Draft in progress; subject to changes.

DreamCoder: Building interpretable hierarchical knowledge representations with wake-sleep Bayesian program learning

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

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 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 guiding the language's use. A new "wake-sleep" learning algorithm interleaves two sleep phases, alternately extending DreamCoder's language with new symbolic abstractions, and training its network on imagined and replayed problems. DreamCoder solves classic programming problems and creative tasks like drawing pictures and building scenes. It rediscovers basics of modern functional programming, vector algebra and classical physics, including Newton's and Coulomb's laws. DreamCoder grows its languages compositionally from concepts learned earlier, building multi-layered symbolic representations that are interpretable and transferrable to new tasks, while still growing scalably and flexibly with experience.

Human intelligence allows us to learn to solve an endless range of problems, from cooking to calculus to graphic design. Remarkably, humans require a relatively modest amount of experience to master any one new individual skill, at least by the standards of contemporary machine learning, and the knowledge they acquire from that experience can be communicated in ways that other humans can interpret and extend. These hallmarks of human learning — broad applicability, sample efficiency, and interpretable reuse of knowledge — are addressed in different ways across machine learning communities. Deep learning approaches are widely applicable, particularly for perception and pattern recognition; probabilistic frameworks yield sample-efficient learning, given good priors; and symbolic representations offer interpretable reuse of learned structures. Yet, in isolation, none of these paradigms alone offers a route toward machines that can efficiently discover a broad range of forms of knowledge.

A complementary approach, program induction, could in principle contribute to such a route. Program induction systems represent knowledge as programs–i.e., as source code–and learn via program synthesis. In broad strokes this approach traces back to the earliest days of AI (*1*). Program

induction complements other AI approaches, bringing three primary advantages. Programs exhibit strong generalization properties–intuitively, they tend to extrapolate rather than interpolate–and this strong generalization confers greater sample efficiency. Furthermore, high-level coding languages are human-understandable: Programs are the language by which the entire digital world is constructed, understood, and extended by human engineers. Yet one of the most powerful properties of programs is their universality, or Turing-completeness: in principle, programs can represent solutions to the full range of computational problems solvable by intelligence. Indeed, different kinds of program induction systems have succeeded for domains as varied as semantic parsing (*2*), computer vision (*3, 4*), cognitive modeling (*5*), programming-by-examples (*6*), and robot motion planning (*7*).

In spite of these desirable features and practical successes, program induction confronts two intertwined obstacles that have impeded its broad use within AI. First, one needs a high-level domain-specific programming language. Without the inductive bias imparted by a specialized language, the programs to be discovered would be prohibitively long. This length exponentially increases the computational demands of program synthesis, and these longer programs are much harder for humans to understand. Second, even given a domain-specific programming language, the problem of program synthesis remains intractable, because of the combinatorial nature of the search space. Essentially every practical application of program induction couples a custom domain-specific programming language to a custom search algorithm.

Here we present DreamCoder, an algorithm that seeks to address these two primary obstacles. DreamCoder aims to acquire the domain expertise needed to induce a new class of programs. We think of this learned domain expertise as belonging to one of two categories: declarative knowledge, embodied by a domain-specific programming language; and procedural skill, implemented by a learned domain-specific search strategy. This partitioning of domain expertise into explicit, declarative knowledge and implicit, procedural skill is loosely inspired by dual-process models in cognitive science (*8*), and the human expertise literature (*9, 10*). Human experts learn declarative (and explicit) concepts that are finely-tuned to their domain. Artists learn concepts like arcs, symmetries, and perspectives; and physicists learn concepts like inner products, vector fields, and inverse square laws. Human experts also acquire procedural (and implicit) skill in deploying those concepts quickly to solve new problems. Compared to novices, experts more faithfully classify problems based on the "deep structure" of their solutions (*9, 10*), intuiting which compositions of concepts are likely to solve a task even before searching for a solution.

