PHILOSOPHICAL TRANSACTIONS A

royalsocietypublishing.org/journal/rsta

Research

Cite this article: Ellis K, Wong L, Nye M, Sablé-Meyer M, Cary L, Anaya Pozo L, Hewitt L, Solar-Lezama A, Tenenbaum JB. 2023 DreamCoder: growing generalizable, interpretable knowledge with wake–sleep Bayesian program learning. *Phil. Trans. R. Soc. A* **381**: 20220050. https://doi.org/10.1098/rsta.2022.0050

Received: 31 January 2023
Accepted: 23 March 2023

One contribution of 11 to a discussion meeting issue 'Cognitive artificial intelligence'.

Subject Areas:
artificial intelligence

Keywords:
program synthesis, expertise, Bayesian program learning

Author for correspondence:
Kevin Ellis
e-mail: kellis@cornell.edu

DreamCoder: growing generalizable, interpretable knowledge with wake–sleep Bayesian program learning

Kevin Ellis[1], Lionel Wong[2], Maxwell Nye[2], Mathias Sablé-Meyer[3], Luc Cary[2], Lore Anaya Pozo[2], Luke Hewitt[2], Armando Solar-Lezama[2] and Joshua B. Tenenbaum[2]

[1]Cornell (work done at MIT), Ithaca, NY, USA
[2]MIT, Cambridge, MA, USA
[3]PSL/Collége de France (work done at MIT), Paris, France

KE, 0000-0001-6586-0632; LAP, 0000-0001-5456-1178

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 domain-specific 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 multilayered symbolic representations that are interpretable and transferrable to new tasks, while still growing scalably and flexibly with experience.

This article is part of a discussion meeting issue 'Cognitive artificial intelligence'.

1. Introduction

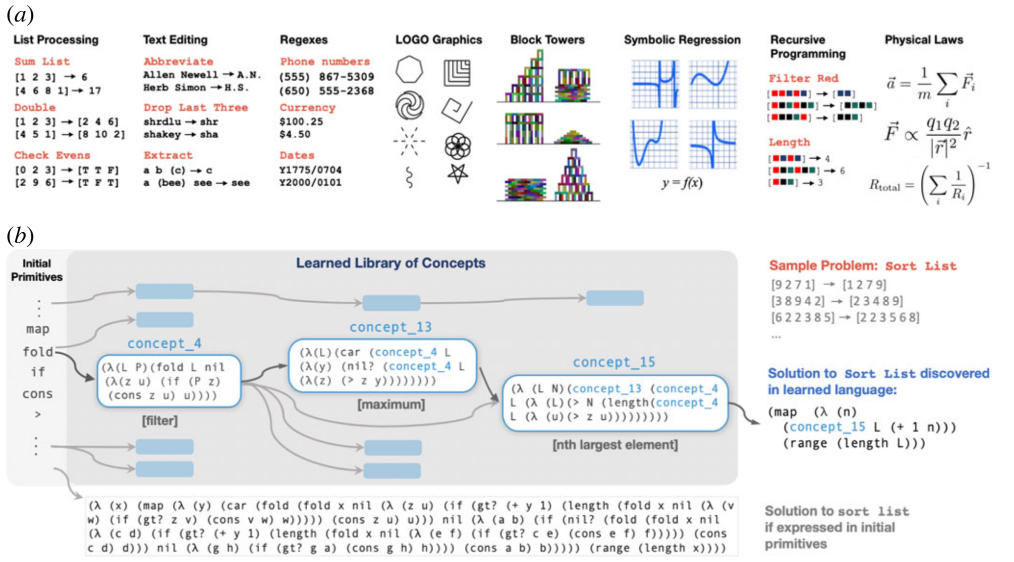
A longstanding dream in artificial intelligence (AI) has been to build a machine that learns like a child [1]—that grows into all the knowledge a human adult does, starting from much less. This dream remains far off, as human intelligence rests on many learning capacities not yet captured in artificial 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 overviews 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. Having previously presented the mechanics of the DreamCoder algorithm in an earlier paper [2], we focus here on giving a conceptual overview of the system and on its broader implications for cognitively informed AI.

Our formulation of learning as program induction traces back to the earliest days of AI [3]: we treat learning a new task as a search for a program that solves it, or which has some intended behaviour. Figure 1*a* shows examples of program induction tasks in eight different domains that DreamCoder has been applied to, along with an in-depth illustration of one task in the classic list processing domain: learning a program that sorts lists of numbers (figure 1*b*), 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 modelling 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 [4–11], 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



