DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning

Kevin Ellis,[1,4,5] Catherine Wong,[1,4,5] Maxwell Nye,[1,4,5]
Mathias Sablé-Meyer,[1,3] Luc Cary,[1] Lucas Morales,[1,4,6] Luke Hewitt,[1,4,5]
Armando Solar-Lezama,[1,2,6] Joshua B. Tenenbaum[1,2,4,5]

[1]MIT [2]CSAIL [3]NeuroSpin [4]Center for Brains, Minds, and Machines

[5]Department of Brain and Cognitive Sciences [6]Department of Electrical Engineering and Computer Science

Expert problem-solving is driven by powerful languages for thinking about problems and their solutions. Acquiring expertise means learning these languages — systems of concepts, alongside the skills to use them. We present DreamCoder, a system that learns to solve problems by writing programs. It builds expertise by creating programming languages for expressing domain concepts, together with neural networks to guide the search for programs within these languages. A "wake-sleep" learning algorithm alternately extends the language with new symbolic abstractions and trains the neural network on imagined and replayed problems. DreamCoder solves both classic inductive programming tasks and creative tasks such as drawing pictures and building scenes. It rediscovers the basics of modern functional programming, vector algebra and classical physics, including Newton's and Coulomb's laws. Concepts are built compositionally from those learned earlier, yielding multi-layered symbolic representations that are interpretable and transferrable to new tasks, while still growing scalably and flexibly with experience.

goal

Method

Outcome

A longstanding dream in artificial intelligence (AI) has been to build a machine that learns like a child (*1*) – that grows into all the knowledge a human adult does, starting from much less. This dream remains far off, as human intelligence rests on many learning capacities not yet captured in artificial systems. While machines are typically designed for a single class of tasks, humans learn to solve an endless range and variety of problems, from cooking to calculus to graphic design. While machine learning is data hungry, typically generalizing weakly from experience, human learners can often generalize strongly from only modest experience. Perhaps most distinctively, humans build expertise: We acquire knowledge that can be communicated and extended, growing new concepts on those built previously to become better and faster learners the more we master a domain.

This paper presents DreamCoder, a machine learning system that aims to take a step closer to these human abilities – to efficiently discover interpretable, reusable, and generalizable knowledge across a broad range of domains. DreamCoder embodies an approach we call "wake-sleep Bayesian program induction", and the rest of this introduction explains the key ideas underlying it: what it means to view learning as program induction, why it is valuable to cast program induction as inference in a Bayesian model, and how a "wake-sleep" algorithm enables the model to grow with experience, learning to learn more efficiently in ways that make the approach practical and scalable.

Our formulation of learning as program induction traces back to the earliest days of AI (*2*): We treat learning a new task as search for a program that solves it, or which has intended behavior. Fig. 1 shows examples of program induction tasks in eight different domains that DreamCoder is applied to (Fig. 1A), along with an in-depth illustration of one task in the classic list-processing domain: learning a program that sorts lists of numbers (Fig. 1B), given a handful of input-output examples. Relative to purely statistical approaches, viewing learning as program induction brings certain advantages. Symbolic programs exhibit strong generalization properties–intuitively, they tend to extrapolate rather than merely interpolate. This also makes learning very sample-efficient: Just a few examples are often sufficient to specify any one function to be learned. By design, programs are richly human-interpretable: They subsume our standard modeling languages from science and engineering, and they expose knowledge that can be reused and composed to solve increasingly complex tasks. Finally, programs are universal: in principle, any Turing-complete language can represent solutions to the full range of computational problems solvable by intelligence.

Yet for all these strengths, and successful applications in a number of domains (*3–9*), program induction has had relatively limited impact in AI. A Bayesian formulation helps to clarify the challenges, as well as a path to solving them. The programming language we search in specifies the hypothesis space and prior for learning; the shorter a program is in that language, the higher its prior probability. While any general programming language can support program induction, previous systems have typically found it essential to start with a carefully engineered domain-specific language (DSL), which imparts a strong, hand-tuned inductive bias or prior. Without a DSL the programs to be discovered would be prohibitively long (low prior probability), and too hard to discover in reasonable search times. Even with a carefully tuned prior, though, search for the best program has almost always been intractable for general-purpose algorithms, because of the combinatorial nature of the search space. Hence most practical applications of program induction require not only a hand-designed DSL but also a search algorithm hand-designed to exploit that DSL for fast inference. Both these requirements limit the scalability and broad applicability of program induction.

DreamCoder addresses both of these bottlenecks by learning to compactly represent and efficiently induce programs in a given domain. The system learns to learn – to write better programs, and to search for them more efficiently – by jointly growing two distinct kinds of domain expertise: (1) explicit declarative knowledge, in the form of a learned domain-specific language, capturing conceptual abstractions common across tasks in a domain; and (2) implicit procedural knowledge, in the form of a neural network that guides how to use the learned language to solve new tasks, embodied by a learned domain-specific search strategy. In Bayesian terms, the system learns both a prior on programs, and an inference algorithm (parameterized by a neural network) to efficiently approximate the posterior on programs conditioned on observed task data.

DreamCoder learns both these ingredients in a self-supervised, bootstrapping fashion, growing them jointly across repeated encounters with a set of training tasks. This allows learning to scale to new domains, and to scale within a domain provided it receives sufficiently varied training tasks. Typically only a moderate number of tasks suffices to bootstrap learning in a new domain. For example, the list sorting function in Fig. 1B represents one of 109 tasks that the system cycles through, learning as it goes to construct a library of around 20 basic operations for lists of numbers which in turn become components for solving many new tasks it will encounter.

DreamCoder's learned languages take the form of multilayered hierarchies of abstraction (Fig. 1B, & Fig. 7A,B). These hierarchies are reminiscent of the internal representations in a deep neural network, but here each layer is built from symbolic code defined in terms of earlier code layers, making the representations naturally interpretable and explainable by humans. The network of abstractions grows

