples of a new concept or function, or a single image or scene depicting a new object. The learned languages span deterministic and probabilistic programs, and programs that act both generatively (e.g., producing an artifact like an image or plan) and conditionally (e.g., mapping inputs to outputs). In total, we contribute the following: (1) a new approach to mining reusable program components based on E-graph matching; (2) a new system that learns a library and a recognition model from a corpus of synthesis problems; and (3) an evaluation across 8 domains.

2 Background and Overview

This section provides a high-level overview of DreamCoder. We largely follow the architecture of the EC^2 system [14], which DreamCoder's structure is based on. In this section, such structure is shared unless otherwise noted.

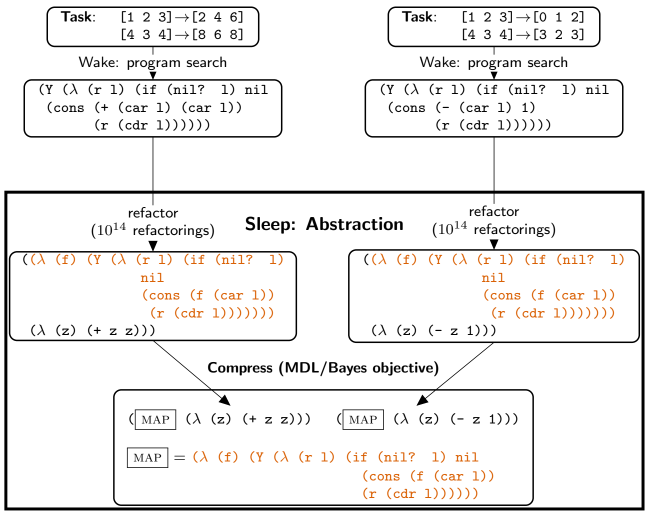
DreamCoder inputs a *training corpus* of synthesis problems and a *base language*. The base language should be low-level yet expressive enough to solve all the training problems, at least in principle. The corpus should contain problems with varying degrees of difficulty, ranging from problems that can be solved with very small programs in the base language to problems that require more complex programs. The programs in the corpus should also make up a consistent domain, meaning that the same abstractions should be useful in solving problems across the difficulty spectrum. A corpus with easy list manipulations but hard graph manipulations would not be suitable because the system will not be able to use the abstractions it learns from solving the easy problems in order to solve the hard problems.

The output of DreamCoder consists of a library of functions built from the base language or from other functions in the library, together with a neural search policy that can be used to efficiently solve similar synthesis tasks in the future. For example, in Fig. 1B, DreamCoder synthesized a function to sort sequences of numbers by invoking a learned library component — take the n^{th} largest element — and this component in turn calls lower-level learned concepts: maximum, and filter. In principle equivalent programs could be written in the base language, but those produced by the final learned language are more interpretable and much shorter. Expressed only in the initial primitives, these programs would be so complex as to be effectively out of the system's reach: they would never be found during a reasonably bounded search.

2.1 The DreamCoder Algorithm

DreamCoder is inspired by the original wake-sleep algorithm of Hinton, Dayan and colleagues [23] (hence the name DreamCoder). Traditional wake-sleep learning is an unsupervised method for training latent variable generative models. It iterates between wake and sleep phases: waking trains the generative model by proposing latent variables via an auxiliary neural net called a *recognition network*, while sleeping trains this recognition network on samples from the generative model. Porting this idea to program synthesis, our generative model is a probability distribution over programs (the latent variables), while the recognition network learns to map from specifications to programs.

Concretely, the waking phase of DreamCoder uses the current DSL (generative model) and neural search policy (recognition network) to solve as many problems from the corpus as it can. Unlike a traditional wake-sleep algorithm, however, our algorithm uses two distinct sleep phases. The first sleep phase, **abstraction sleep**, grows the DSL by consolidating new code abstractions from programs synthesized during waking (Fig. 3 left). The second sleep phase, **dream sleep**, improves the system's synthesis skills by training the neural network to help search for programs. The network



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

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

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

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

(MAP (λ (z) (+ z z)))   (MAP (λ (z) (- z 1)))