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

```
(λ(L)(car (concept_4 L
(λ(y) (nil? (concept_4 L
(λ(z) (> z y))))))))
```

```
(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*) 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 the list processing domain. 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 min 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. Illustration from Ellis *et al.* [2].

domain expertise: (i) explicit declarative knowledge, in the form of a learned DSL, capturing conceptual abstractions common across tasks in a domain and (ii) 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 solve a new domain of problems, given sufficiently varied in-domain tasks. Typically only a moderate number of tasks suffices to bootstrap learning in a new domain. For example, the list sorting function in figure 1*b* 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 new tasks.

DreamCoder's learned languages take the form of multilayered hierarchies of abstraction (figures 1 and 7*a*,*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 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 (figure 1*b*), 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 learner's 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 [12,13]. In general, wake–sleep Bayesian learning [14] 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 datasets ('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 them, 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, library learning [6,15,16] and neurally guided program synthesis [17,18], which have been separately influential in the recent literature but have only been brought together in our work starting with the EC^2 algorithm [19], and now made much more scalable in DreamCoder.

The resulting system has wide applicability. We describe applications to eight domains (figure 1*a*): 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. 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 artefact 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 interpretable, reusable knowledge in AI systems.

2. 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 (figure 2) cycling through a wake phase of trying to solve tasks interleaved with two sleep phases for learning to solve new tasks. Viewed as a probabilistic inference problem, DreamCoder observes a training set of tasks, written X, which we assume come from a single problem domain. It 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 that domain (figure 2 middle). This prior is encoded by a library, written L, which defines a generative model over programs, written $\mathrm{P}[\rho|L]$. 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 [14]. We write $Q(\rho|x)$ for the approximate posterior predicted by the recognition model. At a high level wake/sleep cycles correspond to iterating:

$$\begin{aligned}
\rho_x &= \underset{\substack{\rho: \\ Q(\rho|x) \text{ is large}}}{\arg\max}\ \mathrm{P}[\rho|x,L] \propto \mathrm{P}[x|\rho]\mathrm{P}[\rho|L], \text{ for each task } x \in X \qquad \text{Wake} \\
L &= \underset{L}{\arg\max}\, \mathrm{P}[L] \prod_{x\in X} \max_{\rho \text{ a refactoring of } \rho_x} \mathrm{P}[x|\rho]\mathrm{P}[\rho|L] \qquad \text{Sleep: Abstraction} \\
&\text{Train } Q(\rho|x) \approx \mathrm{P}[\rho|x,L], \text{ where } x \sim X \text{ (‘replay’) or } x \sim L \text{ (‘fantasy’)} \qquad \text{Sleep: Dreaming}
\end{aligned} \tag{2.1}$$

where $\mathrm{P}[L]$ is a description-length prior over libraries 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 three-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 solving tasks with fewer function calls, reducing search *depth*. The neural recognition model down-weights unlikely function calls, reducing the effective *breadth* of search, even as the number of library functions grows.