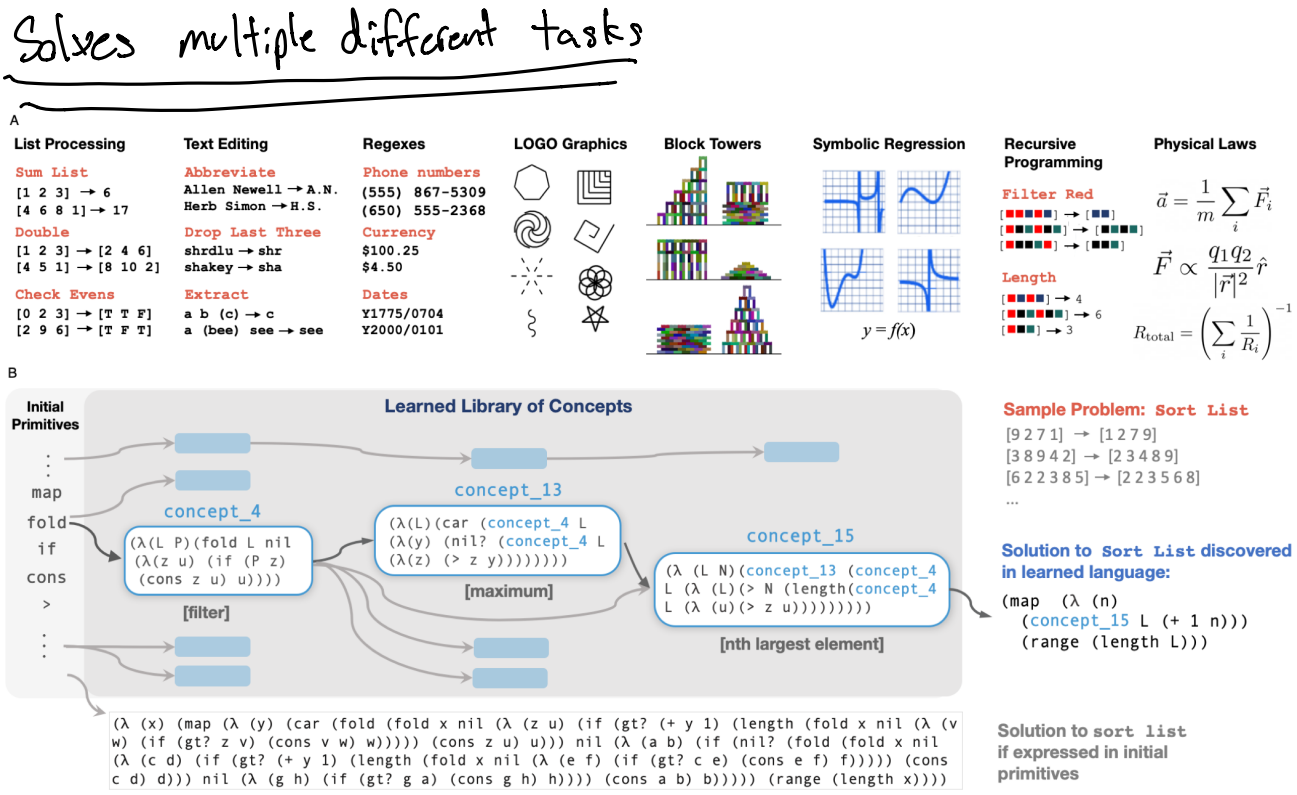

Figure 1: **(A)**: Learning tasks in many different domains can be formulated as inducing a program that explains a small number of input-output examples, or that generates an observed sequence, image or scene. DreamCoder successfully learns to synthesize programs for new tasks in each of these domains. **(B)**: An illustration of how DreamCoder learns to solve problems in one domain, processing lists of integers. Problems are specified by input-output pairs exemplifying a target function (e.g., 'Sort List'). Given initial primitives (left), the model 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 concepts. The learned library enables simpler, faster, and more interpretable problem solving: A typical solution to 'Sort List' (right), discovered after six iterations of learning, can be expressed with just five function calls using the learned library and is found in less than 10 minutes of search. The code reads naturally as "get the n^{th} largest number, for $n = 1, 2, 3, \ldots$." 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.

progressively over time, building each concept on those acquired before, inspired by how humans build conceptual systems: we learn algebra before calculus, and only after arithmetic; we learn to draw simple shapes before more complex designs. For example, in the list processing example (Fig. 1B), our model comes to sort sequences of numbers by invoking a library component four layers deep – take the n^{th} largest element – and this component in turn calls lower-level learned concepts: maximum, and filter. Equivalent programs could in principle be written in the starting 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 learners reach: they would never be found during a reasonably bounded search. Only with acquired domain-specific expertise do most problems become practically solvable.

DreamCoder gets its name from how it grows domain knowledge iteratively, in "wake-sleep" cycles loosely inspired by the memory consolidation processes that occur during different stages of

sleep (*10, 11*). In general, wake-sleep Bayesian learning (*12*) iterates between training a probabilistic *generative model* that defines the learner's prior alongside a neural network *recognition model* that learns to invert this generative model given new data. During "waking" the generative model is used to interpret new data, guided by the recognition model. The recognition model is learned offline during "sleep," from imagined data sets ("dreams" or "fantasies") sampled from the generative model.

DreamCoder develops the wake-sleep approach for learning to learn programs: Its learned language defines a generative model over programs and tasks, where each program solves a particular hypothetical task; its neural network learns to recognize patterns across tasks in order to best predict program components likely to solve any given new task. During waking, the system is presented with data from several tasks and attempts to synthesize programs that solve then, using the neural recognition model to propose candidate programs. Learning occurs during two distinct but interleaved sleep phases, alternately growing the learned language (generative model) by consolidating new abstractions from programs found during waking, and training the neural network (recognition model) on "fantasy" programs sampled from the generative model. This wake-sleep architecture builds on and further integrates a pair of ideas, Bayesian multitask program learning (*5, 13, 14*) and neurally-guided program synthesis (*15, 16*), which have been separately influential in the recent literature but have only been brought together in our work starting with the EC^2 algorithm (*17*), and now made much more scalable in DreamCoder (see S3 for further discussion of prior work).

The resulting system has wide applicability. We describe applications to eight domains (Fig. 1A): 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. All of our tasks involve inducing programs from very minimal data, e.g., five to ten 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). Taken together, we hope these applications illustrate the potential for program induction to become a practical, general-purpose, and data-efficient approach to building intepretable, reusable knowledge in artificial intelligence systems.

Wake/Sleep Program Learning

We now describe the specifics of learning in DreamCoder, beginning with an overview of the algorithm and its mathematical formulation, then turning to the details of its three phases. Learning proceeds iteratively, with each iteration (Eq. 1, Fig. 2) cycling through a wake phase of trying to solve tasks interleaved with two sleep phases for learning to solve new tasks. In the **wake** phase (Fig. 2 top), the system searches for programs that solve tasks drawn from a training set, guided by the neural recognition model which ranks candidate programs based on the observed data for each task. Candidate programs are scored according to how well they solve the presented tasks, and how plausible they are a priori under the learned generative model for programs. The first sleep phase, which we refer to as **abstraction** (Fig. 2 left), grows the library of programming primitives (the generative model) by replaying experiences from waking, finding common program fragments from task solutions, and abstracting out these fragments into new code primitives. This mechanism increases the breadth and depth of the learner's declarative knowledge, its learned library as in Fig. 1B or Fig. 7, when viewed as a network. The second sleep phase, which we refer to as **dreaming** (Fig. 2 right), improves the learner's procedural skill in code-writing by training the neural network that helps search for programs.

The neural recognition model is trained on replayed experiences as well as "fantasies", or programs sampled randomly from the learned library as a generative model. These random programs define tasks which the system solves during the dream phase, and the neural network is trained to predict the solutions found given the observable data for each imagined task.

Viewed as a probabilistic inference problem, DreamCoder observes a training set of tasks, written X, and infers both a program ρ_x solving each task $x \in X$, as well as a prior distribution over programs likely to solve tasks in the domain (Fig. 2 middle). This prior is encoded by a library, written L, which defines a generative model over programs, written $\mathrm{P}[\rho|L]$ (see S4.3). The neural network helps to find programs solving a task by predicting, conditioned on the observed examples for that task, an approximate posterior distribution over programs likely to solve it. The network thus functions as a **recognition model** that is trained jointly with the generative model, in the spirit of the Helmholtz machine (*12*). We write $Q(\rho|x)$ for the approximate posterior predicted by the recognition model. At a high level wake/sleep cycles correspond to iterating the following updates, illustrated in Fig. 2; these updates serve to maximize a lower bound on the posterior over L given X (S4.1).

$$\begin{aligned}
\rho_x = & \underset{\substack{\rho: \\ Q(\rho|x) \text{ is large}}}{\arg\max} \ \mathrm{P}[\rho|x, L] \propto \mathrm{P}[x|\rho]\mathrm{P}[\rho|L], \text{ for each task } x \in X && \textit{Wake} \\
L = & \arg\max_{L} \mathrm{P}[L] \prod_{x \in X} \max_{\rho \text{ a refactoring of } \rho_x} \mathrm{P}[x|\rho]\mathrm{P}[\rho|L] && \textit{Sleep: Abstraction} \\
\text{Train } & Q(\rho|x) \approx \mathrm{P}[\rho|x, L], \text{ where } x \sim X \text{ ('replay') or } x \sim L \text{ ('fantasy')} && \textit{Sleep: Dreaming}
\end{aligned} \quad (1)$$

where $\mathrm{P}[L]$ is a description-length prior over libraries (S4.5) and $\mathrm{P}[x|\rho]$ is the likelihood of a task $x \in X$ given program ρ. For example, this likelihood is 0 or 1 when x is specified by inputs/outputs, and when learning a probabilistic program, the likelihood is the probability of the program generating the observed task.

This 3-phase inference procedure works through two distinct kinds of bootstrapping. During each sleep cycle the next library bootstraps off the concepts learned during earlier cycles, growing an increasingly deep learned library. Simultaneously the generative and recognition models bootstrap each other: A more specialized 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 accompanying program synthesis. Higher-level library routines allow tasks to be solved with fewer function calls, effectively reducing the *depth* of search. The neural recognition model down-weights unlikely trajectories through the search space of all programs, effectively reducing the *breadth* of search.[1]

Wake phase. Waking consists of searching for task-specific programs with high posterior probability, or programs that combine high likelihood (because they solve a task) and high prior probability (because they have short description length in the current language). During each Wake cycle we sample tasks from a random minibatch of the training set (or, depending on domain size and complexity, the entire training set). We then search for programs solving each of these tasks by enumerating programs in decreasing order of their probability under the recognition model $Q(\rho|x)$, and checking if a program ρ assigns positive probability to solving that task ($\mathrm{P}[x|\rho] > 0$). Because the model may

[1] We thank Sam Tenka for this observation. In particular, the difficulty of search during waking is roughly proportional to $\text{breadth}^{\text{depth}}$, where depth is the total size of a program and breadth is the number of library functions with high probability at each decision point in the search tree spanning the space of all programs. Library learning decreases depth at the expense of breadth, while training a neural recognition model effectively decreases breadth by decreasing the number of bits of entropy consumed by each decision (function call) made when constructing a program solving a task.