MAP = (λ (f) (Y (λ (r l) (if (nil?  l) nil
                   (cons (f (car l))
                   (r (cdr l))))))
```

Figure 2. Programs synthesized during waking are refactored during abstraction sleep to expose new candidate primitives. Two refactorings shown, with a common subexpression highlighted in orange. This particular refactoring hinges on new methods introduced in Section 2.2

.

trains on replayed experiences as well as 'dreams', or random programs built from the learned library (Fig. 3 right).

This 3-phase inference procedure works through two distinct kinds of bootstrapping. During each sleep cycle the next library bootstraps off the functions learned during earlier cycles. Simultaneously the generative and recognition models bootstrap each other: A more finely tuned library yields richer dreams for the recognition model to learn from, while a more accurate recognition model solves more tasks during waking which then feed into the next library. Both sleep phases also serve to mitigate the combinatorial explosion of search. Higher-level library routines allow tasks to be solved with fewer function calls, effectively reducing the *depth* of search. The neural recognition model downweights unlikely trajectories through the search space of all programs, effectively reducing the *breadth* of search.

2.2 Abstraction Sleep: Growing the Library

The key novelty in this paper is the abstraction sleep phase which discovers common components that can be used to more succinctly represent the programs discovered during the waking phase. The challenge is illustrated in Fig. 2. In this example, two different recursive solutions to two different tasks (expressed using the Y combinator) appear to have very few meaningful common subexpressions. However, DreamCoder uses a new data structure based on equivalence graphs [11] and constructed via dynamic programming which can summarize a large number of possible refactorings to the original programs. One of those refactorings exposes a common sub-expression that corresponds to the function

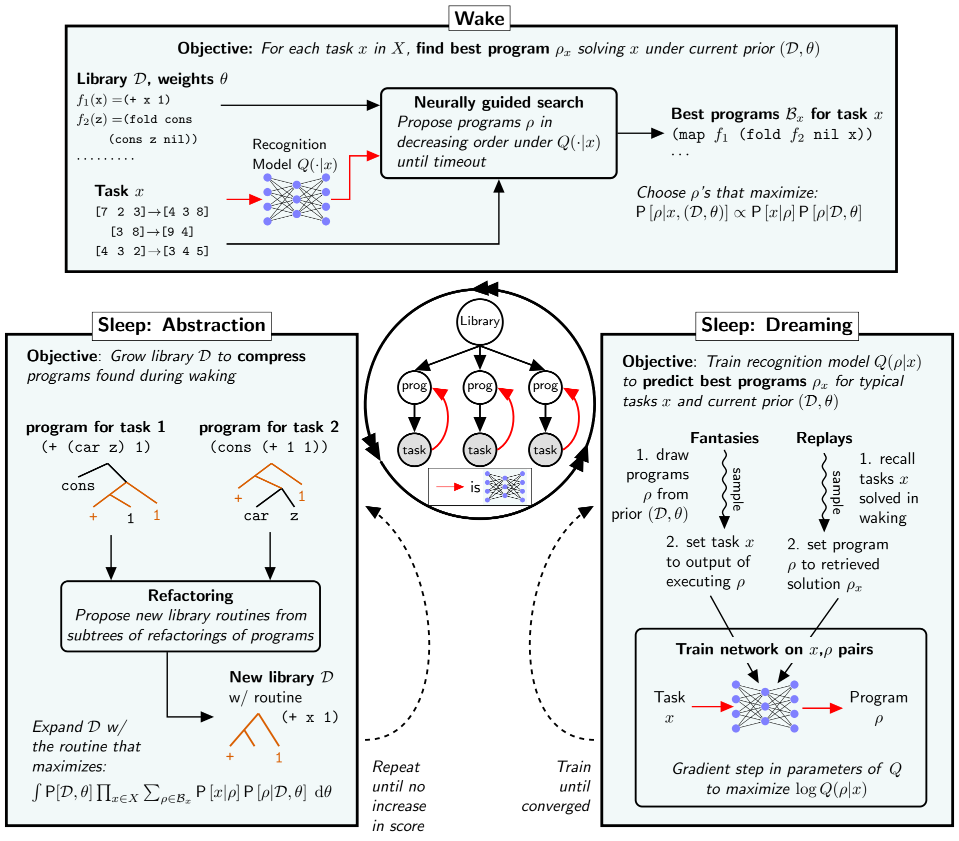

Wake
Objective: For each task x in X, find best program ρ_x solving x under current prior $(\mathcal{D},\theta)$
Library $\mathcal{D}$, weights θ
f_1(x) =(+ x 1)
f_2(z) =(fold cons (cons z nil))
Recognition Model $Q(\cdot|x)$
Neurally guided search
Propose programs ρ in decreasing order under $Q(\cdot|x)$ until timeout
Best programs $\mathcal{B}_x$ for task x
(map f_1 (fold f_2 nil x))
Task x
[7 2 3]→[4 3 8]
[3 8]→[9 4]
[4 3 2]→[3 4 5]
Choose ρ's that maximize:
$\mathrm{P}[\rho|x,(\mathcal{D},\theta)] \propto \mathrm{P}[x|\rho]\,\mathrm{P}[\rho|\mathcal{D},\theta]$
Sleep: Abstraction
Objective: Grow library $\mathcal{D}$ to compress programs found during waking
program for task 1
(+ (car z) 1)
program for task 2
(cons (+ 1 1))
cons
car
z
Refactoring
Propose new library routines from subtrees of refactorings of programs
New library $\mathcal{D}$ w/ routine (+ x 1)
Expand $\mathcal{D}$ w/ the routine that maximizes:
$\int \mathrm{P}[\mathcal{D},\theta] \prod_{x\in X} \sum_{\rho\in\mathcal{B}_x} \mathrm{P}[x|\rho]\,\mathrm{P}[\rho|\mathcal{D},\theta]\; d\theta$
Library
prog
task
is
Repeat until no increase in score
Train until converged
Sleep: Dreaming
Objective: Train recognition model $Q(\rho|x)$ to predict best programs ρ_x for typical tasks x and current prior $(\mathcal{D},\theta)$
Fantasies
1. draw programs ρ from prior $(\mathcal{D},\theta)$
sample
2. set task x to output of executing ρ
Replays
1. recall tasks x solved in waking
2. set program ρ to retrieved solution ρ_x
Train network on x,ρ pairs
Task x
Program ρ
Gradient step in parameters of Q to maximize $\log Q(\rho|x)$

Figure 3. DreamCoder performs approximate Bayesian inference for the graphical model in the **middle**: inputting synthesis tasks, which it explains with latent programs, and infers a latent library capturing cross-program regularities. A neural net, called the *recognition model* (red arrows) is trained to infer programs with high posterior probability. Waking (**top**) infers programs while holding the library and recognition model fixed. Abstraction (**left**) updates the library while holding the programs fixed by refactoring programs found during waking and abstracting out common components (highlighted in orange). Dreaming (**right**) trains the recognition model on 'Fantasies' (programs sampled from library) & 'Replays' (programs found during waking).

map, which can then be used to represent both functions concisely. The use of this data structure results in substantial efficiency gains: A graph with 10^6 nodes, calculated in minutes, can represent the 10^{14} refactorings in Fig. 2 that would otherwise take centuries to explicitly enumerate and search.

2.3 Dream Sleep: Training a Recognition Model

During dreaming, the system trains its recognition model, which later guides the search for programs during waking. Like EC^2, DreamCoder trains a recognition network on (program, task) pairs drawn from two sources of self-supervised data: *replays* of programs discovered during waking, and *fantasies*, or random programs assembled from members of the learned library. Replays ensure that the recognition model is trained on the actual tasks it needs to solve, and does not forget how to solve them, while fantasies provide a large and highly varied dataset to learn from, and are critical for data efficiency: becoming a domain expert is not a few-shot learning problem, but nor is it a big data problem. The input corpus to DreamCoder typically contains 100-200 tasks, which is too few examples for a high-capacity neural network. But after the model learns a library customized to the domain, it can draw unlimited samples or 'dreams' to train the recognition network, facilitating sample-efficient learning. Unlike EC^2 and other prior works, we parameterize and train our network in new ways which teach it to restrict the search over program space via learning to break syntactic symmetries.

2.4 A Bayesian View

In this section, we summarize the original formalization of Ellis et al. [14] of the wake sleep approach as a probabilistic inference problem in the context of DreamCoder. The input to the algorithm is a set of tasks, written X, and the output is both a set of programs $\{\rho_x\}$ solving each task $x \in X$, as well as a prior distribution over programs likely to solve tasks in the domain (Fig. 3 middle). This prior is encoded by a library, written $\mathcal{D}$, which when equipped with a real-valued weight vector, written θ, defines a distribution over programs, written $\mathrm{P}[\rho|\mathcal{D},\theta]$. The library $\mathcal{D}$ is a set of typed λ-calculus expressions, while the weight vector θ acts similarly to the production probabilities of a probabilistic context free grammar, with each single element of θ controlling the probability of using a single element of $\mathcal{D}$. Appendix 6 formally specifies $\mathrm{P}[\rho|\mathcal{D},\theta]$ via a probabilistic program which stochastically generates well-typed expressions obeying variable scoping rules. Using this notation, the joint distribution, J, over the observed tasks and the latent variables is

$$J(\mathcal{D},\theta) \triangleq \mathrm{P}[\mathcal{D},\theta] \prod_{x\in X} \sum_{\rho} \mathrm{P}[x|\rho]\mathrm{P}[\rho|\mathcal{D},\theta] \qquad (1)$$

where $\mathrm{P}[\mathcal{D},\theta]$ is a prior distribution over languages and parameters and $\mathrm{P}[x|\rho]$ scores the likelihood of a task $x \in$

X given a program ρ.[1] The solution to the DreamCoder problem is then the optimal language and weight vector

$$\mathcal{D}^* = \arg\max_{\mathcal{D}} \int J(\mathcal{D}, \theta)\ \mathrm{d}\theta \qquad \theta^* = \arg\max_{\theta} J(\mathcal{D}^*, \theta) \tag{2}$$

Evaluating Eq. 1 entails marginalizing over the infinite set of all programs – which is impossible. We make a particle-based approximation to Eq. 1 and instead marginalize over a finite **beam** of programs, with one beam per task, collectively written $\{\mathcal{B}_x\}_{x\in X}$. This particle-based approximation is written $\mathscr{L}(\mathcal{D}, \theta, \{\mathcal{B}_x\})$ and acts as a lower bound on the joint density: $J(\mathcal{D}, \theta) \geq \mathscr{L}$, where $\mathscr{L}$ is

$$\mathscr{L}(\mathcal{D}, \theta, \{\mathcal{B}_x\}) \triangleq \mathrm{P}[\mathcal{D}, \theta] \prod_{x\in X} \sum_{\rho\in\mathcal{B}_x} \mathrm{P}[x|\rho]\mathrm{P}[\rho|\mathcal{D}, \theta] \tag{3}$$

In all of our experiments we set the maximum beam size $|\mathcal{B}_x|$ to 5. Wake and sleep phases correspond to alternate maximization of $\mathscr{L}$ w.r.t. $\{\mathcal{B}_x\}_{x\in X}$, $\mathcal{D}$, and θ.

Wake: Maxing $\mathscr{L}$ w.r.t. the beams. Here $(\mathcal{D}, \theta)$ is fixed and we want to find new programs to add to the beams so that $\mathscr{L}$ increases the most (and therefore best approximates $J(\mathcal{D}, \theta)$). $\mathscr{L}$ most increases by finding programs where $\mathrm{P}[x|\rho]\mathrm{P}[\rho|\mathcal{D}, \theta] \propto \mathrm{P}[\rho|x, \mathcal{D}, \theta]$ is large, i.e., programs with high posterior probability, which we use as the search objective during waking. Unlike EC^2, during each wake cycle we sample a random minibatch of tasks to try to solve, rather than trying to solve every task at each wake.

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

Finding programs solving tasks is difficult because of the infinitely large, combinatorial search landscape. We ease this difficulty by training a neural recognition model, $Q(\rho|x)$, during the **Dreaming** phase: Q is trained to assign high probability to programs which score highly under the posterior $\mathrm{P}[\rho|x, (\mathcal{D}, \theta)] \propto \mathrm{P}[x|\rho]\mathrm{P}[\rho|(\mathcal{D}, \theta)]$. Thus training the neural network amortizes the cost of finding programs with high posterior probability.

Sleep (Dreaming): tractably maxing $\mathscr{L}$ w.r.t. the beams. Here we train $Q(p|x)$ to assign high probability to programs p where $\mathrm{P}[x|\rho]\mathrm{P}[\rho|\mathcal{D}, \theta]$ is large, because incorporating those programs into the beams will most increase $\mathscr{L}$. In the sections that follow, we elaborate on how these phases are realized.

3 Abstraction Sleep