The abstraction sleep phase grows the 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 means preferring libraries that best compress programs found during waking, and the abstraction sleep objective (equation (2.1)) is equivalent to a compression objective. But rather than compress out reused syntactic structures, we refactor the programs to expose reused semantic patterns. As the number of possible refactorings grows combinatorially with program size, we needed a new data structure for representing and manipulating sets of refactorings, which we designed by combining ideas from version space algebras [20–22] and equivalence graphs [23] (described in our companion manuscript [2]). Recent work improved upon our original refactoring algorithm by making it more expressive [24] as well as orders of magnitude faster [25].

The dreaming sleep phase trains the 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 is biased 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 learning a library, we can draw unlimited samples or ‘dreams’ to train the recognition network.

3. 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

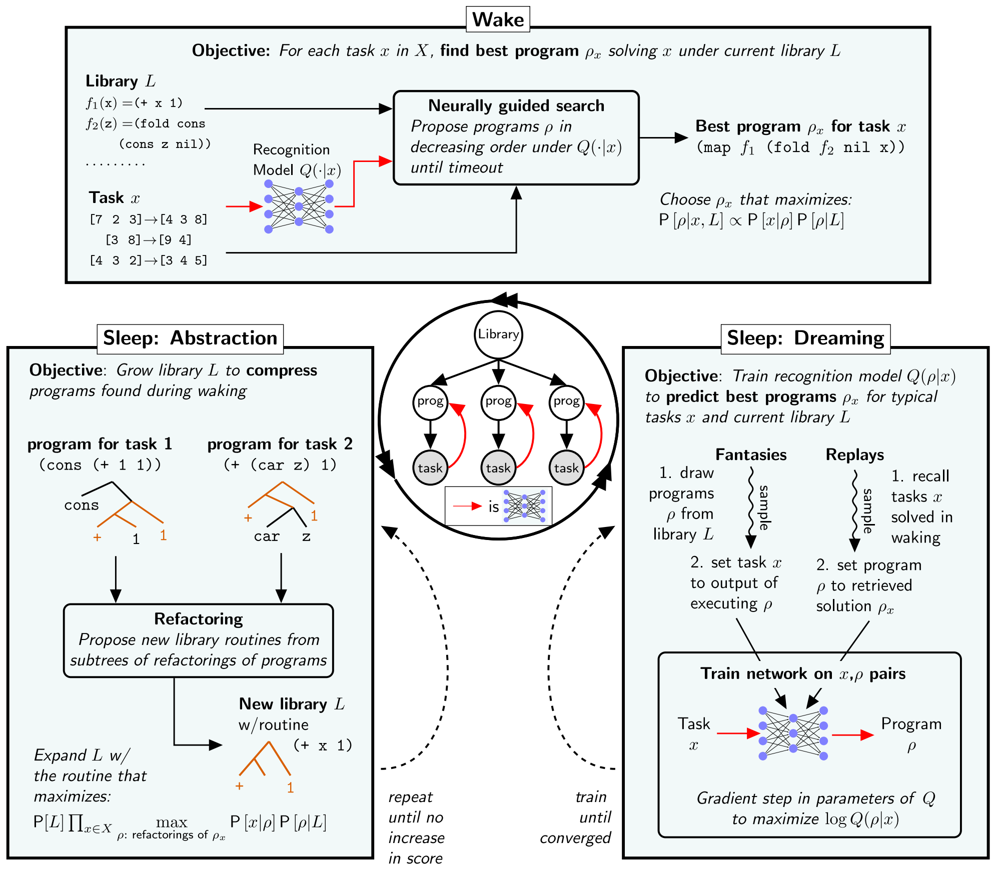

Wake
Objective: For each task x in X, find best program ρ_x solving x under current library L
Library L
f_1(x) =(+ x 1)
f_2(z) =(fold cons (cons z nil))
Task x
[7 2 3]→[4 3 8]
[3 8]→[9 4]
[4 3 2]→[3 4 5]
Recognition Model $Q(\cdot|x)$
Neurally guided search
Propose programs ρ in decreasing order under $Q(\cdot|x)$ until timeout
Best program ρ_x for task x
(map f_1 (fold f_2 nil x))
Choose ρ_x that maximizes:
$\mathrm{P}[\rho|x,L] \propto \mathrm{P}[x|\rho]\,\mathrm{P}[\rho|L]$
Library
prog
task
is
Sleep: Abstraction
Objective: Grow library L to compress programs found during waking
program for task 1
(cons (+ 1 1))
program for task 2
(+ (car z) 1)
cons
+
1
car
z
Refactoring
Propose new library routines from subtrees of refactorings of programs
New library L w/routine
(+ x 1)
Expand L w/ the routine that maximizes:
$\mathrm{P}[L]\prod_{x\in X}\max_{\rho:\text{ refactorings of }\rho_x}\mathrm{P}[x|\rho]\,\mathrm{P}[\rho|L]$
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 library L
Fantasies
1. draw programs ρ from library L
sample
2. set task x to output of executing ρ
Replays
1. recall tasks x solved in waking
sample
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 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). Illustration from [2].

(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 Ellis *et al.* [19], split 50/50 test/train, each with 15 input/output examples. In solving these problems, DreamCoder composed around 20 new library routines, 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 (figure 1*b*).

Synthesizing programs that edit text is a classic problem in the programming languages and AI literatures [20], and algorithms that synthesize text editing programs ship in Microsoft Excel

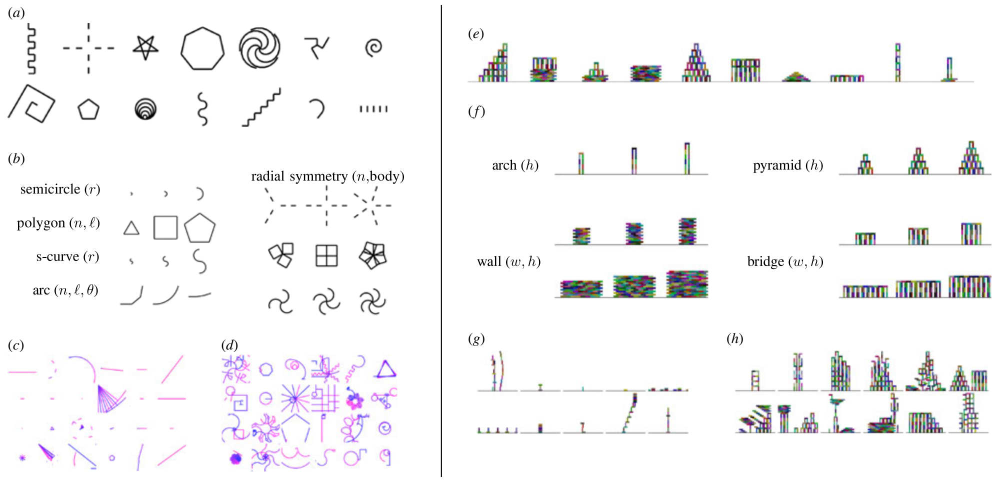


Figure 3. (*a*) 30 (out of 160) LOGO graphics tasks. The model writes programs controlling a 'pen' that draws the target picture. (*b*) Example learned library routines include both parametric routines for drawing families of curves as well as primitives that take entire programs as input. (*c,d*) Dreams, or programs sampled by randomly assembling functions from the model's library, change dramatically over the course of learning reflecting learned expertise. Before learning (*c*) dreams can use only a few simple drawing routines and are largely unstructured; the majority are simple line segments. After 20 iterations of wake–sleep learning (*d*) 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 illustrate the most interesting dreams found across five runs, both before and after learning. (*e–h*) Illustrate the tower building domain and are analogous to (*a–d*).