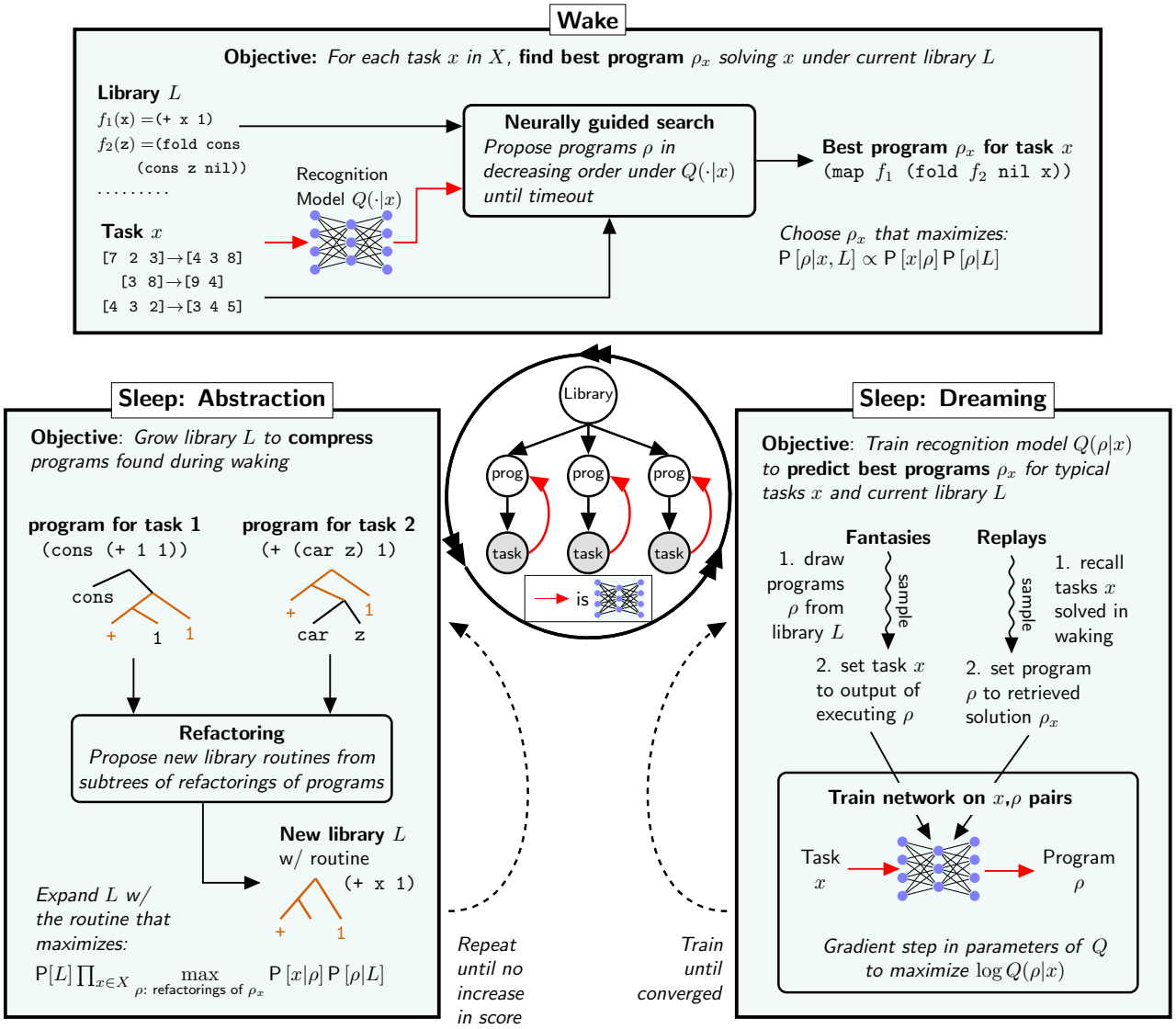

Figure 2: DreamCoder's basic algorithmic cycle, which serves to perform approximate Bayesian inference for the graphical model diagrammed in the **middle**. The system observes programming tasks (e.g., input/outputs for list processing or images for graphics programs), which it explains with latent programs, while jointly inferring a latent library capturing cross-program regularities. A neural network, called the *recognition model* (red arrows) is trained to quickly infer programs with high posterior probability. The Wake phase (**top**) infers programs while holding the library and recognition model fixed. A single task, 'increment and reverse list', is shown here. The Abstraction phase of sleep (**left**) updates the library while holding the programs fixed by refactoring programs found during waking and abstracting out common components (highlighted in orange). Program components that best increase a Bayesian objective (intuitively, that best compress programs found during waking) are incorporated into the library, until no further increase in probability is possible. A second sleep phase, Dreaming (**right**) trains the recognition model to predict an approximate posterior over programs conditioned on a task. The recognition network is trained on 'Fantasies' (programs sampled from library) and 'Replays' (programs found during waking).

find many programs that solve a specific task, we store a small beam of the $k = 5$ programs with the highest posterior probability $\mathrm{P}[\rho|x, L]$, and marginalize over this beam in the sleep updates of Eq. 1. We represent programs as polymorphically typed λ-calculus expressions, an expressive formalism including conditionals, variables, higher-order functions, and the ability to define new functions.

Abstraction phase. During the abstraction sleep phase, the model grows its library of concepts with the goal of discovering specialized abstractions 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, and the abstraction sleep objective (Eq. 1) is equivalent to minimizing the description length of the library ($-\log \mathrm{P}[D]$) plus the description lengths of refactorings of programs found during waking ($\sum_x \min_{\rho \text{ refactors } \rho_x} - \log \mathrm{P}[x|\rho]\mathrm{P}[\rho|D]$). Intuitively, we "compress out" reused code to maximize a Bayesian criterion, but rather than compress out reused syntactic structures, we refactor the programs to expose reused semantic patterns.

Code can be refactored in infinitely many ways, so we bound the number of λ-calculus evaluation steps separating a program from its refactoring, giving a finite but typically astronomically large set of refactorings. Fig. 3 diagrams the model discovering one of the most elemental building blocks of modern functional programming, the higher-order function `map`, starting from a small set of universal primitives, including recursion (via the Y-combinator). In this example there are approximately 10^{14} possible refactorings – a quantity that grows exponentially both as a function of program size and as a function of the bound on evaluation steps. To resolve this exponential growth we introduce a new data structure for representing and manipulating the set of refactorings, combining ideas from version space algebras (*18–20*) and equivalence graphs (*21*), and we derive a dynamic program for its construction (supplementary S4.5). This data structure grows polynomially with program size, owing to a factored representation of shared subtrees, but grows exponentially with a bound on evaluation steps, and the exponential term can be made small (we set the bound to 3) without performance loss. This results in substantial efficiency gains: A version space with 10^6 nodes, calculated in minutes, can represent the 10^{14} refactorings in Fig. 3 that would otherwise take centuries to explicitly enumerate and search.

Dreaming phase. During the dreaming sleep phase, the system trains its recognition model, which later speeds up problem-solving during waking by guiding program search. We implement recognition models as neural networks, injecting domain knowledge through the network architecture: for instance, when inducing graphics programs from images, we use a convolutional network, which imparts a bias toward useful image features. We train a recognition network on (program, task) pairs drawn from two sources of self-supervised data: *replays* of programs discovered during waking, and *fantasies*, or programs drawn from L. 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 neither is it a big data problem. We typically train DreamCoder on 100-200 tasks, which is too few examples for a high-capacity neural network. After the model learns a library customized to the domain, we can draw unlimited samples or 'dreams' to train the recognition network.