During the abstraction phase of sleep, the model grows its library with the goal of discovering specialized library routines that allow it to easily express solutions to the tasks at hand. Ease of expression translates into a preference for libraries that best compress programs found during waking, namely the programs in the beams $\{\mathcal{B}_x\}_{x\in X}$.

In order to do this, however, we need to address a problem with the beam approximation in Eq. 3: the beam contains programs found in the previous waking phase, but those programs will not be written in terms of the new routines that we may want to consider in $\mathcal{D}$, so there will be a missalignment between the programs in $\mathcal{B}_x$, and the proposed $\mathcal{D}$.

There is also a second challenge: integration over θ (Eq. 2) is intractable when θ is high dimensional. Hence we approximate the abstraction sleep objective using the Akaike Information Criterion (AIC: [3]).[2] The AIC is a model selection criterion which replaces integration with maximization, and penalizes the number of continuous degrees of freedom (here, the dimensionality of θ).

We address both of these challenges by refining the optimization objective to the formula below.

$$\log \mathrm{P}[\mathcal{D}] + \arg\max_{\theta} \sum_{x\in X} \log \sum_{\rho\in\mathcal{B}_x} \mathrm{P}[x|\rho] \max_{\rho' \longrightarrow^*_\beta \rho} \mathrm{P}[\rho'|\mathcal{D}, \theta] + \log \mathrm{P}[\theta|\mathcal{D}] - |\theta|_0 \tag{4}$$

We define $\mathrm{P}[\mathcal{D}] \propto \exp\left(-\lambda \sum_{\rho\in\mathcal{D}} \mathrm{size}(\rho)\right)$ using a hyperparameter λ so that it minimizes the size of each library function in $\mathcal{D}$. We place a symmetric Dirichlet prior over the weight vector θ to define $\mathrm{P}[\theta|\mathcal{D}]$. But the most important aspect of this objective is that instead of considering only the programs in the beam, it also considers any of their refactorings. By refactoring, we mean any program which is equivalent up to β-reduction (i.e., function application/variable substitution [39]). We write $\rho \longrightarrow_\beta \rho'$ to mean that ρ rewrites to ρ' in one step of β-reduction, and write $\rho \longrightarrow^*_\beta \rho'$ for the transitive reflexive closure of $\rho \longrightarrow_\beta \rho'$.

Because infinitely many programs may β-reduce to a given expression, DreamCoder bounds the number of β-reduction steps separating a program from its refactorings. Yet this now-bounded set of refactorings grows exponentially, motivating the techniques described next.

3.1 Efficient Refactoring

We tame the combinatorial explosion associated with refactoring using machinery that draws primarily on the ideas of an *E-graph* and a *version space*. A version space is a data structure that compactly represents a large set of programs and supports efficient set operations like union, intersection, and membership checking. An E-graph, in turn, is a data-structure that leverages sharing to compactly represent many alternative versions of the same program. The two ideas are quite closely related, and in fact, prior systems

[1]For example, for list processing, the likelihood is 1 iff the program predicts the correct outputs on the observed inputs. For probabilistic programs, the likelihood is the probability of the program sampling the observation.

[2]The AIC is only one such possible model selection criterion. Other choices, such as Variational Bayes [26], may more closely approximate the marginal over θ, but are more complex and come at greater computational expense.

such as FlashFill have used E-graph-like representations to represent version spaces compactly [20].

Version space. The purpose of a version space is to compactly represent a set of programs. Our basic language will be λ-calculus with some additional primitives. We use deBuijn notation to simplify the treatment of bound variables.

Definition 3.1. A **version space** is either

- A deBuijn index: written $\$i$, for i a natural number
- A primitive, such as the number `42`, or the function `map`
- 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

By a **leaf** we mean either a primitive or a deBuijn index.

The union operator ($\uplus$), which, intuitively, represents a nondeterministic choice between a collection of alternatives, allows the notation above to compactly represent large sets of programs. For example, the version space $(\lambda\uplus\{\$0, \texttt{7}\})(\uplus\{\texttt{4}, \texttt{9}\})$ encodes four different expressions: $(\lambda\$0)\texttt{4}$, $(\lambda\$0)\texttt{9}$, $(\lambda\texttt{7})\texttt{4}$, and $(\lambda\texttt{7})\texttt{9}$. We refer to the set of expressions encoded by a version space as its **extension**:

Definition 3.2. The **extension** of a version space v is written $[\![v]\!]$ and is defined recursively as:

$$[\![\rho]\!] = \{\rho\}\text{, if } \rho \text{ a leaf} \qquad [\![\Lambda]\!] = \Lambda \qquad [\![\varnothing]\!] = \varnothing$$

$$[\![\lambda v]\!] = \{\lambda e : e \in [\![v]\!]\} \qquad [\![\uplus V]\!] = \{e : v \in V,\ e \in [\![v]\!]\}$$

$$[\![(v_1\ v_2)]\!] = \{(e_1\ e_2) : e_1 \in [\![v_1]\!],\ e_2 \in [\![v_2]\!]\}$$

The data structures we use to represent version spaces also support efficient membership checking, which we write as $e \in [\![v]\!]$. Important for our purposes, it is also efficient to extract the smallest member of a version space's extension in terms of a new library–i.e., extracting the most *compressive* member of a version space given a new library. We define $\textsc{extract}(v|\mathcal{D})$ as calculating $\arg\min_{\rho\in[\![v]\!]} \text{size}(\rho|\mathcal{D})$, where $\text{size}(\rho|\mathcal{D})$ for program ρ and library $\mathcal{D}$ is the size of the syntax tree of ρ, when members of $\mathcal{D}$ are counted as having size 1 (Fig. 5A).

Inverse β-reduction. Our next goal is to define an operator over version spaces which calculates the set of n-step refactorings of a program ρ. We call this operator $I\beta_n$, because it inverts β-reduction n times, writing $I\beta_n(\rho)$ for a version space encoding n-step refactorings of program ρ. This operator should satisfy:

$$[\![I\beta_n(\rho)]\!] = \{\rho'\ :\ \underbrace{\rho' \longrightarrow_\beta \rho'' \longrightarrow_\beta \cdots \longrightarrow_\beta \rho}_{\leq\, n \text{ times}}\} \qquad (5)$$

We define $I\beta_n$ in terms of another operator, $I\beta'$, which performs a single step of refactoring (Fig. 5B); in particular, we want to define $I\beta'$ so that its extension obeys

$$[\![I\beta'(v)]\!] = \{\rho'\ :\ \rho' \longrightarrow_\beta \rho \quad \forall \rho \in [\![v]\!]\} \qquad (6)$$

Here we will define and explain $I\beta'$ intuitively but the appendix proves that these definitions are *consistent* (Theorem G.5) and *complete* (Theorem G.6). *Consistency* means that only valid refactorings are output (every program in $[\![I\beta'(v)]\!]$ does actually β-reduced to a program in $[\![v]\!]$), while *completeness* means that every valid refactoring is output (every program which β-reduces to an expression in $[\![v]\!]$ is present in $[\![I\beta'(v)]\!]$). Consistency and completeness imply Eq. 6.

Constructing $I\beta'$ depends on careful consideration of the β-reduction operator we aim to invert (see [39] and Appendix F). The operator $I\beta'$ builds both top-level redexes and also recurses to build redexes within the body of an expression. Fig. 5C defines $I\beta'$ as the union of an operator S, which constructs these top-level substitutions, along with recursive invocations of $I\beta'$. The S operator (Fig. 5D) takes as input a version space u and then returns a version space whose extension contains programs of the form $(\lambda b)v$ where substituting the value v into the body b (i.e. performing β reduction) gives an expression in $[\![u]\!]$. The definition of S is a bit involved both because of the different ways in which one can introduce additional abstraction into a lambda expression as well as because of the need to keep track of deBuijn indices. Index tracking is done by parameterizing S by an index k, so $S(v)$ is defined as $S_0(v)$, but every time S_k is applied to an expression of the form λb, the body b must be handled by S_{k+1} to keep track of the fact that the body is inside an additional λ.

We now define how these version spaces aggregate into a single data structure, one for each program, tracking every equivalence revealed by $I\beta_n$. Observe that every time we calculate $I\beta_n(\rho)$, we obtain a version space containing expressions semantically equivalent to program ρ—and also we obtain $I\beta_n$ for any subexpressions of ρ. In order to allow these subexpressions to be refactored independently, we track the equivalences exposed by $I\beta_n$ and compile all these equivalencies together. For example, when refactoring `(* (+ 1 1) (+ 5 5))`, the $I\beta_1$ operator, which inverts 1 step of β-reduction, will identify that `(+ 1 1)` can be rewritten as `(double 1)` while `(+ 5 5)` can be rewritten as `(double 5)`, where `double = (lambda (x) (+ x x))`. However, $I\beta_1$ will not discover that `(* (+ 1 1) (+ 5 5))` can be rewritten as `(* (double 1) (double 5))`, because this requires inverting 2 steps of β-reduction. Yet this rewrite is clearly licensed by the semantic equivalences exposed by $I\beta_1$.

Inspired by the E-graph equality saturation approach within program analysis [53], we build a graph tracking the equivalences exposed by $I\beta_n$, and finally return a single structure for each program compiling all of these equivalences. This allows *e.g.* refactoring `(* (+ 1 1) (+ 5 5))` into `(* (double 1) (double 5))` without considering more than 1 step of β-reduction. Concretely, for each program ρ, we calculate a

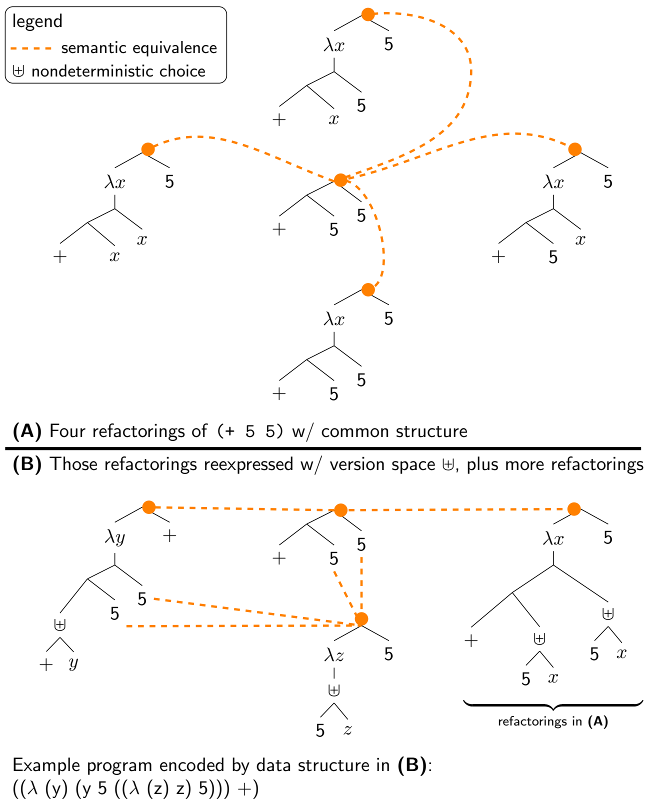

legend
semantic equivalence
⊎ nondeterministic choice
(A) Four refactorings of (+ 5 5) w/ common structure
(B) Those refactorings reexpressed w/ version space ⊎, plus more refactorings
refactorings in (A)
Example program encoded by data structure in (B):
((λ (y) (y 5 ((λ (z) z) 5))) +)

Figure 4. Inverting β-reduction by one step using the ⊎ version space operator and equivalence tracking. **(A)**: four refactorings of `(+ 5 5)`, each semantically equivalent to the original expression– notated with a dashed orange line–by abstracting out the number `5`. Because there are two `5`s, and each can be either left as `5` or replaced with a variable, we have four possibilities. The rightmost tree in **(B)** collapses all four such refactorings into a single tree via ⊎ and shows several other portions of the data structure, but ellides others for clarity.