Concretely, DreamCoder operates by growing out declarative knowledge in the form of a symbolic language for representing problem solutions. It builds procedural skill by training a neural network to guide the online use of this learned language during program synthesis. This learned language consists of a learned library of reusable functions, and acts as a inductive bias over the space of programs. The neural network learns to act as a domain-specific search strategy that probabilistically guides program synthesis. Our algorithm takes as input a modestly-sized corpus of program induction tasks, and it learns both its library and neural network in tandem with solving this corpus of tasks. This architecture integrates a pair of ideas, Bayesian multitask program learning (*4, 11, 12*), and neurally-guided program synthesis (*13, 14*). Both these ideas have been separately influential, but have only been brought together in our work starting with the EC^2 algorithm (*15*), a forerunner of DreamCoder (see S3 for discussion of prior work). DreamCoder tackles the challenges inherent in scaling this approach to larger and more general problem domains. New algorithms are needed for training neural networks to guide efficient program search, and for building libraries of reusable concepts by identifying and abstracting out program pieces not present in the surface forms of problem solutions — so programs must be "refactored" to syntactically expose their deep underlying semantic structure.

DreamCoder gets its name from the fact that it is a "wake-sleep" algorithm (*16*). Wake-sleep approaches iterate training a probabilistic *generative model* alongside training a *recognition model* that learns to invert this generative model. DreamCoder's learned library defines a generative model over programs, and its neural network learns to recognize patterns across tasks in order to predict programs solving those tasks. This neural recognition model is trained on both the actual tasks to be solved, as well as samples, or "dreams," from the learned library. A probabilistic Bayesian wake-sleep framing permits principled handling of uncertainty, noise, and continuous parameters, and serves to provide an objective function from which the mechanics of the algorithm are derived (S4.1).

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., 5-10 examples of a new concept or function, or a single image or scene depicting a new object. The learned languages span deterministic and probabilistic programs, and programs that act both generatively (e.g., producing an artifact like an image or plan) and conditionally (e.g., mapping inputs to outputs).

DreamCoder's library learning works by building multilayered hierarchies of abstractions. Like the internal representations in a deep neural network, these libraries (Fig. 1B, & Fig. 7A,B) consist of layers of learned abstractions, but here the hierarchical representation is built from symbolic code, which is easily interpretable and explainable by humans. Our model's network of abstractions grows progressively over time, inspired by how human learners build concepts on top of those learned previously: Children learn algebra before calculus, and only after arithmetic; and they draw two-dimensional caricatures before sketching three-dimensional scenes. DreamCoder likewise creates new library routines that build on concepts acquired earlier in its learning trajectory. For example, the model comes to sort sequences of numbers by invoking a library component four layers deep (Fig. 1B), and this component in turn calls lower-level concepts. 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. Equivalent programs expressed using the initial primitives are also so complex that they are effectively out of reach: they would never be found during a reasonably bounded search. Only with acquired domain expertise do problems like these become practically and usefully solvable.