Our dream phase works differently from a conventional wake-sleep (*12*) dream phase. A classic wake-sleep approach would sample a random program from the generative model, execute it to generate a task, and train the recognition network to predict the sampled program from the sampled task. We instead think of dreaming as creating an endless stream of random problems, which we then solve during sleep in an active process using the same program search process as in waking. We then train the recognition network to predict the solutions discovered, conditioned on the problems. Specifically, we train Q to perform MAP inference by maximizing $\mathrm{E}\left[\log Q\left(\left(\arg\max_\rho \mathrm{P}[\rho|x, L]\right)|x\right)\right]$, where the

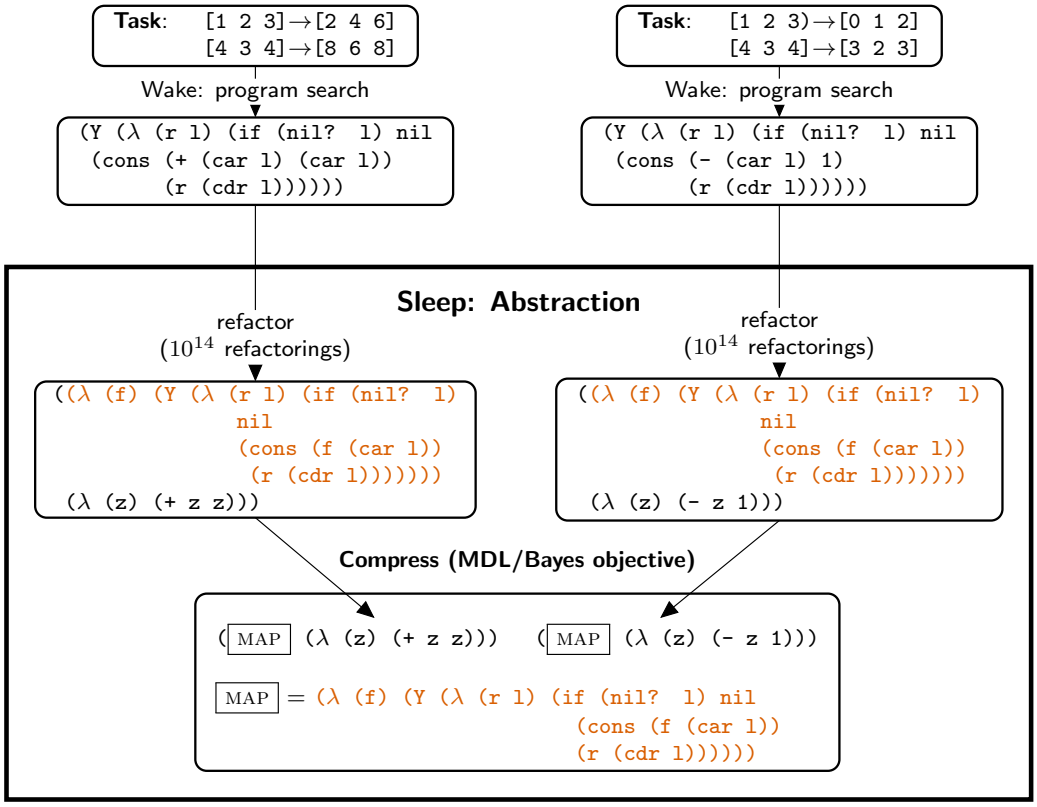

Figure 3: Programs found as solutions during waking are refactored – or rewritten in semantically equivalent but syntactically distinct forms – during the sleep abstraction phase, to expose candidate new primitives for growing DreamCoder's learned library. Here, solutions for two simple list tasks (top left, 'double each list element'; top right, 'subtract one from each list element') are first found using a very basic primitive set, which yields correct but inelegant programs. During sleep, DreamCoder efficiently searches an exponentially large space of refactorings for each program; a single refactoring of each is shown, with a common subexpression highlighted in orange. This expression corresponds to `map`, a core higher-order function in modern functional programming that applies another function to each element of a list. Adding `map` to the library makes existing problem solutions shorter and more interpretable, and crucially bootstraps solutions to many harder problems in later wake cycles.

expectation is taken over tasks. Taking this expectation over the empirical distribution of tasks trains Q on replays; taking it over samples from the generative model trains Q on fantasies. We train on a 50/50 mix of replays and fantasies; for fantasies mapping inputs to outputs, we sample inputs from the training tasks. Although one could train Q to perform full posterior inference, our MAP objective has the advantage of teaching the recognition network to find a simplest canonical solution for each problem. More technically, our MAP objective acts to break syntactic symmetries in the space of programs by forcing the network to place all its probability mass onto a single member of a set of syntactically distinct but semantically equivalent expressions. Hand-coded symmetry breaking has proved vital for many program synthesizers (*22, 23*); see S4.6 for theoretical and empirical analyses of DreamCoder's learned symmetry breaking.

Results

We first experimentally investigate DreamCoder within two classic benchmark domains: list processing and text editing. In both cases we solve tasks specified by a conditional mapping (i.e., input/output examples), starting with a generic functional programming basis, including routines like `map`, `fold`, `cons`, `car`, `cdr`, etc. Our list processing tasks comprise 218 problems taken from (*17*), split 50/50 test/train, each with 15 input/output examples. In solving these problems, DreamCoder composed around 20 new library routines (S1.1), and rediscovered higher-order functions such as `filter`. Each round of abstraction built on concepts discovered in earlier sleep cycles — for example the model first learns `filter`, then uses it to learn to take the maximum element of a list, then uses that routine to learn a new library routine for extracting the n^{th} largest element of a list, which it finally uses to sort lists of numbers (Fig. 1B).

Synthesizing programs that edit text is a classic problem in the programming languages and AI literatures (*18*), and algorithms that synthesize text editing programs ship in Microsoft Excel (*7*). These systems would, for example, see the mapping "Alan Turing" $\rightarrow$ "A.T.", and then infer a program that transforms "Grace Hopper" to "G.H.". Prior text-editing program synthesizers rely on hand-engineered libraries of primitives and hand-engineered search strategies. Here, we jointly learn both these ingredients and perform comparably to a state-of-the-art domain-general program synthesizer. We trained our system on 128 automatically-generated text editing tasks, and tested on the 108 text editing problems from the 2017 SyGuS (*24*) program synthesis competition.[2] Prior to learning, DreamCoder solves 3.7% of the problems within 10 minutes with an average search time of 235 seconds. After learning, it solves 79.6%, and does so much faster, solving them in an average of 40 seconds. The best-performing synthesizer in this competition (CVC4) solved 82.4% of the problems — but here, the competition conditions are 1 hour & 8 CPUs per problem, and with this more generous compute budget we solve 84.3% of the problems. SyGuS additionally comes with a different hand-engineered library of primitives *for each text editing problem*. Here we learned a single library of text-editing concepts that applied generically to any editing task, a prerequisite for real-world use.

We next consider more creative problems: generating images, plans, and text. Procedural or generative visual concepts — from Bongard problems (*25*), to handwritten characters (*5, 26*), to Raven's progressive matrices (*27*) — are studied across AI and cognitive science, because they offer a bridge between low-level perception and high-level reasoning. Here we take inspiration from LOGO Turtle graphics (*28*), tasking our model with drawing a corpus of 160 images (split 50/50 test/train; Fig. 4A) while equipping it with control over a 'pen', along with imperative control flow, and arithmetic operations on angles and distances. After training DreamCoder for 20 wake/sleep cycles, we inspected the learned library (S1.1) and found interpretable parametric drawing routines corresponding to the families of visual objects in its training data, like polygons, circles, and spirals (Fig. 4B) – without supervision the system has learned the basic types of objects in its visual world. 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. 4C).

Visualizing the system's dreams across its learning trajectory shows 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. 4D), offering

[2]We compare with the 2017 benchmarks because 2018 onward introduced non-string manipulation problems; custom string solvers such as FlashFill (*7*) and the latest custom SyGuS solvers are at ceiling for these newest problems.

limited value for training the recognition model. After learning, the system's dreams are richly structured (Fig. 4E), compositionally recombining latent building blocks and motifs acquired from the training data in creative ways never seen in its waking experience, but ideal for training a broadly generalizable recognition model (*29*).