version space $I\beta(\rho)$ defined as

$$I\beta(\rho) = I\beta_n(\rho) \uplus \begin{cases} I\beta(f)\ I\beta(x) & \text{if } \rho = (f\ x) \\ \lambda I\beta(b) & \text{if } \rho = \lambda b \\ \varnothing & \text{if } \rho \text{ is a leaf} \end{cases}$$

where n, the amount of refactoring, is a hyper parameter. We set n to 3 for all experiments unless otherwise noted. Figure 4 diagrams a subset of the refactoring data structure when $n = 1$ and the refactored expression is `(+ 5 5)`.

3.2 Putting Together the Pieces

Algorithm 2 (Appendix) specifies our library learning procedure, which is exponential in the amount of refactoring but polynomial in program size. Naively, the number of candidate new library routines grows linearly with the number of programs, and each candidate must be scored against all other programs, giving quadratic dependence on the number of programs. In practice we found this quadratic dependence prohibitive. So, for each version space v, we perform a beam search to approximately calculate the top 10^6 candidate libraries $\mathcal{D}$ minimizing the size of $\textsc{extract}(v|\mathcal{D})$.

4 Dream Sleep

Our dream phase works differently from a conventional wake-sleep [23] dream phase in the following way: we think of dreaming as creating an endless stream of random problems, which we then solve during sleep, and train the recognition network to predict the solution conditioned on the problem. The classic wake-sleep algorithm would instead sample a random program, execute it to get a task, and train the recognition network to predict the sampled program from the sampled task. Specifically, we train Q to perform MAP inference by maximizing $\mathbb{E}\left[\log Q\left(\left(\arg\max_\rho \mathrm{P}[\rho|x, L]\right)|x\right)\right]$, where the expectation is taken over tasks. Taking this expectation over the empirical distribution of tasks trains Q on replays; taking it over samples from the generative model trains Q on fantasies. To clarify this departure from classic wake/sleep, we define a pair of alternative objectives for the recognition model, $\mathcal{L}^{\text{post.}}$ and $\mathcal{L}^{\text{MAP}}$, which (respectively) train Q to perform full posterior inference (as in EC^2 [14] and classic wake/sleep), or MAP inference (as in DreamCoder):

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

$$\mathcal{L}^{\text{post.}}_{\text{Replay}} = \mathbb{E}_{x\sim X}\left[\sum_{\rho\in\mathcal{B}_x} \frac{\mathrm{P}\left[x,\rho|\mathcal{D},\theta\right]}{\sum_{\rho'\in\mathcal{B}_x}\mathrm{P}\left[x,\rho'|\mathcal{D},\theta\right]}\log Q(\rho|x)\right]$$

$$\mathcal{L}^{\text{MAP}}_{\text{Replay}} = \mathbb{E}_{x\sim X}\left[\log Q\left(\arg\max_{\rho\in\mathcal{B}_x}\mathrm{P}[\rho|x,\mathcal{D},\theta]\ \middle|\ x\right)\right]$$

$$\mathcal{L}^{\text{post.}}_{\text{Fantasy}} = \mathbb{E}_{(\rho,x)\sim(\mathcal{D},\theta)}\left[\log Q(\rho|x)\right]$$

$$\mathcal{L}^{\text{MAP}}_{\text{Fantasy}} = \mathbb{E}_{x\sim(\mathcal{D},\theta)}\left[\log Q\left(\arg\max_{\rho}\mathrm{P}[\rho|x,\mathcal{D},\theta]\ \middle|\ x\right)\right]$$

Evaluating $\mathcal{L}_{\text{Fantasy}}$ involves drawing programs from the current library, 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. We simply sample the inputs from the empirical observed distribution of inputs in X. The $\mathcal{L}^{\text{MAP}}_{\text{Fantasy}}$ objective also involves finding the MAP program solving a task drawn from the library. To make this tractable, rather than *sample* programs as training data for $\mathcal{L}^{\text{MAP}}_{\text{Fantasy}}$, we *enumerate* programs in decreasing order of their prior probability, tracking, for each dreamed task x, the set of enumerated programs maximizing $\mathrm{P}[x,\rho|\mathcal{D},\theta]$ (Appendix Algorithm 3).

We chose to have DreamCoder maximize $\mathcal{L}^{\text{MAP}}$ rather than $\mathcal{L}^{\text{post.}}$ because, when, combined with our parameterization of Q, described next, we will show that $\mathcal{L}^{\text{MAP}}$ encourages the recognition model to break syntactic symmetries.

(A) extract smallest program

$$\textsc{extract}(v|\mathcal{D}) = \begin{cases} e, \text{ if } e \in \mathcal{D} \text{ and } e \in [\![v]\!] \\ \textsc{extract}'(v|\mathcal{D}), \text{ otherwise.} \end{cases}$$

$$\textsc{extract}'(e|\mathcal{D}) = e, \text{ if } e \text{ is a leaf}$$

$$\textsc{extract}'(\lambda b|\mathcal{D}) = \lambda\textsc{extract}(b|\mathcal{D})$$

$$\textsc{extract}'(f\ x|\mathcal{D}) = \textsc{extract}(f|\mathcal{D})\ \textsc{extract}(x|\mathcal{D})$$

$$\textsc{extract}'(\uplus V|\mathcal{D}) = \underset{e \in \{\textsc{extract}(v|\mathcal{D})\ :\ v \in V\}}{\arg\min} \operatorname{size}(e|\mathcal{D})$$

(B) invert n steps of β-reduction

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

(C) invert 1 step of β-reduction

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

(D) build top-level redex (i.e. a substitution)

$$S(v) = S_0(v) \qquad S_k(v) = \left\{(\lambda\$k)(\downarrow_0^k v)\right\} \cup S'_k(v)$$

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

(E) downshift utility used by **(D)**

$$\downarrow_c^k \$i = \begin{cases} \$i, \text{ if } i < c \\ \$(i-k), \text{ if } i \ge c+k \\ \varnothing, \text{ if } c \le i < c+k \end{cases} \qquad \downarrow_c^k v = v, \text{ if } v \text{ is a primitive or } \varnothing \text{ or } \Lambda$$

$$\downarrow_c^k \lambda b = \lambda \downarrow_{c+1}^k b \qquad \downarrow_c^k (f\ x) = (\downarrow_c^k f\ \downarrow_c^k x) \qquad \downarrow_c^k \uplus V = \uplus \left\{\downarrow_c^k v \mid v \in V\right\}$$

Figure 5. Definitions of core refactoring operations. We write these recursively, but actually implement them using a dynamic program: we hash cons each version space, and only calculate the operators $I\beta_n$, $I\beta'$, and S_k once per each version space.

Parameterizing Q. Mathematically, the neural recognition model Q assigns a probability to every possible program. Mechanically, how should we parameterize this probability distribution? Because Q acts as a (task-conditioned) probabilistic language model over program syntax trees, this is equivalent to asking what kind of language model Q's output should embody.

The simplest such parameterization is a *unigram* distribution over library components. This approach–developed in [34] and employed by EC^2 –predicts the probability of each library component independently of what other components are present in the program. Unigram probabilities are fast to compute and fast to query, so at test time, the synthesizer is not bottlenecked by neural network calculations. Recall though that effective domain-specific synthesizers not only expose high-level building blocks, but also carefully restrict the ways in which those building blocks are allowed to compose: *e.g.* disallowing adding zero, or forcing right-associative addition. Unigrams render these symmetry-breaking restrictions impossible, because the parameterization cannot exclude certain primitives based on their local syntactic context (*e.g.* disallowing zero as the child of addition).

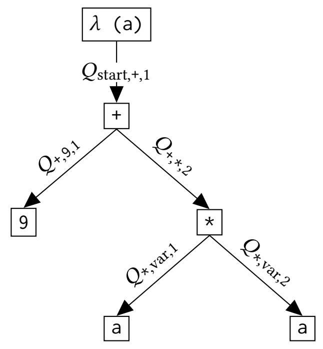
At the other end of the spectrum are *autoregressive neural language models* (*e.g.* recurrent networks [13] and transformers [48]). By conditioning on the entirety of a partially generated program, these models can break arbitrary syntactic symmetries–but at the cost of heavyweight neural network computations at test time. Thus they are incompatible with rapid enumerative search.

A *bigram* parameterization strikes a middle ground which both breaks symmetries and is synthesizer friendly. Just as the familiar bigram distribution over *sequences* conditions only on the immediately preceding token, our bigram distribution over *syntax trees* conditions on the immediate ancestor in the syntax tree. This can break many symmetries: if the immediate ancestor is multiplication, a bigram can condition on this fact and assign zero probability to its child being zero, and so rule out multiplication by zero. Yet the neural net need only run once per task to get a bigram transition matrix, so bigram-guided enumerative search is not slowed down by the neural network.

Note though that n-gram models cannot break *all* symmetries: *e.g.* breaking commutativity requires ordering arguments lexicographicly.

Our bigram model conditions not just on the immediate ancestor, but also conditions on which child is being generated as its argument. This allows more symmetry breaking: *e.g.* prohibiting one as the second child of division (don't divide by one), but allowing it as the first child (allowed to divide one by something). Thus, Q outputs a bigram transition matrix that is actually a 3-index tensor: indices for the parent and child library components, and an index for which argument of the parent is being generated. Fig. 6 (top) diagrams how this 3-index tensor–written Q_{ijk}–stochastically generates a syntax tree. Here i indexes possible children ($i \in \mathcal{D} \cup \{\text{var}\}$, with 'var' for variables), j indexes possible parents ($j \in \mathcal{D} \cup \{\text{start}, \text{var}\}$, with 'start' for no parent), and k indexes which argument of the parent is being generated ($1 \le k \le A$, with A the arity of the parent).

Together, this bigram parameterization interacts with $\mathcal{L}^{\text{MAP}}$ to learn to break symmetries. This interaction occurs because the bigram parameterization can disallow library routines depending on their local syntactic context, while the $\mathcal{L}^{\text{MAP}}$



	Unigram	Bigram

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

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

Figure 6. Top: Syntax tree annotated with bigram probabilities from recognition network. **Bottom:** Breaking symmetries requires both bigrams and $\mathcal{L}^{\text{MAP}}$ objective. Model learns to associate addition to the right and avoiding adding zero. % right-associative over 500 samples. $\mathcal{L}^{\text{MAP}}$/Unigram incorrectly learns to never generate programs with `0`'s, while $\mathcal{L}^{\text{MAP}}$/Bigram correctly learns that `0` should only be disallowed as an argument of addition.