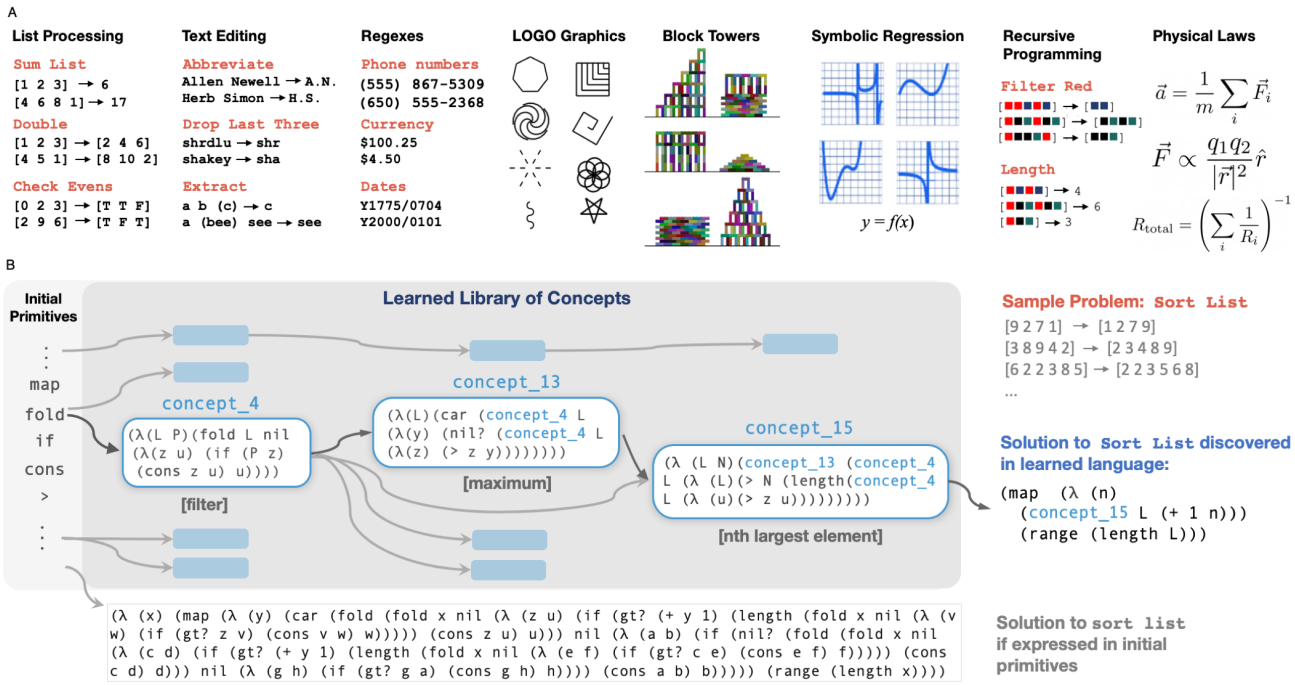


Figure 1: **(A)**: Eight problem-solving domains DreamCoder is applied to. **(B)**: An illustration of how DreamCoder learns to solve problems in one domain, processing lists of integers. Problems are specified by input-output pairs exemplifying a target function (e.g., 'Sort List'). Given initial primitives (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.

Wake/Sleep Program Learning

Learning in DreamCoder works through a novel kind of "wake-sleep" learning, inspired by but crucially different than the original wake-sleep algorithm of Hinton, Dayan and colleagues (*16*). Each DreamCoder iteration (Eq. 1, Fig. 2) comprises a wake cycle — where the model solves problems by writing programs — interleaved with two sleep cycles. The first sleep cycle, which we refer to as **abstraction**, grows the library of code (declarative knowledge) by replaying experiences from waking and consolidating them into new code abstractions (Fig. 2 left). This mechanism increases the breadth and depth of learned libraries like those in Fig. 1B and Fig. 7, when viewed as networks. The second sleep cycle, which we refer to as **dreaming**, improves the agent's procedural skill in code-writing by training a neural network to help quickly search for programs. The neural net is trained on replayed experiences as well as 'fantasies', or sampled programs, built from the learned library (Fig. 2 right).

Viewed as a probabilistic inference problem, DreamCoder observes a 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 finds programs solving a task by predicting, conditioned on 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 (*16*). 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).

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

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 program representation. Simultaneously the generative and recognition models bootstrap each other: A more finely tuned library of concepts 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]

Waking consists of searching for task-specific programs with high posterior probability, or programs which are a priori likely and which solve a task. During a Wake cycle we sample a minibatch of tasks and find programs solving a specific task by enumerating programs in decreasing order of their probability under the recognition model, then checking if a program ρ assigns positive probability to solving a task ($\mathrm{P}[x|\rho] > 0$). Because the model may 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.

During the abstraction phase of sleep, 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.

[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.