Inspired by the classic AI 'copy demo' – where an agent looks at a tower made of toy blocks then re-creates it (*30*) – we next gave DreamCoder 107 tower 'copy tasks' (split 50/50 test/train, Fig. 5A), where the system observes both an image of a tower and the locations of each of its blocks, and must write a program that plans how a simulated hand would build the tower. The system starts with the same control flow primitives as with LOGO graphics. Inside its learned library we find parametric 'options' (*31*) for building blocks towers (Fig. 5B), including concepts like arches, staircases, and bridges, which one also sees in the model's dreams (Fig. 5C-D).

Next we consider few-shot learning of probabilistic generative concepts, an ability that comes naturally to humans, from learning new rules in natural language (*32*), to learning routines for symbols and signs (*5*), to learning new motor routines for producing words (*33*). We first task DreamCoder with inferring a probabilistic regular expression (or Regex, see Fig. 1A for examples) from a small number of strings, where these strings are drawn from 256 CSV columns crawled from the web (data from (*34*), tasks split 50/50 test/train, 5 example strings per concept). The system learns to learn regular expressions that describe the structure of typically occurring text concepts, such as phone numbers, dates, times, or monetary amounts (Fig. S5). 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, though DreamCoder knows nothing about dollar amounts it can infer an 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`.

We last consider inferring real-valued parametric equations generating smooth trajectories (see S2.1.6 and Fig. 1A, 'Symbolic Regression'). Each task is to fit data generated by a specific curve – either a rational function or a polynomial of up to degree 4. We initialize DreamCoder with addition, multiplication, division, and, critically, arbitrary real-valued parameters, which we optimize over via inner-loop gradient descent. We model each parametric program as probabilistically generating a family of curves, and penalize use of these continuous parameters via the Bayesian Information Criterion (BIC) (*35*). Our Bayesian machinery learns to home in on programs generating curves that explain the data while parsimoniously avoiding extraneous continuous parameters. For example, given real-valued data from $1.7x^2 - 2.1x + 1.8$ it infers a program with three continuous parameters, but given data from $\frac{2.3}{x-2.8}$ it infers a program with two continuous parameters.

Quantitative analyses of DreamCoder across domains

To better understand how DreamCoder learns, 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). We contrast with several baselines: *Exploration-Compression* (*13*), which alternately searches for programs, and then compresses out reused components into a learned library, but without our refactoring algorithm; *Neural Program*

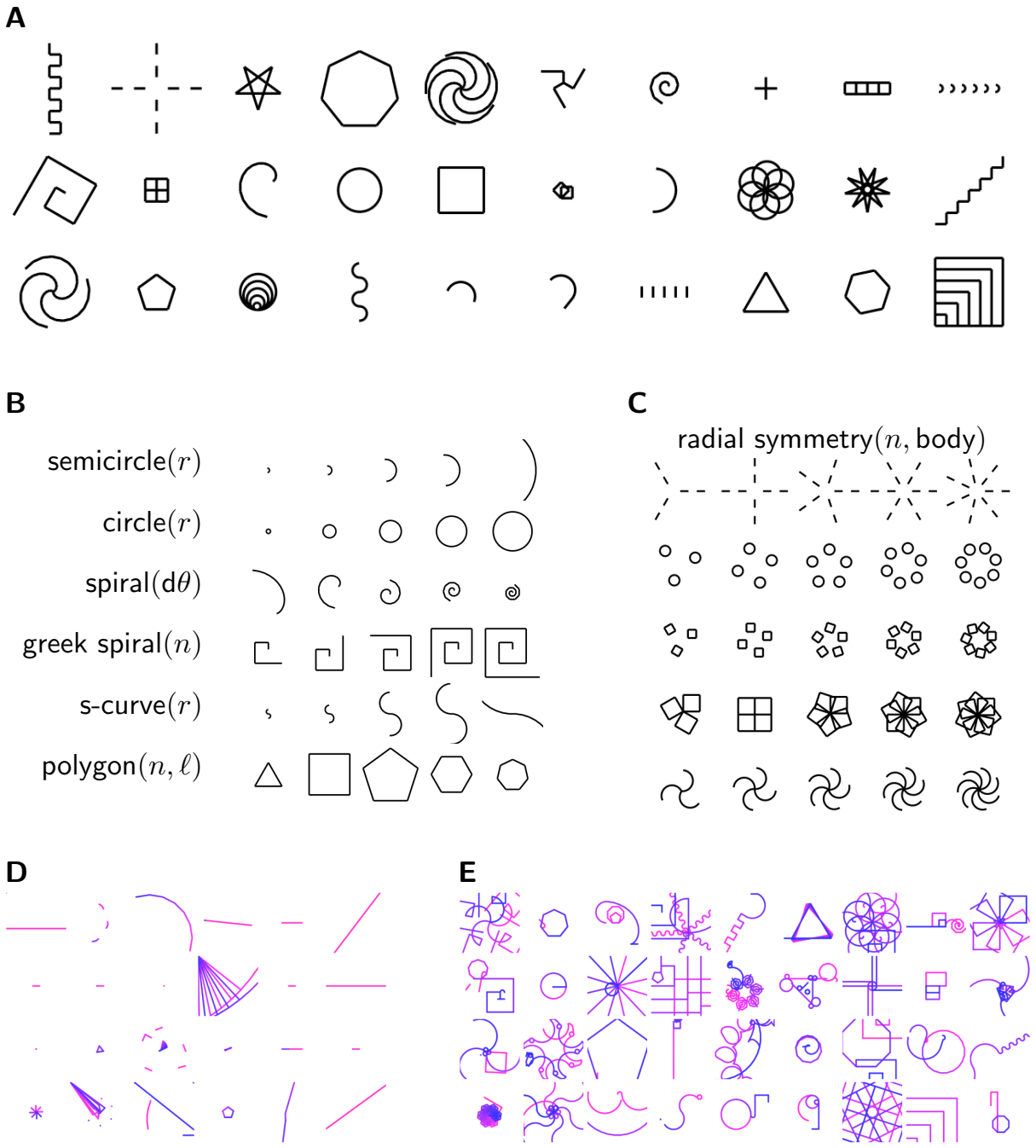

Figure 4: **(A):** 30 (out of 160) LOGO graphics tasks. The model writes programs controlling a 'pen' that draws the target picture. **(B-C):** Example learned library routines include both parametric routines for drawing families of curves (**B**) as well as primitives that take entire programs as input (**C**). Each row in **B** shows the same code executed with different parameters. Each image in **C** shows the same code executed with different parameters and a different subprogram as input. **(D-E)**: Dreams, or programs sampled by randomly assembling functions from the model's library, 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. Dreams are sampled from the prior learned over tasks solved during waking, and provide an infinite stream of data for training the neural recognition model. 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, both before and after learning. Fig. S6 shows 150 random dreams at each stage.