[8]. These systems would, for example, see the mapping 'Alan Turing' → '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 by training our system on 128 automatically generated text editing tasks; we then test on the 108 text editing problems from the 2017 SyGuS [26] program synthesis competition.[1] Prior to learning, DreamCoder solves 3.7% of the problems within 10 min with an average search time of 235 s. After learning, it solves 79.6%, and does so much faster, solving them in an average of 40 s.

We next consider more creative problems: generating images, plans and text. Procedural or generative visual concepts—from Bongard problems [27], to handwritten characters [6,28], to Raven's progressive matrices [29]—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 [30], tasking our model with drawing a corpus of 160 images (split 50/50 test/train; figure 3*a*) 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 and found interpretable parametric drawing routines corresponding to the families of visual objects in its training data, like polygons, circles and spirals (figure 3*b*)—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.

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

[1]We compare with the 2017 benchmarks because 2018 onward introduced non-string manipulation problems, hence the dataset no longer comprises a single domain, as DreamCoder assumes. The latest custom SyGuS solvers are at ceiling for these newest problems.

Input		MAP program	Samples	Input		MAP program	Samples
`(210)` `(220)` `(41)` `(635)` `(38)`	Full	`(dd(d)*)`	`(220)` `(461)`	`Y2015/1093` `Y2013/1010` `Y2014/1017` `Y2015/1421` `Y2017/1162`	Full	`Y201d/dddd`	`Y2010/3308` `Y2010/1163`
	No Library	`(dd(d)*.`	`(14u` `(2040)`		No Library	`Y201d/dddd`	`Y2011/1131` `Y2015/7116`
	No Rec	`(dd(d)*)`	`(68)` `(308)`		No Rec	`Y201(d)*d/(d)*`	`Y20127/20411` `Y2011214/1`
`$5.70` `$3.40` `$2.80` `$5.40` `$3.70`	Full	`$d.d0`	`$2.40` `$3.30`	`-00:16:05.9` `-00:19:52.9` `-00:33:24.7` `-00:44:02.3` `-00:24:25.0`	Full	`-00:dd.dd.d`	`-00:93:53.2` `-00:23=43.3`
	No Library	`$d.d0`	`$5=50` `$7#40`		No Library	`-00:dd.dd.d`	`-00:16g22:5` `-00:22.53\t2`
	No Rec	`$(d)*.(d)*0`	`$.0` `$873.30`		No Rec	`(-00:)?(.)*dd.d`	`-00:i47.5` `-00:r59.0`
`(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`	`L - ??` `L - 31.0 lbs.` `L - 10.0 lbs.` `S - 8.6 lbs.` `L - 25.2 lbs.`	Full	`u - (dd.(d)* )*(.)*`	`L - 13.05 ssb\t` `L - 12.3 02.1 s`
	No Library	`.ddd) ddd.dddd`	`?773) 726-6866` `m627) 674,0602`		No Library	`. - (d(.)*)*..`	`. -  . tY` `- E5`
	No Rec	`.(d)*) (d)*dd.(d)*`	`z40192) 51(8` `=2) 279-876273`		No Rec	`u - (d)*(.)*`	`L - 5208s.` `S - 5533..`

Figure 4. Results on held-out text generation tasks. System observes five 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).

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 (figure 3*c*), offering limited value for training the recognition model. After learning, the system's dreams are richly structured (figure 3*d*), 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 [31].

Inspired by the classic AI 'copy demo'—where an agent looks at a tower made of toy blocks then re-creates it [32]—we next gave DreamCoder 107 tower 'copy tasks' (split 50/50 test/train, figure 3*e*), 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' [33] for planning (figure 3*f*), including arches, staircases and bridges, which also appear in the model's dreams (figure 3*g*,*h*).