objective forces all probability mass onto a single member of a set of syntactically distinct but semantically equivalent expressions. We experimentally confirm this symmetry-breaking behavior by training recognition models to minimize either $\mathcal{L}^{\text{MAP}}/\mathcal{L}^{\text{post.}}$ and to use either a bigram/unigram parameterization. Figure 6 (bottom) shows the result of training Q in these four regimes and then sampling programs. On this particular run, the combination of bigrams and $\mathcal{L}^{\text{MAP}}$ learns to avoid adding zero and associate addition to the right — different random initializations lead to either right or left association. Appendix H proves that any global optimizer of $\mathcal{L}^{\text{MAP}}$ breaks symmetries, while Appendix I gives our neural network training details and architectures.

5 Evaluation

We study DreamCoder across synthesis domains with the goal of answering three questions: (1) whether combining abstraction learning and recognition model training yields faster solving of more held-out problems than either alone; (2) whether recognition model training influences the structure of the learned libraries; and (3) whether refactoring proves necessary for generalization to new synthesis problems. To answer these questions we evaluate across the following six domains:

- List processing: functional programming problems taken from [14], specified via input/outputs, split 50/50 test/train. We further increased test-set difficulty by excluding 31 test problems which were solvable within 10 minutes via enumerative search (making our list processing results not directly comparable with [14]: our list results would be improved on the original data). The system is initialized with functional programming primitives (`map`, `fold`, `cons`, `car`, `cdr`, `if`, `length`, `index`, `=`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`) and numerical routines (`mod`, `*`, `>`, `is-square`, and `is-prime`).
- Text editing: in the style of FlashFill [20], testing on 108 2017 SyGuS problems [1] and training on text editing problems from [14]. This system is initialized with the same functional programming primitives as above but also including character and string constants.
- LOGO graphics: inverse graphics where each synthesis problem is specified by a raster image, and solved by synthesizing a program in LOGO Turtle [54] graphics. These programs control a simulated 'pen' as it moves over a canvas. The base language includes 'for' loops, a stack for saving/restoring the pen state, and arithmetic on angles and distances.
- Blocks towers: a planning domain where each synthesis problem is a tower built on a simulated 'stage' and the system must write a program planning how to control a simulated 'hand' to build the target tower. The base language includes the same control flow as in LOGO graphics.
- Generative regular expressions: a probabilistic programming domain where the system infers a probabilistic generative model–encoded as a regex–from only positive examples of strings. Data taken from [22], which crawled the web for CSV files.
- Symbolic regression: synthesizing programs with real-valued parameters. The system is given input/outputs of polynomials and rational functions, and tasked with writing a program fitting those inputs/outputs, following [14]. The recognition model additionally observes an image of the function's graph, processed via a convolutional neural network. We fit continuous parameters of the symbolic function via an inner loop of gradient descent.

We compared our full system on held out test problems with ablations missing either the neural recognition model (the "dreaming" sleep phase) or ability to form new library routines (the "abstraction" sleep phase). To isolate the role of refactoring, we construct two *Memorize* baselines. These baselines incorporate every task solution wholesale into the library, thereby memorizing the best solutions found during waking (cf. [8]). We evaluate memorize variants both with and without recognition models. To further probe refactoring, we compare with *Exploration-Compression* (EC [10]) and *EC*^{2} [14]. We modify EC^2 to randomly minibatch tasks, as DreamCoder does, and reimplement EC by running abstraction sleep without any refactoring while disabling our neural recognition model. We compare with two other baselines: *Neural Program Synthesis*, which trains RobustFill [13] on samples from the initial library; and *Enumeration*, which performs type-directed enumeration [17] for 24 hours per task, generating up to 400 million programs for each task.

In every domain, our model always solves at least as many problems as the best alternative approach on that domain (Fig. 7A-B). It also generally solves these problems in the least time (mean 54.1sec; median 15.0sec; Appendix Fig. 20). These results establish that each of DreamCoder's core components – library learning in the sleep-abstraction phase, and recognition model training in the sleep-dreaming phase – contributes significantly to its overall performance. The synergy between these components is especially clear in the more creative, generative structure building domains, LOGO graphics and tower building, where neither ablation ever solves more than 60% of held-out tasks while DreamCoder learns to solve nearly 100% of them (Fig. 7A). The time needed to train DreamCoder to convergence varies across domains, but typically takes around a day using moderate compute resources (20-100 CPUs; Appendix J).

Compared to EC^2 on its own domains–list, text and symbolic regression–DreamCoder converges with far less compute, owing to its random minibatching of tasks during waking. DreamCoder converges after 10 wake/sleep cycles for list and text, with a synthesis timeout of 12 min (on 64 CPUs), while EC^2's nonbatched approach requires 3 wake/sleep cycles with a timeout of 120 minutes (on 128 CPUs–see [14]): thus batching buys a speedup of 6×. A similar calculation shows a 15× speedup on symbolic regression.

Thus, to make the comparison more fair yet also more challenging, we evaluate against EC^2 modified to use the same random batching scheme that we found to accelerate convergence with DreamCoder. This also better illustrates the consequences of our core improvements over EC^2, namely refactoring and symmetry breaking during abstraction and dream sleep, respectively. However, as a consequence our results are no longer directly comparable to the results presented in [14]: doubly so for list processing, where we made the test set harder. Fig. 7B compares against this new, randomly minibatched version of EC^2. Unsurprisingly our new refactoring algorithm helps the most for functional programming (list processing), but we also get modest gains for construction tasks (LOGO/towers).

Examining how the learned libraries grow over time, both with and without learned recognition models, reveals functionally significant differences in their depths and sizes. Across domains, deeper libraries correlate well with solving more tasks ($r = 0.79$), and a recognition model leads to better performance at all depths. The recognition model also leads to deeper libraries by the end of learning, with correspondingly higher asymptotic performance levels (Fig. 7C-D). Similar but weaker relationships hold between the size of the learned library and performance. Thus the recognition model appears to bootstrap "better" libraries, where "better" correlates with both depth and breadth.

5.1 Qualitative Results

We qualitatively discuss DreamCoders behavior, focusing here on the more creative, generative problems: synthesizing images, plans, and text. In inspecting DreamCoders learned LOGO graphics library, we find parametric drawing routines corresponding to the families of visual objects in its training data, like polygons, circles, and spirals (Fig. 8B). We also find more surprising routines, such as those Fig. 8B dubs 's-curve' and 'arc', which may be less interpretable but the system nonetheless finds useful for many drawing tasks. It additionally learns more abstract visual relationships, like radial symmetry, which it models by abstracting out a new higher-order function into its library (Fig. 8C).

Visualizing the system's dreams across its learning trajectory provides a window into how the generative model bootstraps recognition model training: As the library grows and becomes more finely tuned to the domain, the neural net receives richer and more varied training data. At the beginning of learning, random programs written using the library are simple and largely unstructured (Fig. 8D), offering limited value for training the recognition model. After learning, the system's dreams are richly structured (Fig. 8E), compositionally recombining latent building blocks and motifs acquired from the training data in ways never seen in its waking experience, but ideal for training a broadly generalizable recognition model [55].

Inspired by the classic AI 'copy demo' – where an agent looks at a tower made of toy blocks then re-creates it [58] – DreamCoder learns to solve tower 'copy tasks'. Analogously to LOGO graphics, we find inside its learned library parametric 'options' [52], or 'planning macros', for building blocks towers (Fig. 9B), including concepts like arches, staircases, and bridges, which we also see recombined in many novel ways in the model's dreams (Fig. 9C,D).

In LOGO graphics and towers, the system's learned library effectively embodies a probabilistic generative model over images and plans, respectively. For generative regexes, each *individual* program is a probabilistic generative model. The system here learns to learn regular expressions that describe the structure of typically occurring text concepts, such as phone numbers, dates, times, or monetary amounts (Fig. 10). It can explain many real-world text patterns and use its explanations as a probabilistic generative model to imagine new examples of these concepts. For instance, it can infer an

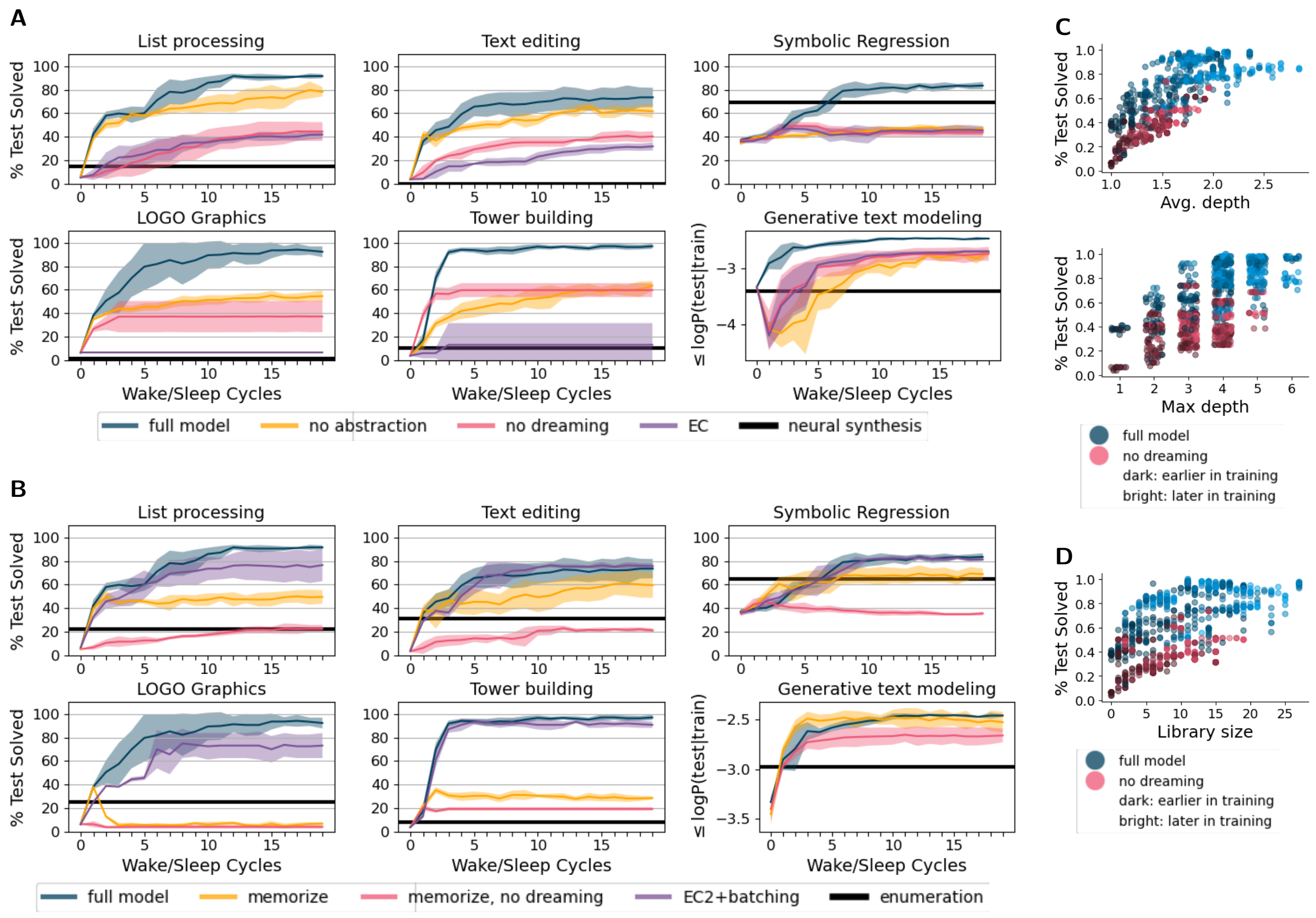


Figure 7. (A-B) Test set accuracy. Generative text modeling: posterior predictive likelihood of held-out strings on held out tasks, normalized per-character. Error bars: ±1 std. dev. over 5 runs with different random seeds (compute limits restricted us to 3 EC^2 runs, except 2 such runs for LOGO and 0 for generative text). **(C-D)** Evolution of library structure over wake/sleep cycles (darker: earlier cycles; brighter: later cycles). Each dot is a single wake/sleep cycle for a single run on a single domain. Deeper (**C**) and larger (**D**) libraries correlate with solving more tasks. The dreaming phase bootstraps these deeper, broader libraries, and also, for a fixed library structure, dreaming leads to higher performance.