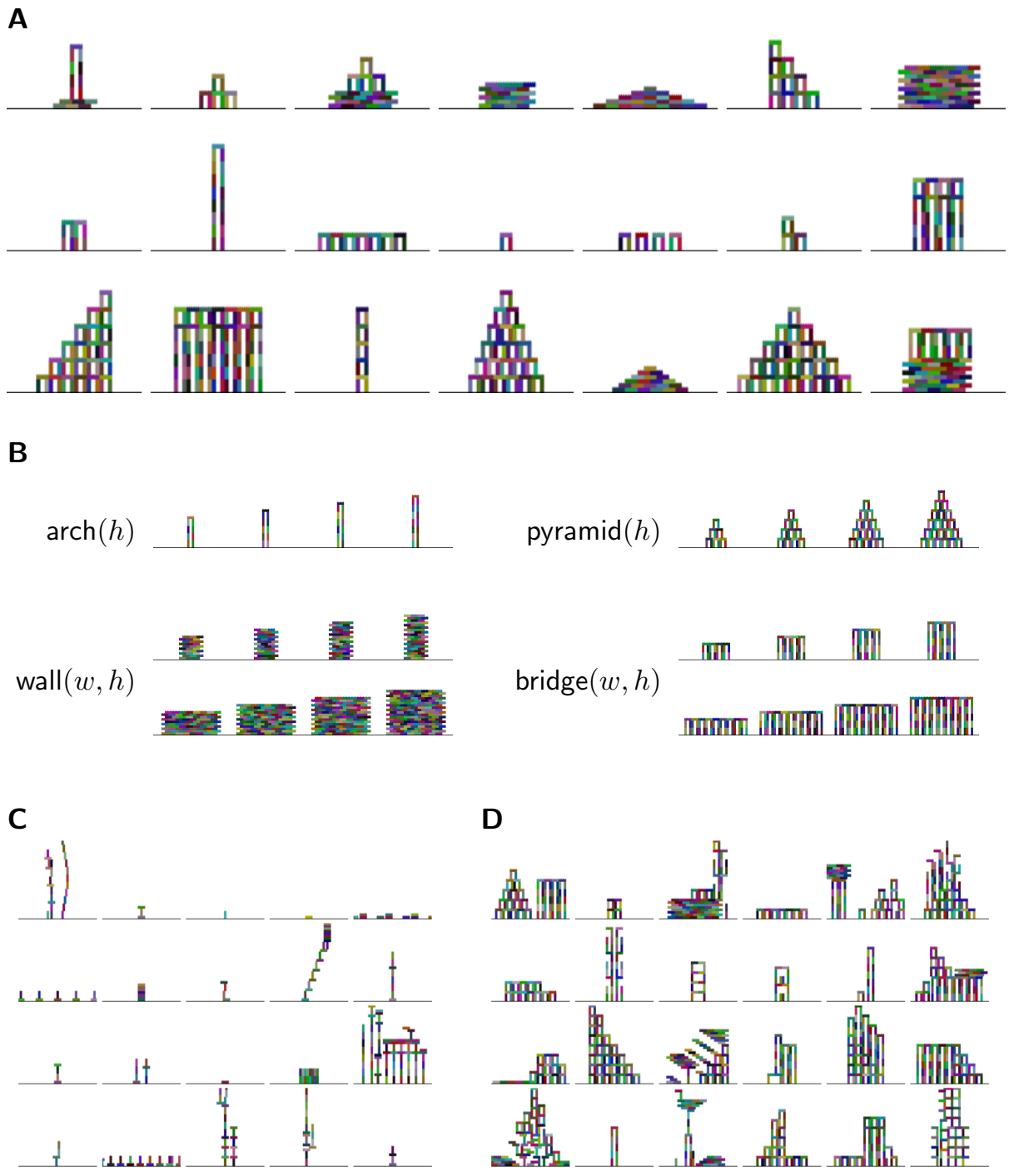

Figure 5: **(A)** 21 (out of 107) tower building tasks. The model writes a program controlling a 'hand' that builds the target tower. **(B)** Four learned library routines. These components act like parametric options (*31*), giving human-understandable, higher-level building blocks that the system can use to plan. Dreams both before and after learning **(C-D)** show representative plans the system can imagine building. After 20 wake-sleep iterations (**D**) the model fantasizes complex structures it has not seen during waking, but that combine building motifs abstracted from solved tasks in order to provide training data for a robust neural recognition model. Dreams are selected from five different runs; Fig. S7 shows 150 random dreams at each stage.

Synthesis, which trains a RobustFill (*16*) model on samples from the initial library; and *Enumeration*, which performs type-directed enumeration (*23*) for 24 hours per task, generating and testing up to 400 million programs for each task. To isolate the role of compression in learning good libraries, we also construct two *Memorize* baselines. These variants extend the library by incorporating task solutions wholesale as new primitives; they do not attempt to compress but simply memorize solutions found during waking for potential reuse on new problems (cf. (*36*)). We evaluate memorize variants both with and without neural recognition models.

Across domains, our model always solves the most held-out tasks (Fig. 6A; see Fig. S13 for memorization baselines) and generally solves them in the least time (mean 54.1s; median 15.0s; Fig. S11). These results establish that each of DreamCoder's core components – library learning with refactoring and compression during the sleep-abstraction phase, and recognition model learning during the sleep-dreaming phase – contributes substantively 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 no alternative model ever solves more than 60% of held-out tasks while DreamCoder learns to solve nearly 100% of them. The time needed to train DreamCoder to the points of convergence shown in Fig. 6A varies across domains, but typically takes around a day using moderate compute resources (20-100 CPUs).

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 the presence of a learned 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. 6B, Fig. S1). 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 the depth and breadth of the learned symbolic representation.

Insight into how DreamCoder's recognition model bootstraps the learned library comes from looking at how these representations jointly embed the similarity structure of tasks to be solved. DreamCoder first encodes a task in the activations of its recognition network, then rerepresents that task in terms of a symbolic program solving it. Over the course of learning, these implicit initial representations realign with the explicit structure of the final program solutions, as measured by increasing correlations between the similarity of problems in the recognition network's activation space and the similarity of code components used to solve these problems (see Fig. S4; $p < 10^{-4}$ using χ^2 test pre/post learning). Visualizing these learned task similarities (with t-SNE embeddings) suggests that, as the model gains a richer conceptual vocabulary, its representations evolve to group together tasks sharing more abstract commonalities (Fig. S3) – possibly analogous to how human domain experts learn to classify problems by the underlying principles that govern their solution rather than superficial similarities (*37, 38*).

From learning libraries to learning languages

Our experiments up to now have studied 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

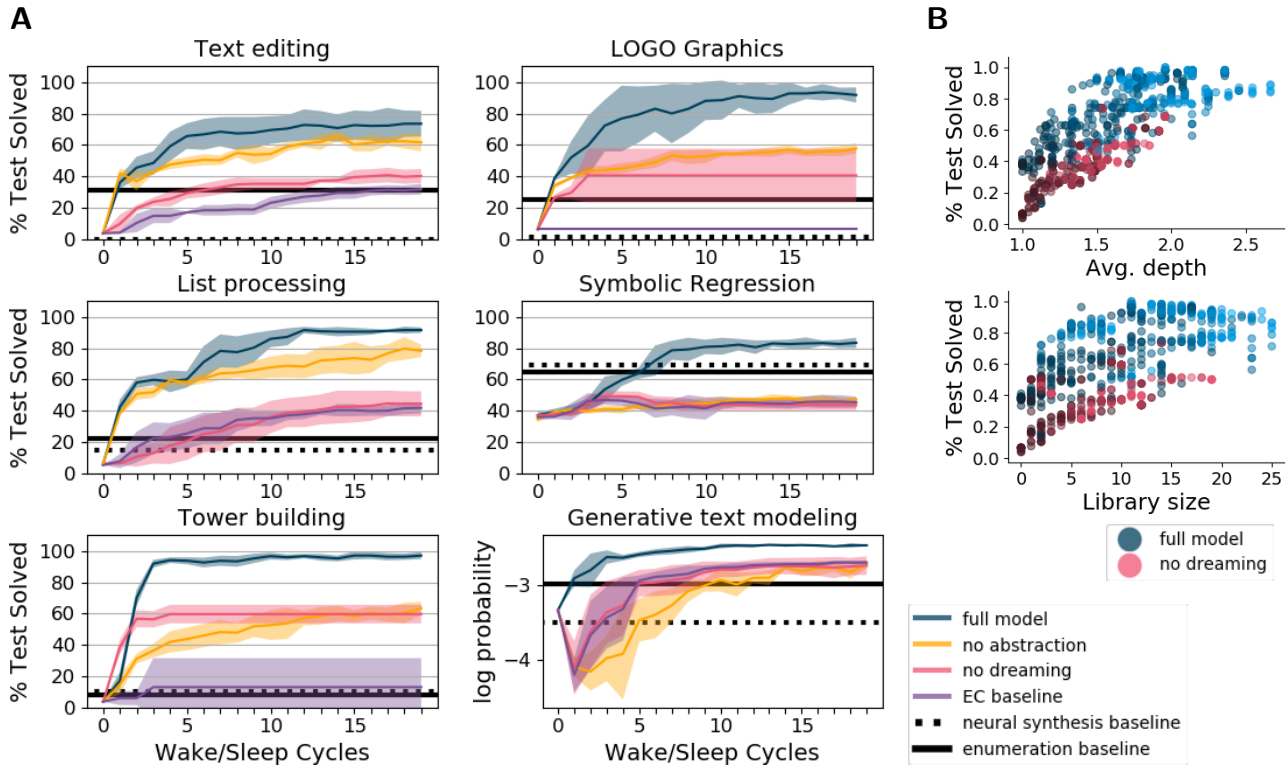

Figure 6: Quantitative comparisons of DreamCoder performance with ablations and baseline program induction methods; further baselines shown in Fig. S13. **(A)** Held-out test set accuracy, across 20 iterations of wake/sleep learning for six domains. Generative text modeling plots show posterior predictive likelihood of held-out strings on held out tasks, normalized per-character. Error bars: ± 1 std. dev. over five runs. **(B)** 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. Larger, deeper libraries are correlated with solving more tasks. The dreaming phase bootstraps these deeper, broader libraries, and also, for a fixed library structure, dreaming leads to higher performance.

concepts allowing it to solve all the problems in a domain?