Next we consider few-shot learning of probabilistic generative concepts, an ability that comes naturally to humans, from learning new rules in natural language [34], to learning routines for symbols and signs [6], to learning new motor routines for producing words. We first task DreamCoder with inferring a probabilistic regular expression (or Regex, see figure 1*a* for examples) from a small number of strings, where these strings are drawn from 256 CSV columns crawled from the web (data from Hewitt *et al.* [35], tasks split 50/50 test/train, five 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 (figure 4). 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 (figure 1*a*, '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, hence our Bayesian machinery penalizes spurious use of these continuous parameters. DreamCoder 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 $2.3/(x - 2.8)$ it infers a program with two continuous parameters.

(i) 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* [16], which alternately searches for programs, and then compresses out reused components into a learned library, but without our refactoring algorithm; *Neural Program Synthesis*, which trains a RobustFill [18] model on samples from the initial library; and *Enumeration*, which enumerates well-typed programs for 24 h per task, generating and testing up to 400 million programs for each task.

Across domains, our model always solves the most held-out tasks (figure 5*a*) and generally solves them in the least time (mean 54.1 s; median 15.0 s). 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 figure 5*a* 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 (figure 5*b*). Similar but weaker relationships hold between performance and total library size. 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 (as measured by Representational Similarity Analysis [36]; $p < 10^{-4}$ 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 (figure 6)—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].

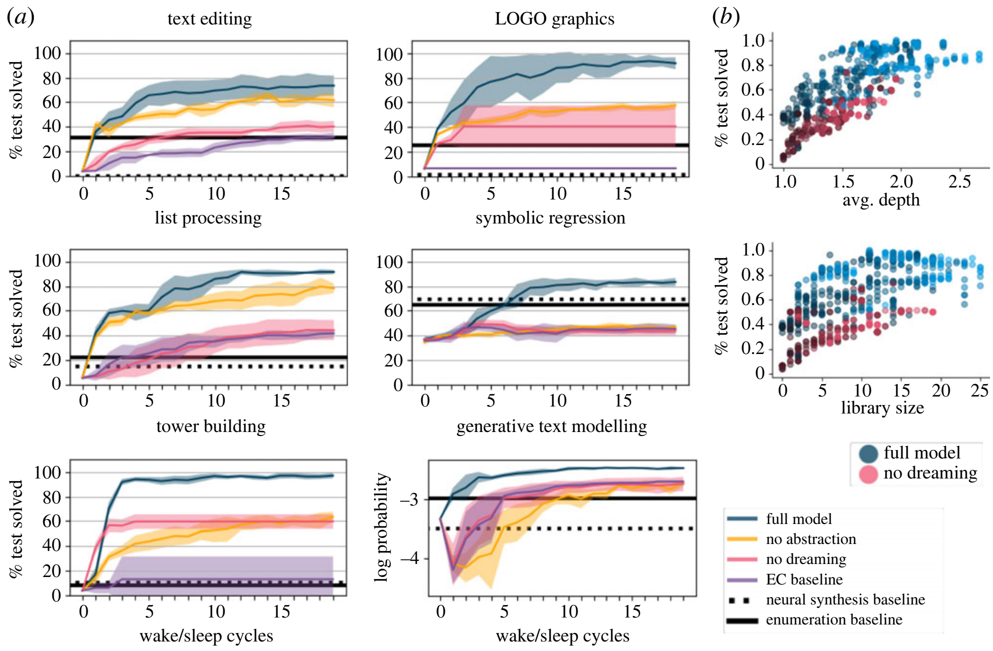


Figure 5. Comparing DreamCoder performance against ablations and baseline program induction methods; further baselines shown in [2]. (*a*) Hold-out accuracy, across 20 iterations of wake/sleep learning. Generative text modelling plots show posterior predictive likelihood of held-out strings on held-out tasks, normalized per-character. Error bars: ±1 s.d. 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 improves performance.

(ii) 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 generic programming and arithmetic primitives, and grow a DSL 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 [39,40], 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 eight 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 (figure 7*a*). 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

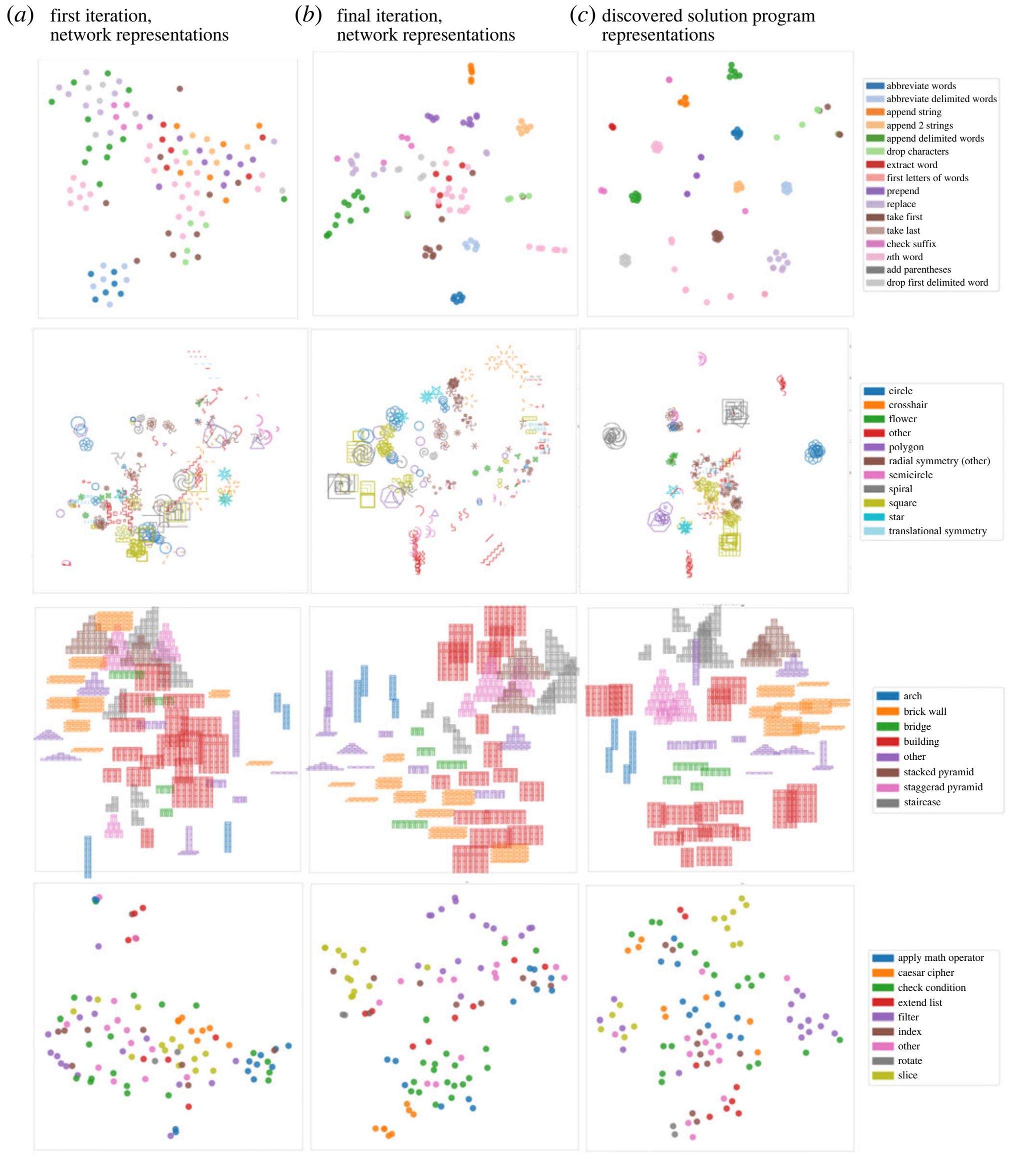
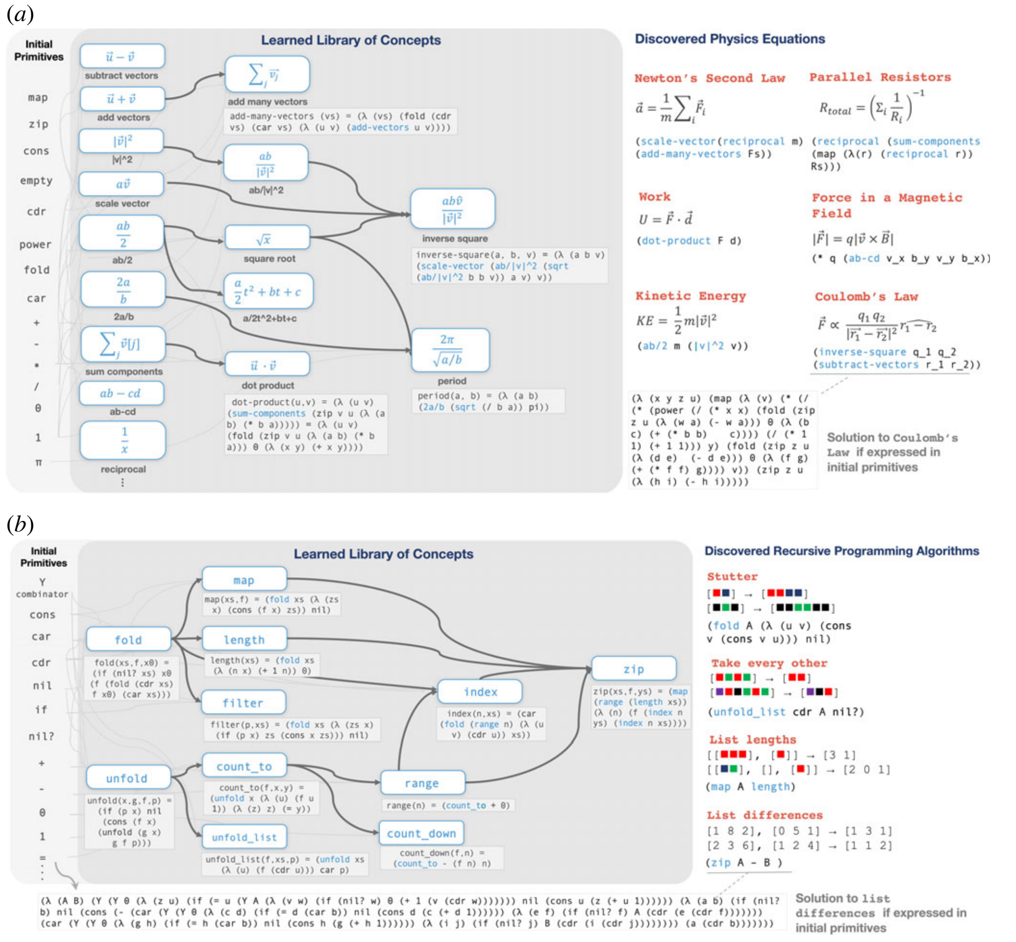


Figure 6. (*a*) TSNE embedding of model's 'at-a-glance' (before searching for solution) organization of tasks (neural net activations), after first wake/sleep cycle. (*b*) at-a-glance organization at last wake/sleep cycle. (*c*) TSNE visualization of programs actually used to solve tasks; intuitively, how model organizes tasks after searching for a solution. We convert each program into a feature vector by counting the uses of each library routine, and then embed those feature vectors.

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 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 expressing 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