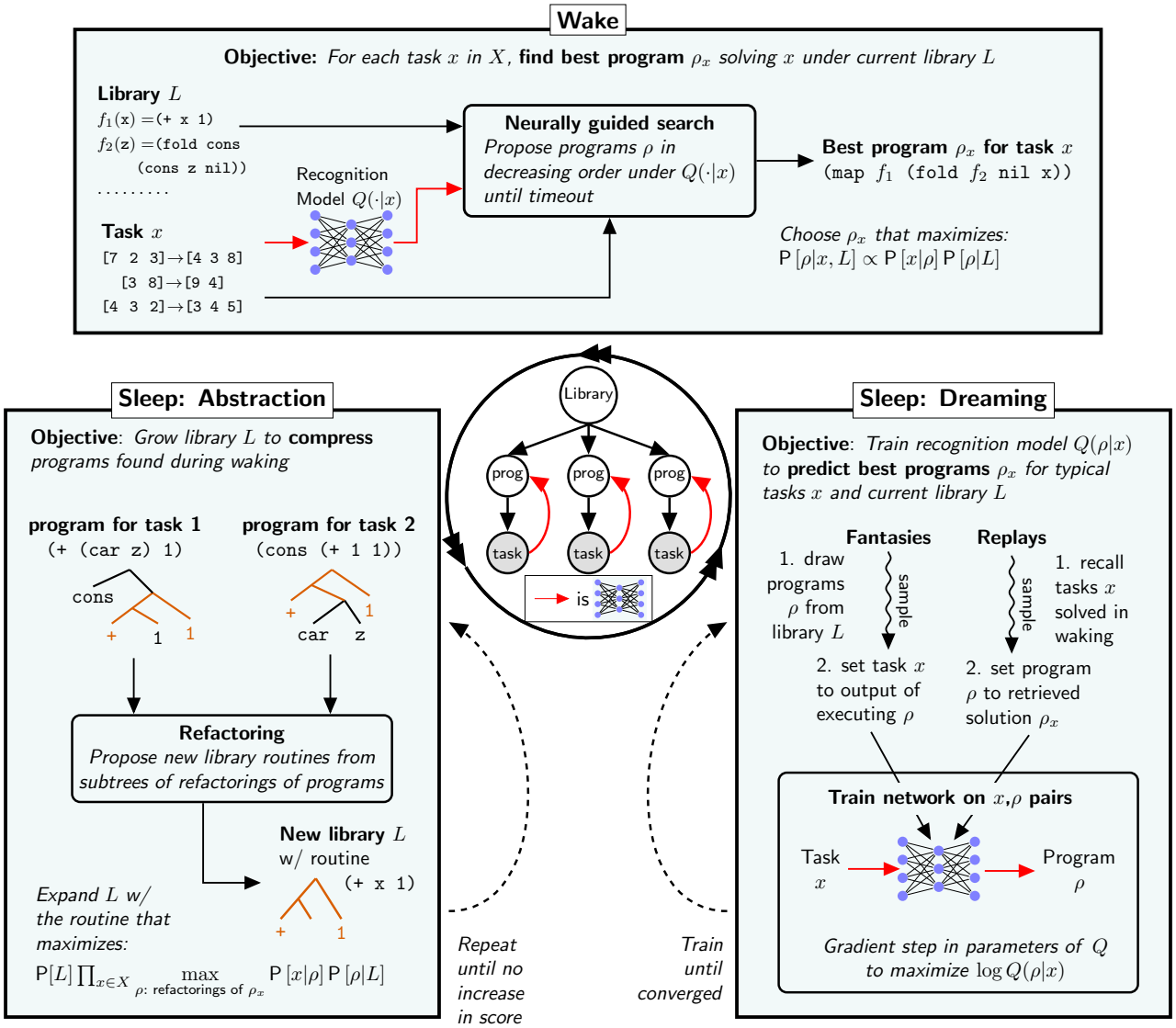


Figure 2: DreamCoder's basic algorithmic cycle, which serves to perform approximate Bayesian inference for the graphical model diagrammed in the **middle**. The system observes programming tasks (e.g., input/outputs for list processing or images for graphics programs), which it explains with latent programs, 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) & 'Replays' (programs found during waking).

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 (*17–19*) and equivalence graphs (*20*), 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.

During dreaming 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, a convolutional network imparts a bias toward generally useful image features, useful when inducing graphics programs from images. 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 nor 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 (*16*) dream phase: we think of dreaming as creating an endless stream of random problems, which we then solve during sleep, and train the recognition network to predict the solution conditioned on the problem. The classic wake-sleep algorithm would instead sample a random program, execute it to get a task, and train the recognition network to predict the sampled program from the sampled task. Specifically, we train Q to perform MAP inference by maximizing $\mathrm{E}\left[\log Q\left(\left(\arg\max_\rho \mathrm{P}[\rho|x,L]\right)|x\right)\right]$, where the expectation is taken over tasks. Taking this expectation over the empirical distribution of tasks trains Q on replays; taking it over samples from the generative model trains Q on fantasies. We train on a 50/50 mix of replays and fantasies; for fantasies mapping inputs to outputs, we randomly draw 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 of 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 (*21, 22*); see supplement S4.6 for theoretical and empirical analyses of DreamCoder's learned symmetry breaking.