Motivated by classic work on inferring physical laws from experimental data (*39–41*), 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. The full dataset includes data generated from many well-known laws in mechanics and electromagnetism, which are naturally expressed using concepts like 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. Indeed, after 8 wake/sleep cycles DreamCoder learns 93% of the laws and identities in the dataset, by first learning the building blocks of vector algebra, such as inner products, vector sums, and norms (Fig. 7A). It 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, 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 (`car`, `cdr`, `cons`, ...) and asked it to solve 20

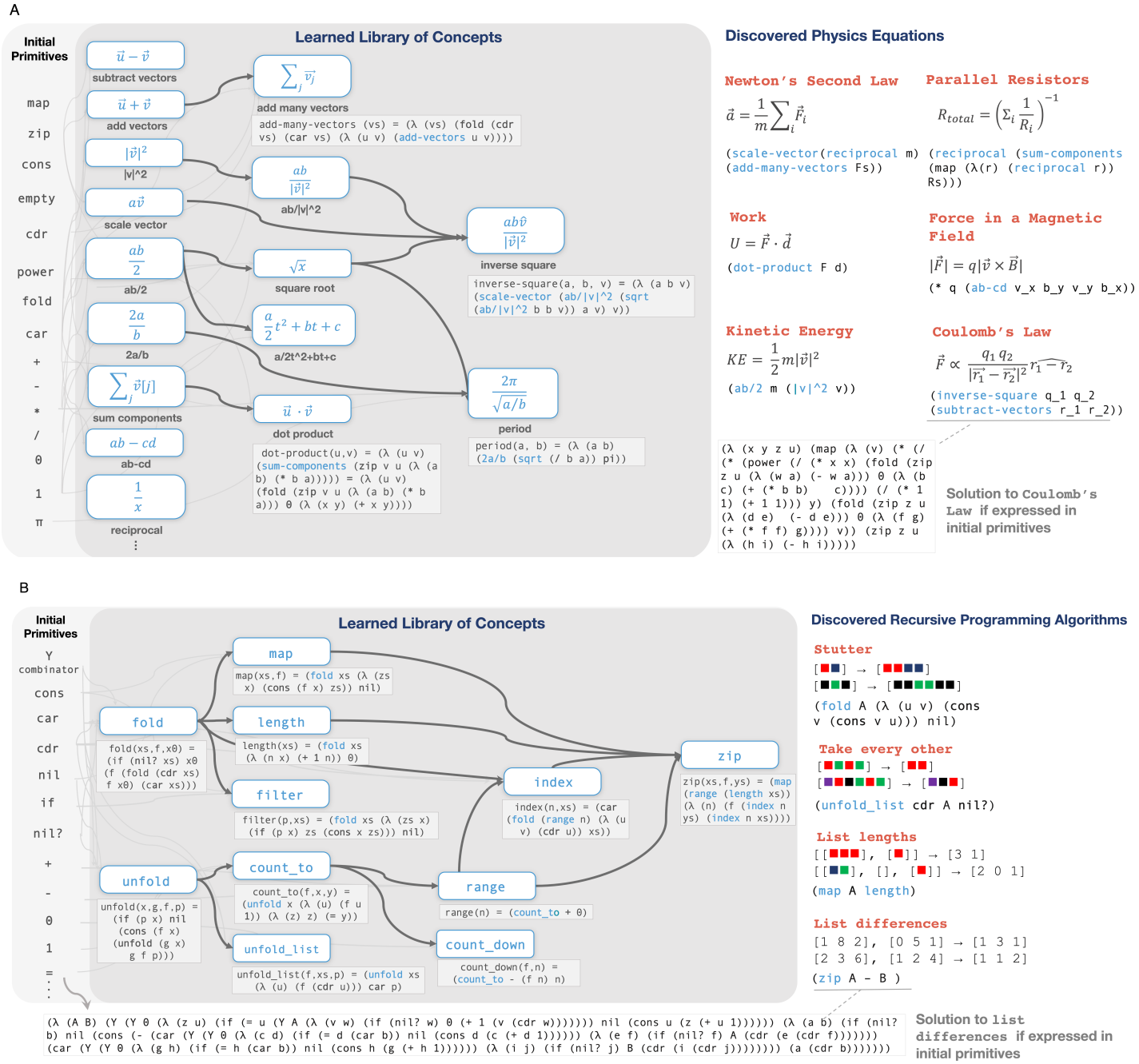

Figure 7: DreamCoder develops languages for physical laws (starting from recursive functions) and recursion patterns (starting from the Y-combinator, `cons`, `if`, etc.) **(A)** Learning a language for physical laws starting with recursive list routines such as `map` and `fold`. DreamCoder observes numerical 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), and more advanced primitives (e.g., `index`) that bring together the `fold` and `unfold` families.

basic programming tasks, like those used in introductory computer science classes. Crucially the initial language 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. 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. 7B). All these library functions 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" (*42*). DreamCoder retraced the discovery of origami programming: first reinventing fold, then unfold, and then defining all other recursive functions in terms of folding and unfolding.

Discussion

Our work shows that it is possible and practical to build a single general-purpose program induction system that learns the expertise needed to represent and solve new learning tasks in many qualitatively different domains, and that improves its expertise with experience. Optimal expertise in DreamCoder hinges on learning explicit declarative knowledge together with the implicit procedural skill to use it. More generally, DreamCoder's ability to learn deep explicit representations of a domain's conceptual structure shows the power of combining symbolic, probabilistic and neural learning approaches: Hierarchical representation learning algorithms can create knowledge understandable to humans, in contrast to conventional deep learning with neural networks, yielding symbolic representations of expertise that flexibly adapt and grow with experience, in contrast to traditional AI expert systems.

We focused here on problems where the solution space is well captured by crisp symbolic forms, even in domains that admit other complexities such as pixel image inputs, or exceptions and irregularities in generative text patterns, or continuous parameters in our symbolic regression examples. Nonetheless, much real-world data is far messier. A key challenge for program induction going forward is to handle more pervasive noise and uncertainty, by leaning more heavily on probabilistic and neural AI approaches (*5, 43, 44*). Recent research has explored program induction with various hybrid neuro-symbolic representations (*45–49*), and integrating these approaches with the library learning and bootstrapping capacities of DreamCoder could be especially valuable going forward.

Scaling up program induction to the full AI landscape — to commonsense reasoning, natural language understanding, or causal inference, for instance — will demand much more innovation but holds great promise. As a substrate for learning, programs uniquely combine universal expressiveness, data-efficient generalization, and the potential for interpretable, compositional reuse. Now that we can start to learn not just individual programs, but whole domain-specific languages for programming, a further property takes on heightened importance: Programs represent knowledge in a way that is mutually understandable by both humans and machines. Recognizing that every AI system is in reality the joint product of human and machine intelligence, we see the toolkit presented here as helping to lay the foundation for a scaling path to AI that people and machines can truly build together.

In the rest of this discussion, we consider the broader implications of our work for building better models of human learning, and more human-like forms of machine learning.

Interfaces with biological learning

DreamCoder's wake-sleep mechanics draw inspiration from the Helmholtz machine, which is itself loosely inspired by human learning during sleep. DreamCoder adds the notion of a pair of interleaved sleep cycles, and intriguingly, biological sleep similarly comes in multiple stages. Fast-wave REM sleep, or dream sleep, is associated with learning processes that give rise to implicit procedural skill (*11*), and engages both episodic replay and dreaming, analogous to our model's dream sleep phase. Slow-wave sleep is associated with the formation and consolidation of new declarative abstractions (*10*), roughly mapping to our model's abstraction sleep phase. While neither DreamCoder nor the Helmholtz machine are intended as biological models, we speculate that our approach could bring wake-sleep learning algorithms closer to the actual learning processes that occur during human sleep.