```
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))
```

Newton's Second Law

$\vec{a}=\frac{1}{m}\sum_i \vec{F}_i$

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

Parallel Resistors

$R_{total}=\left(\Sigma_i \frac{1}{R_i}\right)^{-1}$

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

Work

$U=\vec{F}\cdot\vec{d}$

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

Force in a Magnetic Field

$|\vec{F}|=q|\vec{v}\times\vec{B}|$

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

Kinetic Energy

$KE=\frac{1}{2}m|\vec{v}|^2$

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

Coulomb's Law

$\vec{F}\propto\frac{q_1 q_2}{|\vec{r_1}-\vec{r_2}|^2}\widehat{r_1-r_2}$

```
(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)
```

Stutter

[🟥🟦] → [🟥🟥🟦🟦]

[⬛🟩⬛] → [⬛⬛🟩🟩⬛⬛]

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

Take every other

[🟥🟩🟥🟩] → [🟥🟥]

[🟪🟥⬛🟩🟥🟩] → [🟪⬛🟥]

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

List lengths

[[🟥🟥🟥], [🟥]] → [3 1]

[[🟦🟩], [], [🟥]] → [2 0 1]

```
(map A length)
```

List differences

```
[1 8 2], [0 5 1] → [1 3 1]
[2 3 6], [1 2 4] → [1 1 2]
(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 7. DreamCoder develops languages for physical laws (starting from recursive functions) and recursion patterns (starting from Y combinator, `cons`, `if`, etc.) (*a*) Learning a language for physical laws starting with recursive list routines such as `map` and `fold`. DreamCoder observes 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 represented as lists of numbers. Physical constants 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 unite the `fold` and `unfold` families. From Ellis *et al.* [2].

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 figure 7*b*). 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' [41]. 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.

4. 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 whose 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. But much real-world data is far messier. Going forward, a key challenge for program induction is to handle more pervasive noise and uncertainty by leaning more heavily on probabilistic and neural AI approaches [6,42,43]. Recent research has explored program induction with various hybrid neuro-symbolic representations [44–48], and integrating them with our library learning and bootstrapping capacities could be especially valuable.

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 DSLs 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.

(a) 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 [13], 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 [12], 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' [49] and 'starting small' [50]. 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. Systems 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 [51,52] 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.

DreamCoder and related systems have proved useful as platforms for building computational models of human learning [53–58], especially for understanding how human learners acquire interrelated systems of concepts. The ability of DreamCoder to learn drawing routines, and especially to quickly adapt its drawing style to new visual motifs, was used in Sablé-Meyer *et al.* [56] and Tian *et al.* [54] to model human participants when generating and comprehending novel visual stimuli. Those works show that a carefully crafted initial library is important to modelling human behaviour, in addition to learning the library: one needs to encode the right basic starter knowledge for geometry. The work in [54] also suggests that a bias toward efficient drawing procedures informs how humans interpret images, in addition to the parsimony bias that DreamCoder implements out-of-the-box. Yet another direction is to combine DreamCoder with natural language. Humans often learn a domain of concepts alongside the natural language used to describe that domain, a learning setup modelled in Wong *et al.* [53], who also show how to incorporate a well-known human bias into the joint language–program inference problem [59]. Last, Kumar *et al.* [55] show that a neural network's inductive bias can be better aligned with human biases when solving a task if the neural network is asked to also predict the DreamCoder program which solves that task, but this effect is strongest only after library learning has resculpted the space of possible programs.

(b) 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, as articulated by 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.'[2] 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 [3,60]. 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' [61,62], 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 [63]. 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

[2]In [1], Turing then goes on to advocate for sophisticated learning mechanisms.

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 [64–66]. 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.

Data accessibility. We have packaged our code and data in a publicly available artifact: https://dl.acm.org/do/10.1145/3410302/full/.

Authors' contributions. K.E.: conceptualization, data curation, investigation, methodology, software, supervision, writing—original draft, writing—review and editing; L.W.: conceptualization, data curation, formal analysis, investigation, methodology, software, validation, visualization, writing—original draft; M.N.: data curation, methodology, visualization; M.S.-M: conceptualization, investigation, methodology, software, visualization; L.A.P.: conceptualization, investigation, resources, software, validation, writing—original draft; L.H.: conceptualization, resources, software, visualization; L.C.: data curation, software, validation, visualization; A.S.-L.: conceptualization, funding acquisition, project administration, resources, supervision; J.B.T.: conceptualization, investigation, methodology, project administration, resources, supervision, writing—original draft, writing—review and editing.

All authors gave final approval for publication and agreed to be held accountable for the work performed therein.

Conflict of interest declaration. We declare we have no competing interests.

Funding. This work was supported by 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.

Acknowledgements. We thank L. Schulz, S. Tenka, J. Andreas, T. Kulkarni, M. Kleiman-Weiner, J. M. Tenenbaum, M. Bernstein and E. Spelke for comments and suggestions that greatly improved the manuscript.

References

1. Turing AM. 1950 Computing machinery and intelligence. *Mind*. **236**, 433–460.
2. Ellis K, Wong C, Nye M, Sablé-Meyer M, Morales L, Hewitt L, Cary L, Solar-Lezama A, Tenenbaum JB. 2021 Dreamcoder: bootstrapping inductive program synthesis with wake–sleep library learning. In *PLDI* **42**, 835–850.
3. Solomonoff RJ. 1964 A formal theory of inductive inference. *Inf. Control* **7**, 1–22. (doi:10.1016/S0019-9958(64)90223-2)
4. Liang P, Jordan MI, Klein D. 2011 Learning dependency-based compositional semantics. *ACL* **49**, 590–599.
5. Kulkarni TD, Kohli P, Tenenbaum JB, Mansinghka V. 2015 Picture: a probabilistic programming language for scene perception. In *Proc. 2015 IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, 07–12 June 2015*, pp. 4390–4399. New York, NY: IEEE.
6. Lake BM, Salakhutdinov R, Tenenbaum JB. 2015 Human-level concept learning through probabilistic program induction. *Science* **350**, 1332–1338. (doi:10.1126/science.aab3050)
7. Chater N, Oaksford M. 2013 Programs as causal models: speculations on mental programs and mental representation. *Cogn. Sci.* **37**, 1171–1191. (doi:10.1111/cogs.12062)
8. Gulwani S. 2011 Automating string processing in spreadsheets using input-output examples. In *POPL '11: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Austin, TX, 26–28 January 2011*, pp. 317–330. New York, NY: ACM.
9. Lázaro-Gredilla M, Lin D, Guntupalli JS, George D. 2019 Beyond imitation: zero-shot task transfer on robots by learning concepts as cognitive programs. *Sci. Rob.* **4**, eaav3150. (doi:10.1126/scirobotics.aav3150)