abstract pattern behind the examples `$5.70`, `$2.80`, `$7.60`, . . . , to generate `$2.40` and `$3.30` as other examples of the same concept. Given patterns with exceptions, such as `-4.26`, `-1.69`, `-1.622`, . . . , `-1` it infers a probabilistic model that typically generates strings such as `-9.9` and occasionally generates strings such as `-2`. It can also learn more esoteric concepts, which humans may find unfamiliar but can still readily learn and generalize from a few examples: Given examples `-00:16:05.9`, `-00:19:52.9`, `-00:33:24.7`, . . . , it infers a generative concept that produces `-00:93:53.2`, as well as plausible near misses such as `-00:23=43.3`.

5.2 From Learning Libraries to Learning Languages

Our experiments up to now study how DreamCoder grows from a "beginner" state given basic domain-specific procedures, such that only the easiest problems have simple, short solutions, to an "expert" state with concepts allowing even the hardest problems to be solved with short, meaningful programs. Now we ask whether it is possible to learn from a more minimal starting state, without even basic domain knowledge: Can DreamCoder start with only highly generic programming and arithmetic primitives, and grow a domain-specific language with both basic and advanced domain concepts allowing it to solve all the problems in a domain?

Motivated by classic work on inferring physical laws from experimental data [28, 45, 49], we first task DreamCoder with learning equations describing 60 different physical laws and mathematical identities taken from AP and MCAT physics "cheat sheets", based on numerical examples of data obeying each equation (Appendix Tbl. 1). The full dataset includes many well-known laws in mechanics and electromagnetism, which are naturally expressed using concepts like

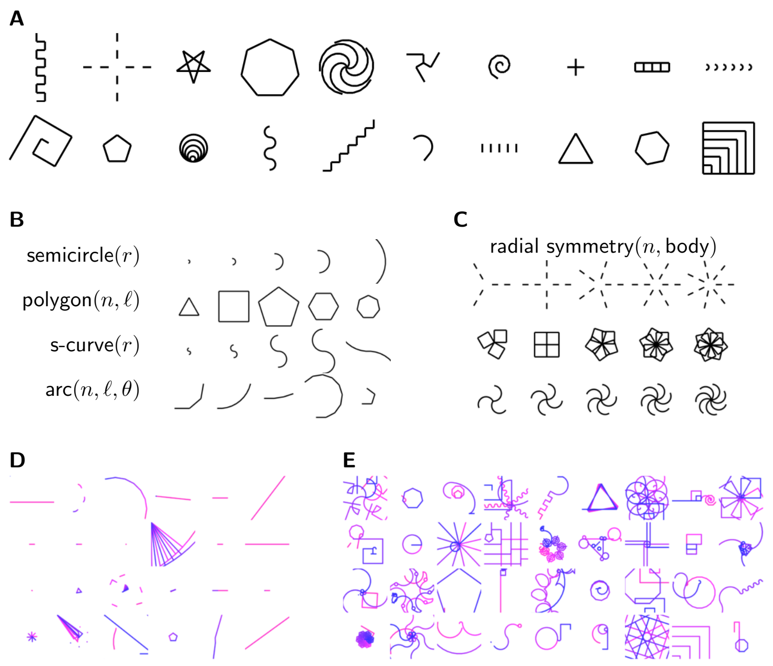


Figure 8. (A): 20 (out of 160) LOGO graphics tasks. **(B-C):** Example learned library routines for drawing families of curves (**B**) as well as primitives that take entire programs as input (**C**). Each image shows the same library code executed with different arguments. **(D-E)**: Dreams change dramatically over the course of learning reflecting learned expertise. Before learning (**D**) dreams can use only a few simple drawing routines and are largely unstructured; the majority are simple line segments. After twenty iterations of wake-sleep learning (**E**) dreams become more complex by recombining learned library concepts in ways never seen in the training tasks. Color shows the model's drawing trajectory, from start (blue) to finish (pink). Panels **(D-E)** illustrate the most interesting dreams found across five runs, before and after learning.

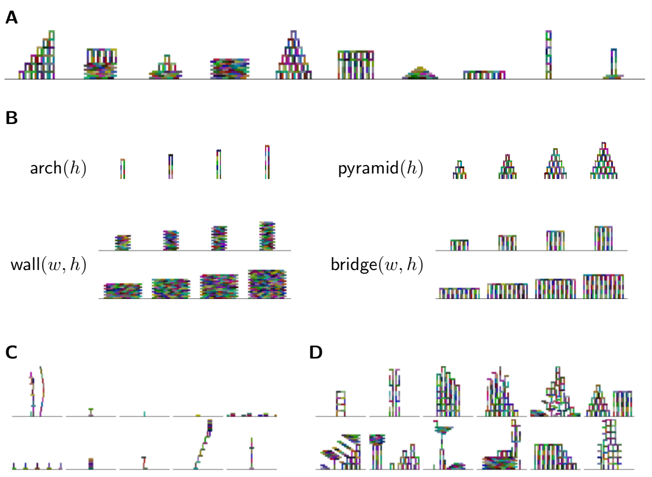


Figure 9. (A) Example tower building tasks. **(B)** Four learned library routines. Dreams both before **(C)** and after **(D)** learning show representative plans the model can imagine building. Dreams are selected from five different runs.

Input		MAP program	Samples
(210) (220) (41) (635) (38)	Full	(dd(d)*)	(220) (461)
	No Library	(dd(d)*.	(14u (2040)
	No Rec	(dd(d)*)	(68) (308)
$5.70 $3.40 $2.80 $5.40 $3.70	Full	$d.d0	$2.40 $3.30
	No Library	$d.d0	$5=50 $7#40
	No Rec	$(d)*.(d)*0	$.0 $873.30
(715) 967-2697 (608) 819-2220 (920) 988-2524 (608) 442-0253 (262) 723-4043	Full	(ddd) ddd-dddd	(099) 242-2029 (948) 452-9842
	No Library	.ddd) ddd.dddd	?773) 726-6866 m627) 674,0602
	No Rec	.(d)*) (d)*dd.(d)*	z40192) 51(8 =2) 279-876273

Figure 10. Results on held-out text generation tasks. System observes 5 strings ('Input') and infers a probabilistic regex ('MAP program'–regex primitives highlighted in red), from which it can imagine more examples of the text concept ('Samples'). No Library: Dreaming only (ablates library learning). No Rec: Abstraction only (ablates recognition model). See also Appendix Fig. 12

vectors, forces, and ratios. Rather than give DreamCoder these mathematical abstractions, we initialize the system with a much more generic basis — just a small number of recursive sequence manipulation primitives like `map` and `fold`, and arithmetic — and test whether it can learn an appropriate mathematical language of physics. After 8 wake/sleep cycles DreamCoder can solve 93.3% of the laws and identities in the dataset (best of five runs, mean 84.3%).[3] It first learns the building blocks of vector algebra, such as inner products,

[3]For held-out test problems, as Sec. 5.1 studies, mean accuracy matters most. For finding the "best" library, one looks at the run with max solved

vector sums, and norms (Fig. 11A). These operations have many algebraic symmetries, and accordingly our symmetry-breaking recognition model gives a slight edge over EC^2, which solves up to 86.6% of the tasks (best of three runs, mean 81.1%; compute limits restricted us to 3 runs). After acquiring these vector operations, DreamCoder then uses this mathematical vocabulary to construct concepts underlying multiple physical laws, such as the inverse square law schema that enables it to learn Newton's law of gravitation and Coulomb's law of electrostatic force, or the quadratic form underlying equations of ballistic motion and angular motion over time, effectively undergoing a 'change of basis' from the initial recursive sequence processing language to a physics-style basis.

Could DreamCoder also learn this recursive sequence manipulation language? We initialized the system with a minimal subset of 1959 Lisp primitives (`if`, `=`, `>`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`, all present in some form in McCarthy's 1959 Lisp [33]), and asked it to solve 20 basic programming tasks, like those used in introductory computer science classes (Figure 19). Crucially the initial language also includes primitive recursion (the Y combinator), which in principle allows learning to express any recursive function, but no other recursive function is given to start; previously we had sequestered recursion within higher-order functions (`map`, `fold`, ...) given to the learner as primitives. 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.

With enough compute time (roughly five days on 64 CPUs), DreamCoder learns to solve all 20 problems, and in so doing assembles a library equivalent to the modern repertoire of functional programming idioms, including `map`, `fold`, `zip`, `length`, and arithmetic operations such as building lists of natural numbers between an interval (see Fig. 11B). All these routines are expressible in terms of the higher-order function `fold` and its dual `unfold`, which, in a precise formal manner, are the two most elemental operations over recursive data – a discovery termed "origami programming" [19]. DreamCoder retraced the origami style: first reinventing fold, then unfold, and then defining all other recursive functions in terms of folding and unfolding.

In contrast, EC^2 uses a simpler library learning method. When run on this data set, it fails to uncover the origami basis: instead, it builds a bigger library (14 functions vs. 11 w/ DreamCoder) of less generic operators, and cannot solve the harder 'zipping' tasks. As with our list processing domain (Fig. 7B), sophisticated refactoring proves valuable for skillful functional programming.

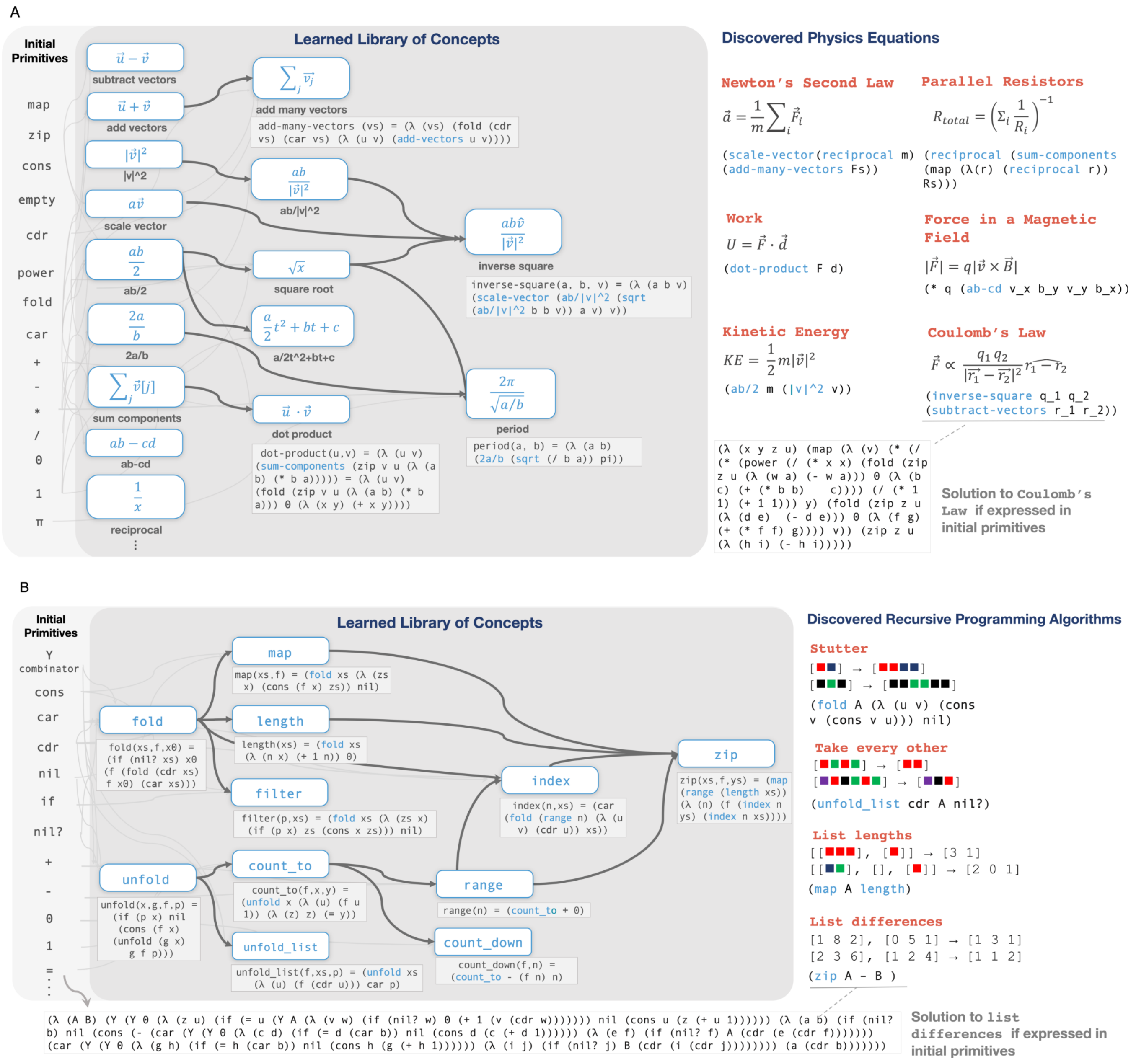
6 Discussion

Related work. Existing neural program synthesis systems pair a fixed DSL to a learned neural network that guides program search [4, 15, 18, 38, 42]. All such systems predict a distribution over programs conditioned on a specification, ranging from unigram distributions [34], to bigram-like ones [29, 30], to neural autoregressive models [12]. We build on these ideas by showing that a bigram distribution, trained correctly, suffices to break many syntactic symmetries in the domain-specific language, which is especially valuable when that language grows over time, as in DreamCoder.

Recent works for learning DSLs or libraries function by inferring reusable pieces of code [10, 14, 31, 32] (c.f. predicate invention in ILP [36] and ADFs in genetic programming [40]). These systems typically work through either memoization (caching reused subtrees, i.e. [10, 32], or reused command sequences, as an [29]), or antiunification (caching reused tree-templates that unify with program syntax trees, i.e. [14, 21, 24, 47]). Our abstraction sleep algorithm works through automatic refactoring, discovering and incorporating into the library syntax trees which are not present in the surface form of programs available to the learner.

Conceptual connections exist between DreamCoder and the cognitive science literatures on dual-process models [16], and human expertise [5, 6]. These models partition expertise into *declarative* and *procedural* knowledge. Declarative concepts have explicit, symbolic structure: i.e., for a programmer, concepts such as depth-first-search or sorting. Human experts also possess implicit procedural skill in deploying those concepts to quickly solve new problems. Compared to novices, experts more faithfully classify problems based on the "deep structure" of their solutions [5, 6], intuiting which compositions of concepts are likely to solve a task even before searching for a solution. Most learning-to-synthesize systems embed fixed declarative knowledge in a DSL, and acquire procedural skill by learning to map a problem specification to a distribution over programs in the DSL. In contrast, DreamCoder is more reminiscent of human learning in jointly acquiring both procedural and declarative knowledge from its experience solving problems in a given corpus.

Outlook. We have fused and improved two technologies for learned program synthesis–library learning and neurally-guided search–showing how they can synergistically bootstrap each other in a wake-sleep system applicable to classic synthesis domains and AI problems. Further combining library learning with hybrid discrete/continuous or neurosymbolic program representations [56] could be valuable, especially for scaling to real-world domains with messier structure than those considered here. Still, the full problem of automatically constructing a domain-specific language and learning to use it remains a major open challenge: learning not just the components of a DSL, but also its data types, idioms, and design patterns, and all the dimensions of expertise needed to use these tools effectively to solve new problems as human programmers do.