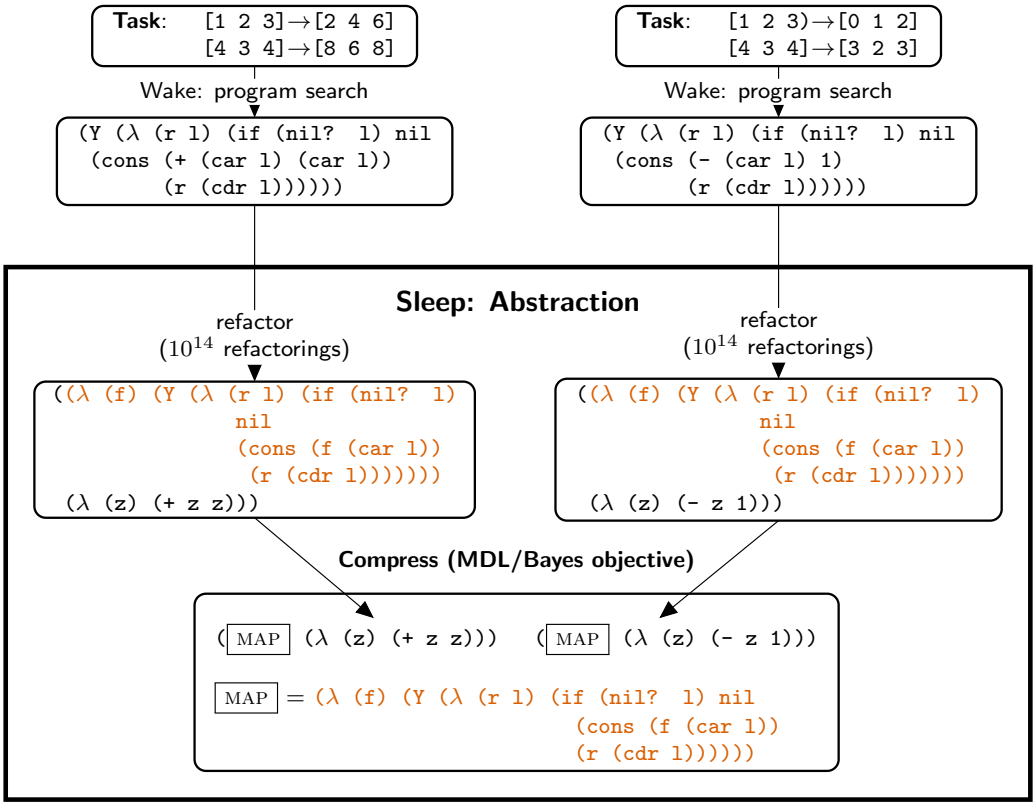


Figure 3: Programs found as solutions during waking are refactored – or rewritten in semantically equivalent but syntactically distinct forms – during the sleep abstraction phase, to expose candidate new primitives for growing DreamCoder's learned library. Here, solutions for two simple list tasks (top 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.

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 (*15*), 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 (*17*), and algorithms that synthesize text editing programs ship in Microsoft Excel (*6*). 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 the state-of-the-art domain-general program synthesizers. We trained our system on 128 automatically-generated text editing tasks, and tested on the 108 text editing problems from the 2017 SyGuS (*23*) 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 libraries 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 (*24*), to handwritten characters (*4, 25*), to Raven's progressive matrices (*26*) — 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 (*27*), 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 agent 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 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 (*28*).

Inspired by the classic AI 'copy demo' – where an agent looks at a tower made of toy blocks then re-creates it (*29*) – we next gave DreamCoder 107 tower 'copy tasks' (split 50/50 test/train, Fig. 5A), where the agent 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' (*30*) 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 (*31*), to learning routines for

[2]We compare with the 2017 benchmarks because 2018 onward introduced non-string manipulation problems; custom string solvers such as FlashFill (*6*) and the latest custom SyGuS solvers are at ceiling for these newest problems.