10. Devlin J, Bunel RR, Singh R, Hausknecht M, Kohli P. 2017 Neural program meta-induction. *NIPS* **31,** 2077–2085.
11. King RD, Whelan KE, Jones FM, Reiser PGK, Bryant CH, Muggleton SH, Kell DB, Oliver SG. 2004 Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature* **427**, 247–252. (doi:10.1038/nature02236)
12. Dudai Y, Karni A, Born J. 2015 The consolidation and transformation of memory. *Neuron* **88**, 20–32. (doi:10.1016/j.neuron.2015.09.004)
13. Fosse MJ, Fosse R, Hobson JA, Stickgold RJ. 2003 Dreaming and episodic memory: a functional dissociation? *J. Cogn. Neurosci.* **15**, 1–9. (doi:10.1162/089892903321107774)
14. Hinton GE, Dayan P, Frey BJ, Neal RM. 1995 The 'wake–sleep' algorithm for unsupervised neural networks. *Science* **268**, 1158–1161. (doi:10.1126/science.7761831)
15. Dumancic S, Guns T, Cropper A. 2021 Knowledge refactoring for inductive program synthesis. *Proc. AAAI Conf. on Artif. Intell.* **35,** 7271–7278. (doi:10.1609/aaai.v35i8.16893)
16. Dechter E, Malmaud J, Adams RP, Tenenbaum JB. 2013 Bootstrap learning via modular concept discovery. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, Beijing, China, 3–9 August 2013*, pp. 1302–1309. Washington, DC: AAAI Press.
17. Balog M, Gaunt AL, Brockschmidt M, Nowozin S, Tarlow D. 2016 Deepcoder: learning to write programs. *ICLR 2017, Toulouse, France, 24–26 April 2016*. (doi:10.48550/arXiv.1611.01989)
18. Devlin J, Uesato J, Bhupatiraju S, Singh R, Mohamed A-R, Kohli P. 2017 Robustfill: neural program learning under noisy i/o. In *ICML'17: Proceedings of the 34th International Conference on Machine Learning*, Sydney, Australia, 6 August 2017, pp. 990-998. New York, NY: ACM.
19. Ellis K, Morales L, Sablé-Meyer M, Solar-Lezama A, Tenenbaum J. 2018 Library learning for neurally-guided bayesian program induction. In *NIPS'18: Proceedings of the 32nd International Conference on Neural Information Processing Systems, Montréal, Canada, 2–8 December 2018*, pp. 7816–7826. New York, NY: ACM.
20. Lau T, Wolfman SA, Domingos P, Weld DS. 2003 Programming by demonstration using version space algebra. *Mach. Learn.* **53**, 111–156. (doi:10.1023/A:1025671410623)
21. Mitchell TM. 1977 Version spaces: a candidate elimination approach to rule learning. In *IJCAI'77: Proceedings of the 5th international joint conference on Artificial intelligence, Cambridge, MA, 22–25 August 1977*, pp. 305–310. Washington, DC: AAAI.
22. Polozov O, Gulwani S. 2015 Flashmeta: a framework for inductive program synthesis. *ACM SIGPLAN Not.* **50**, 107–126. (doi:10.1145/2858965.2814310)
23. Tate R, Stepp M, Tatlock Z, Lerner S. 2009 Equality saturation: a new approach to optimization. *ACM SIGPLAN Notices* **44,** 264–276.
24. Cao D, Kunkel R, Nandi C, Willsey M, Tatlock Z, Polikarpova N. 2023 Babble: learning better abstractions with e-graphs and anti-unification. *POPL* **7**, 396–424. (doi:10.1145/3571207)
25. Bowers M, Olausson TX, Wong L, Grand G, Tenenbaum JB, Ellis K, Solar-Lezama A. 2023 Top-down synthesis for library learning. *POPL* **7**, 1182–1213. (doi:10.1145/3571234)
26. Alur R, Fisman D, Singh R, Solar-Lezama A. Sygus-comp 2017: results and analysis. (http://arxiv.org/abs/1711.11438). 2017.
27. Bongard MM. 1970 *Pattern recognition*. London, UK: Spartan Books.
28. Hofstadter D 1996 Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought. New York, NY: HarperCollins.
29. Raven J. 2003 Raven progressive matrices. In *Handbook of nonverbal assessment* (ed. S McCallum), pp. 223–237. Cham, Switzerland: Springer.
30. Thornburg DD. 1983 Friends of the turtle. *Compute!*, March.
31. Tobin J, Fong R, Ray A, Schneider J, Zaremba W, Abbeel P. 2017 Domain randomization for transferring deep neural networks from simulation to the real world. *In Proceedings of the* 2017 *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Vancouver, 24–28 September 2017*, pp. 23–30. New York, NY: IEEE.
32. Winston P. 1972 The MIT robot. *Mach. Intell.* **7**.
33. Sutton RS, Precup D, Singh S. 1999 Between mdps and semi-mdps: a framework for temporal abstraction in reinforcement learning. *Artif. Intell.* **112**, 181–211. (doi:10.1016/S0004-3702(99)00052-1)
34. Marcus GF, Vijayan S, Rao SB, Vishton PM. 1999 Rule learning by seven-month-old infants. *Science* **283**, 77–80. (doi:10.1126/science.283.5398.77)