```
add-many-vectors (vs) = (λ (vs) (fold (cdr
vs) (car vs) (λ (u v) (add-vectors u v))))
```

```
inverse-square(a, b, v) = (λ (a b v)
(scale-vector (ab/|v|^2 (sqrt
(ab/|v|^2 b b v)) a v) v))
```

```
dot-product(u,v) = (λ (u v)
(sum-components (zip v u (λ (a
b) (* b a))))) = (λ (u v)
(fold (zip v u (λ (a b) (* b
a))) 0 (λ (x y) (+ x y))))
```

```
period(a, b) = (λ (a b)
(2a/b (sqrt (/ b a)) pi))
```

```
(scale-vector(reciprocal m)
(add-many-vectors Fs))
```

```
(reciprocal (sum-components
(map (λ(r) (reciprocal r))
Rs)))
```

```
(dot-product F d)
```

```
(* q (ab-cd v_x b_y v_y b_x))
```

```
(ab/2 m (|v|^2 v))
```

```
(inverse-square q_1 q_2
(subtract-vectors r_1 r_2))
```

```
(λ (x y z u) (map (λ (v) (* (/
(* (power (/ (* x x) (fold (zip
z u (λ (w a) (- w a))) 0 (λ (b
c) (+ (* b b)    c)))) (/ (* 1
1) (+ 1 1))) y) (fold (zip z u
(λ (d e)  (- d e))) 0 (λ (f g)
(+ (* f f) g)))) v)) (zip z u
(λ (h i) (- h i)))))
```

```
map(xs,f) = (fold xs (λ (zs
x) (cons (f x) zs)) nil)
```

```
fold(xs,f,x0) =
(if (nil? xs) x0
(f (fold (cdr xs)
f x0) (car xs)))
```

```
length(xs) = (fold xs
(λ (n x) (+ 1 n)) 0)
```

```
filter(p,xs) = (fold xs (λ (zs x)
(if (p x) zs (cons x zs))) nil)
```

```
index(n,xs) = (car
(fold (range n) (λ (u
v) (cdr u)) xs))
```

```
zip(xs,f,ys) = (map
(range (length xs))
(λ (n) (f (index n
ys) (index n xs))))
```