DreamCoder's knowledge grows gradually, with dynamics related to but different from earlier developmental proposals for "curriculum learning" (*50*) and "starting small" (*51*). Instead of solving increasingly difficult tasks ordered by a human teacher (the "curriculum"), DreamCoder learns in a way that is arguably more like natural unsupervised exploration: It attempts to solve random samples of tasks, searching out to the boundary of its abilities during waking, and then pushing that boundary outward during its sleep cycles, bootstrapping solutions to harder tasks from concepts learned with easier ones. But humans learn in much more active ways: They can choose which tasks to solve, and even generate their own tasks, either as stepping stones towards harder unsolved problems or motivated by considerations like curiosity and aesthetics. Building agents that generate their own problems in these human-like ways is an important next step.

Our division of domain expertise into explicit declarative knowledge and implicit procedural skill is loosely inspired by dual-process models in cognitive science (*52, 53*) and the study of human expertise (*37, 38*). Human experts learn both declarative domain concepts that they can talk about in words – artists learn arcs, symmetries, and perspectives; physicists learn inner products, vector fields, and inverse square laws – as well procedural (and implicit) skill in deploying those concepts quickly to solve new problems. Together, these two kinds of knowledge let experts more faithfully classify problems based on the "deep structure" of their solutions (*37, 38*), and intuit which concepts are likely to be useful in solving a task even before they start searching for a solution. We believe both kinds of expertise are necessary ingredients in learning systems, both biological and artificial, and see neural and symbolic approaches playing complementary roles here.

What to build in, and how to learn the rest

The goal of learning like a human—in particular, a human child—is often equated with the goal of learning "from scratch", by researchers who presume, following Turing (*1*), that children start off close to a blank slate: "something like a notebook as one buys it from the stationers. Rather little mechanism and lots of blank sheets." The roots of program induction as an approach to general AI also lie in this vision, motivated by early results showing that in principle, from only a minimal Turing-complete language, it is possible to induce programs that solve any problem with a computable answer (*2, 54–56*). DreamCoder's ability to start from minimal bases and discover the vocabularies of functional programming, vector algebra, and physics could be seen as another step towards that goal. Could this approach be extended to learn not just one domain at a time, but to simultaneously develop expertise across many different classes of problems, starting from only a single minimal basis? Progress could be enabled by metalearning a cross-domain library or "language of thought" (*57, 58*), as humans have built collectively through biological and cultural evolution, which can then differentiate

itself into representations for unboundedly many new domains of problems.

While these avenues would be fascinating to explore, trying to learn so much starting from so little is unlikely to be our best route to AI – especially when we have the shoulders of so many giants to stand on. Even if learning from scratch is possible in principle, such approaches suffer from a notorious thirst for data–as in neural networks–or, if not data, then massive compute: Just to construct 'origami' functional programming, DreamCoder took approximately a year of total CPU time. Instead, we draw inspiration from the sketching approach to program synthesis (*22*). Sketching approaches consider single synthesis problems in isolation, and expect a human engineer to outline the skeleton of a solution. Analogously, here we built in what we know constitutes useful ingredients for learning to solve synthesis tasks in many different domains – relatively spartan but generically powerful sets of control flow operators, higher-order functions, and types. We then used learning to grow specialized languages atop these foundations. The future of learning in program synthesis may lie with systems initialized with even richer yet broadly applicable resources, such as those embodied by simulation engines or by the standard libraries of modern programming languages.

This vision also shapes how we see program induction best contributing to the goal of building more human-like AI – not in terms of blank-slate learning, but learning on top of rich systems of built-in knowledge. Prior to learning the domains we consider here, human children begin life with "core knowledge": conceptual systems for representing and reasoning about objects, agents, space, and other commonsense notions (*59–61*). We strongly endorse approaches to AI that aim to build human-understandable knowledge, beginning with the kinds of conceptual resources that humans do. This may be our best route to growing artificial intelligence that lives in a human world, alongside and synergistically with human intelligence.

References

1. Alan M Turing. Computing machinery and intelligence. *Mind*, 1950.

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

3. Percy Liang, Michael I. Jordan, and Dan Klein. Learning dependency-based compositional semantics. In *ACL*, pages 590–599, 2011.

4. Tejas D Kulkarni, Pushmeet Kohli, Joshua B Tenenbaum, and Vikash Mansinghka. Picture: A probabilistic programming language for scene perception. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4390–4399, 2015.

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

6. Nick Chater and Mike Oaksford. Programs as causal models: Speculations on mental programs and mental representation. *Cognitive Science*, 37(6):1171–1191, 2013.

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

8. Miguel Lázaro-Gredilla, Dianhuan Lin, J Swaroop Guntupalli, and Dileep George. Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs. *Science Robotics*, 4(26):eaav3150, 2019.

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

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

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

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

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

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

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

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

17. Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Library learning for neurally-guided bayesian program induction. In *NeurIPS*, 2018.

18. Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

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

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

21. Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *ACM SIGPLAN Notices*, volume 44, pages 264–276. ACM, 2009.

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

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

24. Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438*, 2017.

25. M. M. Bongard. *Pattern Recognition*. Spartan Books, 1970.

26. Douglas Hofstadter and Gary McGraw. Letter spirit: An emergent model of the perception and creation of alphabetic style. 1993.

27. Jean Raven et al. Raven progressive matrices. In *Handbook of nonverbal assessment*, pages 223–237. Springer, 2003.

28. David D. Thornburg. Friends of the turtle. *Compute!*, March 1983.

29. Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. 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)*, pages 23–30. IEEE, 2017.

30. Patrick Winston. The MIT robot. *Machine Intelligence*, 1972.

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

32. Gary F Marcus, Sugumaran Vijayan, S Bandi Rao, and Peter M Vishton. Rule learning by seven-month-old infants. *Science*, 283(5398):77–80, 1999.

33. Brenden Lake, Chia-ying Lee, James Glass, and Josh Tenenbaum. One-shot learning of generative speech concepts. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 36, 2014.

34. Luke Hewitt and Joshua Tenenbaum. Learning structured generative models with memoised wake-sleep. *under review*, 2019.

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

36. Andrew Cropper. Playgol: Learning programs through play. *IJCAI*, 2019.

37. Michelene TH Chi, Paul J Feltovich, and Robert Glaser. Categorization and representation of physics problems by experts and novices. *Cognitive science*, 5(2), 1981.

38. M.T.H. Chi, R. Glaser, and M.J. Farr. *The Nature of Expertise*. Taylor & Francis Group, 1988.

39. Herbert A Simon, Patrick W Langley, and Gary L Bradshaw. Scientific discovery as problem solving. *Synthese*, 47(1):1–27, 1981.

40. Pat Langley. *Scientific discovery: Computational explorations of the creative processes*. MIT Press, 1987.

41. Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *science*, 324(5923):81–85, 2009.

42. Jeremy Gibbons. *Origami programming*. 2003.

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

44. Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471, 2016.

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

46. Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 39–48, 2016.

47. Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. In *Advances in Neural Information Processing Systems*, pages 3749–3759, 2018.

48. Halley Young, Osbert Bastani, and Mayur Naik. Learning neurosymbolic generative models via program synthesis. *arXiv preprint arXiv:1901.08565*, 2019.

49. Reuben Feinman and Brenden M. Lake. Generating new concepts with hybrid neuro-symbolic models, 2020.

50. Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *ICML*, 2009.

51. Jeffrey L Elman. Learning and development in neural networks: The importance of starting small. *Cognition*, 48(1):71–99, 1993.

52. Jonathan St BT Evans. Heuristic and analytic processes in reasoning. *British Journal of Psychology*, 75(4):451–468, 1984.

53. Daniel Kahneman. *Thinking, fast and slow*. Macmillan, 2011.

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

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

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

57. Jerry A Fodor. *The language of thought*, volume 5. Harvard University Press, 1975.

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

59. Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40, 2017.

60. Elizabeth S Spelke, Karen Breinlinger, Janet Macomber, and Kristen Jacobson. Origins of knowledge. *Psychological review*, 99(4):605, 1992.

61. Susan Carey. The origin of concepts: A précis. *The Behavioral and brain sciences*, 34(3):113, 2011.

Supplement. Supplementary materials available at `https://web.mit.edu/ellisk/www/dreamcodersupplement.pdf`. **Acknowledgments.** We thank L. Schulz, J. Andreas, T. Kulkarni, M. Kleiman-Weiner, J. M. Tenenbaum, M. Bernstein,, and E. Spelke for comments and suggestions that greatly improved the manuscript. Supported by grants from the Air Force Office of Scientific Research, the Army Research Office, the National Science Foundation-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.