35. Hewitt L, Le TA, Tenenbaum J. 2020 Learning to learn generative programs with memoised wake–sleep. In *Uncertainty in Artificial Intelligence* PMLR volume 124. (http://proceedings.mlr.press/v124/hewitt20a/hewitt20a.pdf)
36. Kriegeskorte N, Mur M, Bandettini PA. 2008 Representational similarity analysis-connecting the branches of systems neuroscience. *Front. Syst. Neurosci.* **2**, 4. (doi:10.3389/neuro.06.004.2008)
37. Chi MTH, Feltovich PJ, Glaser R. 1981 Categorization and representation of physics problems by experts and novices. *Cogn. Sci.* **5**, 121–152. (doi:10.1207/s15516709cog0502_2)
38. Chi MTH, Glaser R, Farr MJ. 1988 *The nature of expertise*. London, UK: Taylor & Francis Group.
39. Langley P. 1987 *Scientific discovery: computational explorations of the creative processes*. Cambridge, MA: MIT Press.
40. Schmidt M, Lipson H. 2009 Distilling free-form natural laws from experimental data. *Science* **324**, 81–85. (doi:10.1126/science.1165893)
41. Gibbons J. 2003 Origami programming. In *The fun of programming* (eds J Gibbons, O de Moor). London, UK: Red Globe Press.
42. Ellis K, Ritchie D, Solar-Lezama A, Tenenbaum JB. 2018 Learning to infer graphics programs from hand-drawn images. In *NIPS'18: Proceedings of the 32nd International Conference on Neural Information Processing Systems, Montréal, Canada, 2–8 December 2018*, pp. 6062–6071. New York, NY: ACM.
43. Graves A *et al.* 2016 Hybrid computing using a neural network with dynamic external memory. *Nature* **538**, 471. (doi:10.1038/nature20101)
44. Valkov L, Chaudhari D, Srivastava A, Sutton C, Chaudhuri S. 2018 Houdini: lifelong learning as program synthesis. *NeurIPS* **31,** 8701–8712.
45. Andreas J, Rohrbach M, Darrell T, Klein D. 2016 Neural module networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, 27–30 June 2016*, pp. 39–48. New York, NY: IEEE.
46. Manhaeve R, Dumancic S, Kimmig A, Demeester T, Raedt LD. 2018 Deepproblog: neural probabilistic logic programming. In *NIPS'18: Proceedings of the 32nd International Conference on Neural Information Processing Systems, Montréal, Canada, 2–8 December 2018*. New York, NY: ACM.
47. Young H, Bastani O, Naik M. 2019 Learning neurosymbolic generative models via program synthesis. *ICML* **42,** 7144–7153.
48. Feinman R, Lake BM. 2020 Generating new concepts with hybrid neuro-symbolic models. In *Proceedings of the Annual Meeting of the Cognitive Science Society, Virtual, 29 July – 1 August 2020*. Seattle, WA: Cognitive Science Society.
49. Bengio Y, Louradour J, Collobert R, Weston J. 2009 Curriculum learning. In *ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning, Montreal, 14–18 June 2009*, pp. 41–48. New York, NY: ACM.
50. Elman JL. 1993 Learning and development in neural networks: the importance of starting small. *Cognition* **48**, 71–99. (doi:10.1016/0010-0277(93)90058-4)
51. Evans JSBT. 1984 Heuristic and analytic processes in reasoning. *Br. J. Psychol.* **75**, 451–468. (doi:10.1111/j.2044-8295.1984.tb01915.x)
52. Kahneman D. 2011 *Thinking, fast and slow*. New York, NY: Macmillan.
53. Wong C, Ellis K, Tenenbaum JB, Andreas J. 2021 Leveraging language to learn program abstractions and search heuristics. In *Proceedings of the 38th International Conference on Machine Learning, Virtual, 18–24 July 2021*, pp. 11 193–11 204. PMLR.
54. Tian L, Ellis K, Kryven M, Tenenbaum J. 2020 Learning abstract structure for drawing by efficient motor program induction. *NeurIPS* **33,** 2686–2697.
55. Kumar S *et al.* 2022 Using natural language and program abstractions to instill human inductive biases in machines. *NeurIPS* **35,** 167–180.
56. Sablé-Meyer M, Ellis K, Tenenbaum J, Dehaene S. 2022 A language of thought for the mental representation of geometric shapes. *Cognit. Psychol.* **139**, 101527.
57. Ellis K, Albright A, Solar-Lezama A, Tenenbaum JB, O'Donnell TJ. 2022 Synthesizing theories of human language with bayesian program induction. *Nat. Commun.* **13**, 5024. (doi:10.1038/s41467-022-32012-w)

58. Wong C, McCarthy WP, Grand G, Friedman Y, Tenenbaum JB, Andreas J, Hawkins RD, Fan JE. 2022 Identifying concept libraries from language about object structure. In *Proceedings of the Annual Meeting of the Cognitive Science society*. Preprint.
59. Markman EM, Wachtel GF. 1988 Children's use of mutual exclusivity to constrain the meanings of words. *Cognit. Psychol.* **20**, 121–157. (doi:10.1016/0010-0285(88)90017-5)
60. Hutter M. 2004 *Universal artificial intelligence: sequential decisions based on algorithmic probability*. Berlin, Heidelberg: Springer Science & Business Media.
61. Fodor JA. 1975 *The language of thought*, vol. 5. Cambridge, MA: Harvard University Press.
62. Piantadosi ST. 2011 *Learning and the language of thought*. PhD thesis, MIT.
63. Solar Lezama A. 2008 *Program synthesis by sketching*. PhD thesis.
64. Lake BM, Ullman TD, Tenenbaum JB, Gershman SJ. 2017 Building machines that learn and think like people. *Behav. Brain Sci.* **40**, e253. (doi:10.1017/S0140525X16001837)
65. Spelke ES, Breinlinger K, Macomber J, Jacobson K. 1992 Origins of knowledge. *Psychol. Rev.* **99**, 605. (doi:10.1037/0033-295X.99.4.605)
66. Carey S. 2011 The origin of concepts: a précis. *Behav. Brain Sci.* **34**, 113. (doi:10.1017/S0140525X10000919)