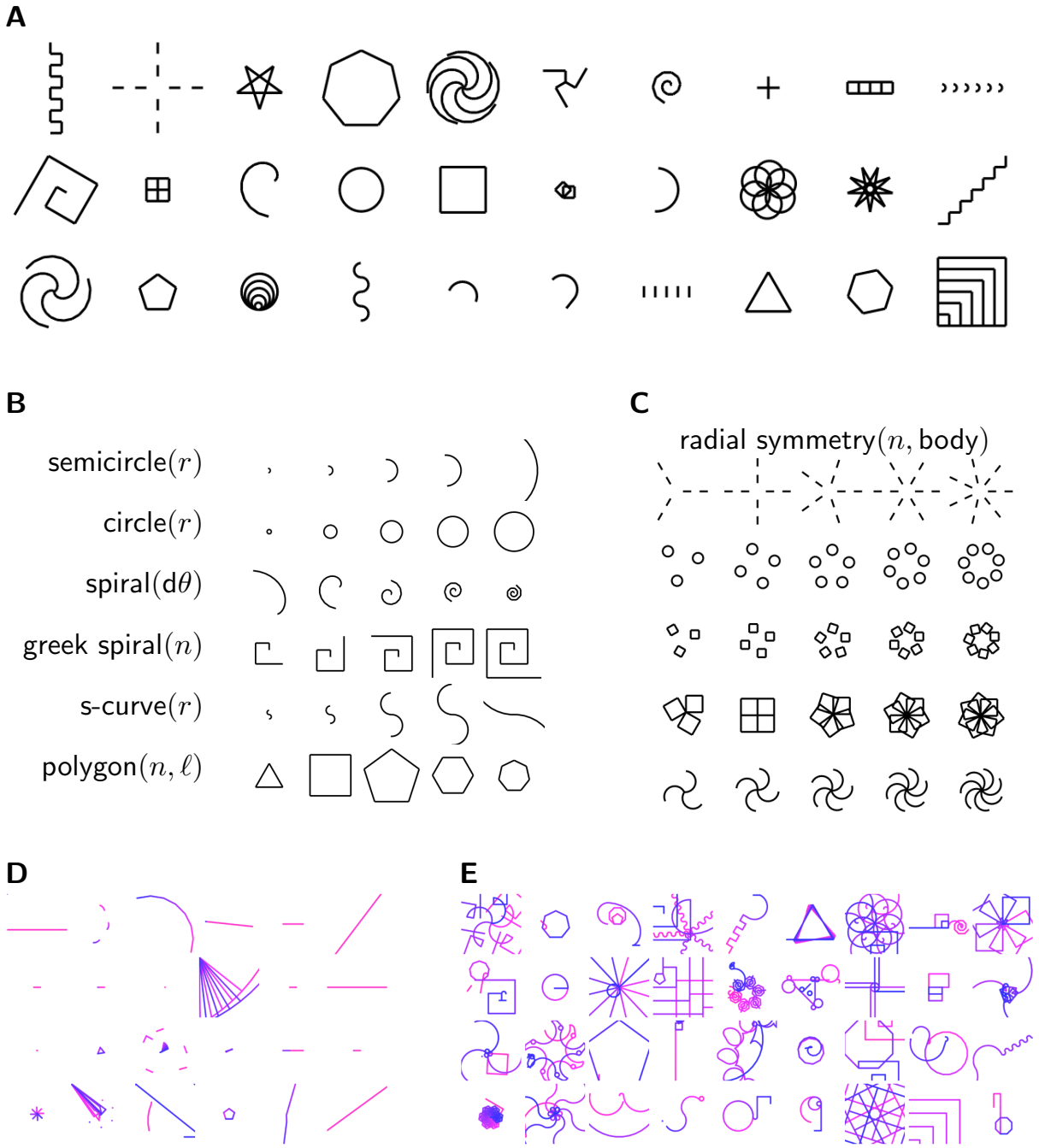


Figure 4: **(A):** 30 (out of 160) LOGO graphics tasks. The model writes programs controlling a 'pen' that draws the target picture. **(B-C):** Example learned library routines include both parametric routines for drawing families of curves (**B**) as well as primitives that take entire programs as input (**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. S5 shows 150 random dreams at each stage.