```
unfold(x,g,f,p) =
(if (p x) nil
(cons (f x)
(unfold (g x)
g f p)))
```

```
count_to(f,x,y) =
(unfold x (λ (u) (f u
1)) (λ (z) z) (= y))
```

```
range(n) = (count_to + 0)
```

```
unfold_list(f,xs,p) = (unfold xs
(λ (u) (f (cdr u))) car p)
```

```
count_down(f,n) =
(count_to - (f n) n)
```

```
(fold A (λ (u v) (cons
v (cons v u))) nil)
```

```
(unfold_list cdr A nil?)
```

```
(map A length)
```

```
(zip A - B )
```

```
(λ (A B) (Y (Y 0 (λ (z u) (if (= u (Y A (λ (v w) (if (nil? w) 0 (+ 1 (v (cdr w))))))) nil (cons u (z (+ u 1)))))) (λ (a b) (if (nil?
b) nil (cons (- (car (Y (Y 0 (λ (c d) (if (= d (car b)) nil (cons d (c (+ d 1)))))) (λ (e f) (if (nil? f) A (cdr (e (cdr f)))))))
(car (Y (Y 0 (λ (g h) (if (= h (car b)) nil (cons h (g (+ h 1)))))) (λ (i j) (if (nil? j) B (cdr (i (cdr j)))))))) (a (cdr b)))))))
```

Figure 11. (A) Learning a language for physical laws starting with recursive list routines such as `map` and `fold`. DreamCoder observes numerical data from 60 physical laws and relations, and learns concepts from vector algebra (e.g., dot products) and classical physics (e.g., inverse-square laws). Vectors are represented as lists of numbers. Physical constants are expressed in Planck units. **(B)** Learning a language for recursive list routines starting with only recursion and primitives found in 1959 Lisp. DreamCoder rediscovers the "origami" basis of functional programming, learning `fold` and `unfold` at the root, with other basic primitives as variations on one of those two families (e.g., `map` and `filter` in the `fold` family).

Acknowledgments

We thank L. Schulz, J. Andreas, T. Kulkarni, M. Kleiman-Weiner, J. M. Tenenbaum, M. Bernstein, S. Tenka, N. Fijalkow, P. Bielik, E. Spelke, and our anonymous reviewers for comments that greatly improved the manuscript. Supported by grants from the Air Force Office of Scientific Research, the Army Research Office, the NSF under Grant No. 1918839 and NSF-funded Center for Brains, Minds, and Machines, the MIT-IBM Watson AI Lab, Google, Microsoft and Amazon, and NSF graduate fellowships to K. Ellis and M. Nye. Opinions and findings in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438* (2017). https://doi.org/10.4204/EPTCS.260.9

[2] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *ICLR* (2016).

[3] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, Inc.

[4] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Execution-guided neural program synthesis. *ICLR* (2018).

[5] M.T.H. Chi, R. Glaser, and M.J. Farr. 1988. *The Nature of Expertise*. Taylor & Francis Group. https://doi.org/10.4324/9781315799681

[6] Michelene TH Chi, Paul J Feltovich, and Robert Glaser. 1981. Categorization and representation of physics problems by experts and novices. *Cognitive science* 5, 2 (1981). https://doi.org/10.1207/s15516709cog0502_2

[7] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014). https://doi.org/10.3115/v1/D14-1179

[8] Andrew Cropper. 2019. Playgol: Learning Programs Through Play. *IJCAI* (2019). https://doi.org/10.24963/ijcai.2019/841

[9] Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 207–212. https://doi.org/10.1145/582153.582176

[10] Eyal Dechter, Jon Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. 2013. Bootstrap Learning via Modular Concept Discovery. In *IJCAI*.

[11] David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: a theorem prover for program checking. *J. ACM* 52, 3 (2005), 365–473. https://doi.org/10.1145/1066100.1066102

[12] Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. 2017. Neural Program Meta-Induction. In *NIPS*.

[13] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. *ICML* (2017).

[14] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Library Learning for Neurally-Guided Bayesian Program Induction. In *NeurIPS*.

[15] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. 2019. Write, execute, assess: Program synthesis with a repl. In *Advances in Neural Information Processing Systems*. 9169–9178.

[16] Jonathan St BT Evans. 1984. Heuristic and analytic processes in reasoning. *British Journal of Psychology* 75, 4 (1984), 451–468. https://doi.org/10.1111/j.2044-8295.1984.tb01915.x

[17] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *PLDI*. https://doi.org/10.1145/2737924.2737977

[18] Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S. M. Ali Eslami, and Oriol Vinyals. 2018. Synthesizing Programs for Images using Reinforced Adversarial Learning. *ICML* (2018).

[19] Jeremy Gibbons. 2003. *Origami programming*. https://doi.org/10.1017/S0956796804245324

[20] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 317–330. https://doi.org/10.1145/1926385.1926423

[21] Robert John Henderson. 2013. *Cumulative learning in the lambda calculus*. Ph.D. Dissertation. Imperial College London. https://doi.org/10.25560/24759

[22] Luke Hewitt, Tuan Anh Le, and Joshua Tenenbaum. 2020. Learning to learn generative programs with Memoised Wake-Sleep. In *Conference on Uncertainty in Artificial Intelligence*. PMLR, 1278–1287.

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

[24] Irvin Hwang, Andreas Stuhlmüller, and Noah D Goodman. 2011. Inducing probabilistic programs by Bayesian program merging. *arXiv preprint arXiv:1110.5667* (2011).

[25] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[26] Kenichi Kurihara and Taisuke Sato. 2006. Variational Bayesian grammar induction for natural language. In *International Colloquium on Grammatical Inference*. Springer, 84–96. https://doi.org/10.1007/11872436_8

[27] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. 2015. Human-level concept learning through probabilistic program induction. *Science* 350, 6266 (2015), 1332–1338. https://doi.org/10.1126/science.aab3050

[28] Pat Langley. 1987. *Scientific discovery: Computational explorations of the creative processes*. MIT Press. https://doi.org/10.1177/027046768800800417

[29] Miguel Lázaro-Gredilla, Dianhuan Lin, J Swaroop Guntupalli, and Dileep George. 2019. Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs. *Science Robotics* 4, 26 (2019), eaav3150. https://doi.org/10.1126/scirobotics.aav3150

[30] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices* 53, 4 (2018), 436–449. https://doi.org/10.1145/3296979.3192410

[31] Percy Liang, Michael I. Jordan, and Dan Klein. 2010. Learning Programs: A Hierarchical Bayesian Approach. In *ICML*.

[32] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenenbaum, and Stephen Muggleton. 2014. Bias reformulation for one-shot function induction. In *ECAI 2014*. https://doi.org/10.3233/978-1-61499-419-0-525

[33] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3, 4 (1960), 184–195. https://doi.org/10.1145/367177.367199

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

[35] Microsoft. 2016. F# Guide: Units of Measure. https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/units-of-measure

[36] Stephen H Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. 2015. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning* 100, 1 (2015), 49–73. https://doi.org/10.1007/s10994-014-5471-y

[37] Stephen H Muggleton, Ute Schmid, Christina Zeller, Alireza Tamaddoni-Nezhad, and Tarek Besold. 2018. Ultra-Strong Machine Learning: comprehensibility of programs learned with ILP. *Machine Learning* 107, 7 (2018), 1119–1140. https://doi.org/10.1007/s10994-018-5707-3

[38] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to infer program sketches. *ICML* (2019).

[39] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press. I–XXI, 1–623 pages.

[40] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. 2008. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

[41] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538. https://doi.org/10.1145/2908080.2908093

[42] Illia Polosukhin and Alexander Skidanov. 2018. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335* (2018).

[43] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices* 50, 10 (2015), 107–126. https://doi.org/10.1145/2858965.2814310

[44] Stuart J. Russell and Peter Norvig. 2003. *Artificial Intelligence: A Modern Approach* (2 ed.). Pearson Education.

[45] Michael Schmidt and Hod Lipson. 2009. Distilling free-form natural laws from experimental data. *science* 324, 5923 (2009), 81–85. https://doi.org/10.1126/science.1165893

[46] Sanjit A. Seshia. 2012. Sciduction: Combining Induction, Deduction, and Structure for Verification and Synthesis. In *Proceedings of the Design Automation Conference (DAC)*. 356–365. https://doi.org/10.1145/2228360.2228425

[47] Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Oleksandr Polozov. 2019. Program Synthesis and Semantic Parsing with Learned Code Idioms. *NeurIPS* (2019).

[48] Vighnesh Shiv and Chris Quirk. 2019. Novel positional encodings to enable tree-based transformers. In *Advances in Neural Information Processing Systems*.

[49] Herbert A Simon, Patrick W Langley, and Gary L Bradshaw. 1981. Scientific discovery as problem solving. *Synthese* 47, 1 (1981), 1–27. https://doi.org/10.1080/02698599208573403

[50] Jake Snell, Kevin Swersky, and Richard Zemel. 2017. Prototypical Networks for Few-shot Learning. In *Advances in Neural Information Processing Systems*.

[51] Shashank Srivastava, Oleksandr Polozov, Nebojsa Jojic, and Christopher Meek. 2020. Learning Web-based Procedures by Reasoning over Explanations and Demonstrations in Context. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Association for Computational Linguistics, 7652–7662. https://doi.org/10.18653/v1/2020.acl-main.684

[52] Richard S Sutton, Doina Precup, and Satinder Singh. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence* 112, 1-2 (1999), 181–211.

[53] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 264–276. https://doi.org/10.1145/1480881.1480915

[54] David D. Thornburg. 1983. Friends of the Turtle. *Compute!* (March 1983).

[55] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. 2017. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 23–30. https://doi.org/10.1109/IROS.2017.8202133

[56] Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. 2018. Houdini: Lifelong learning as program synthesis. In *Advances in Neural Information Processing Systems*. 8687–8698.

[57] Philip Wadler. 1990. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*. ACM, 61–78. https://doi.org/10.1145/91556.91592

[58] Patrick Winston. 1972. The MIT Robot. *Machine Intelligence* (1972).