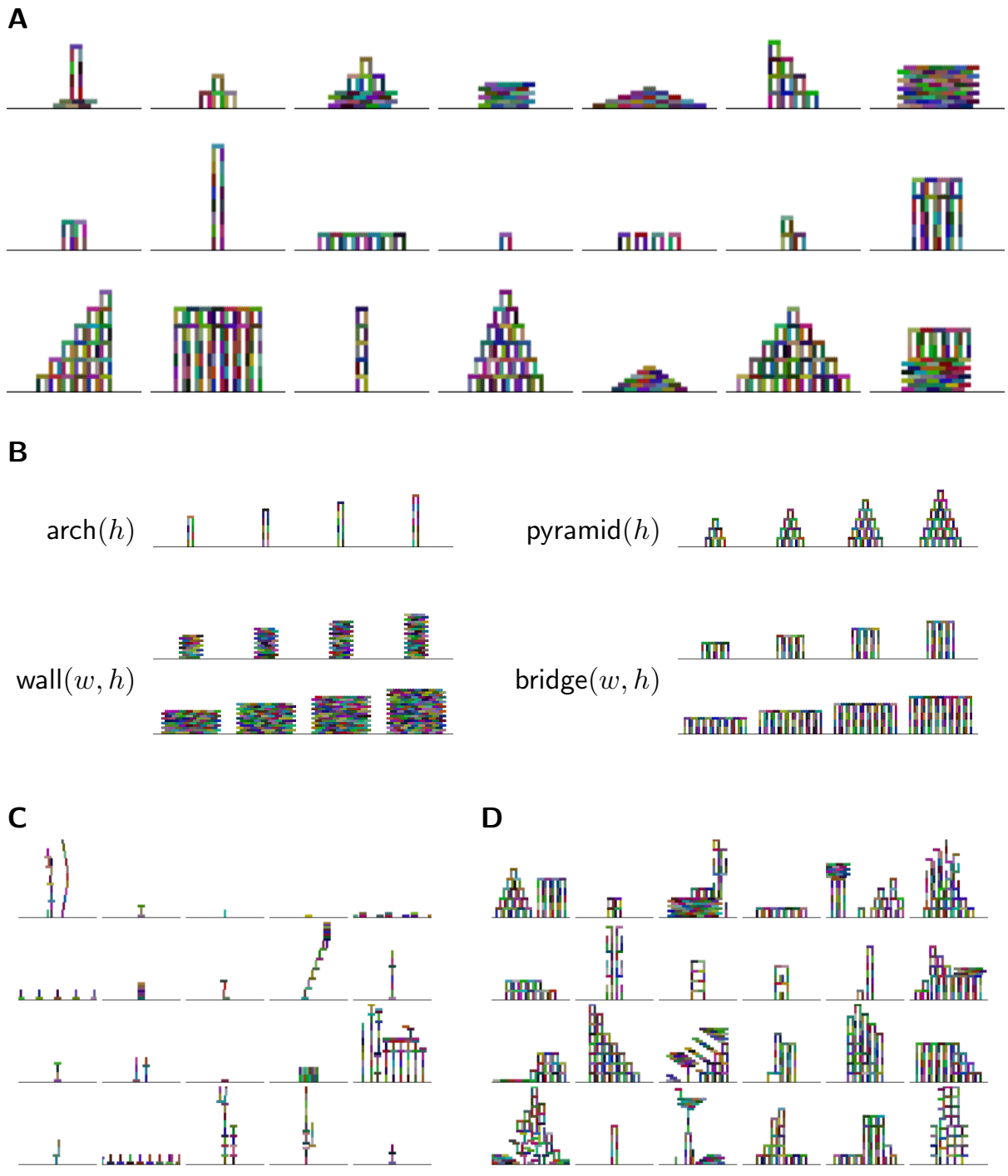


Figure 5: **(A)** 21 (out of 107) tower building tasks. The model writes a program controlling a 'hand' that builds the target tower. **(B)** Four learned library routines. These components act like parametric options (*30*), giving human-understandable, higher-level building blocks that the agent can use to plan. Dreams both before and after learning **(C-D)** show representative plans the model 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. S6 shows 150 random dreams at each stage.

symbols and signs (*4*), to learning new motor routines for producing words (*32*). 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 (*33*), 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. S4). 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) (*34*). 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* (*11*), which alternatingly 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 (*14*) model on samples from the initial library; and *Enumeration*, which performs type-directed enumeration (*22*) for 24 hours per task, generating and testing up to 400 million programs for each task. To isolate the role of refactoring, we construct two *Memorize* baselines. These baselines incorporate task solutions wholesale into the library, effectively memorizing solutions found during waking (cf. (*35*)). We evaluate memorize variants both with and without recognition models.

Across domains, our model always solves the most held-out tasks (Fig. 6A; see Fig. S12 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 and compression during the sleep-abstraction phase, and recognition model learning during the sleep-dreaming phase – contributes significantly to its overall performance. The synergy between these components is especially clear in the more creative, generative structure building domains, LOGO graphics and tower building, where no alternative model ever solves more than 60% of held-out tasks

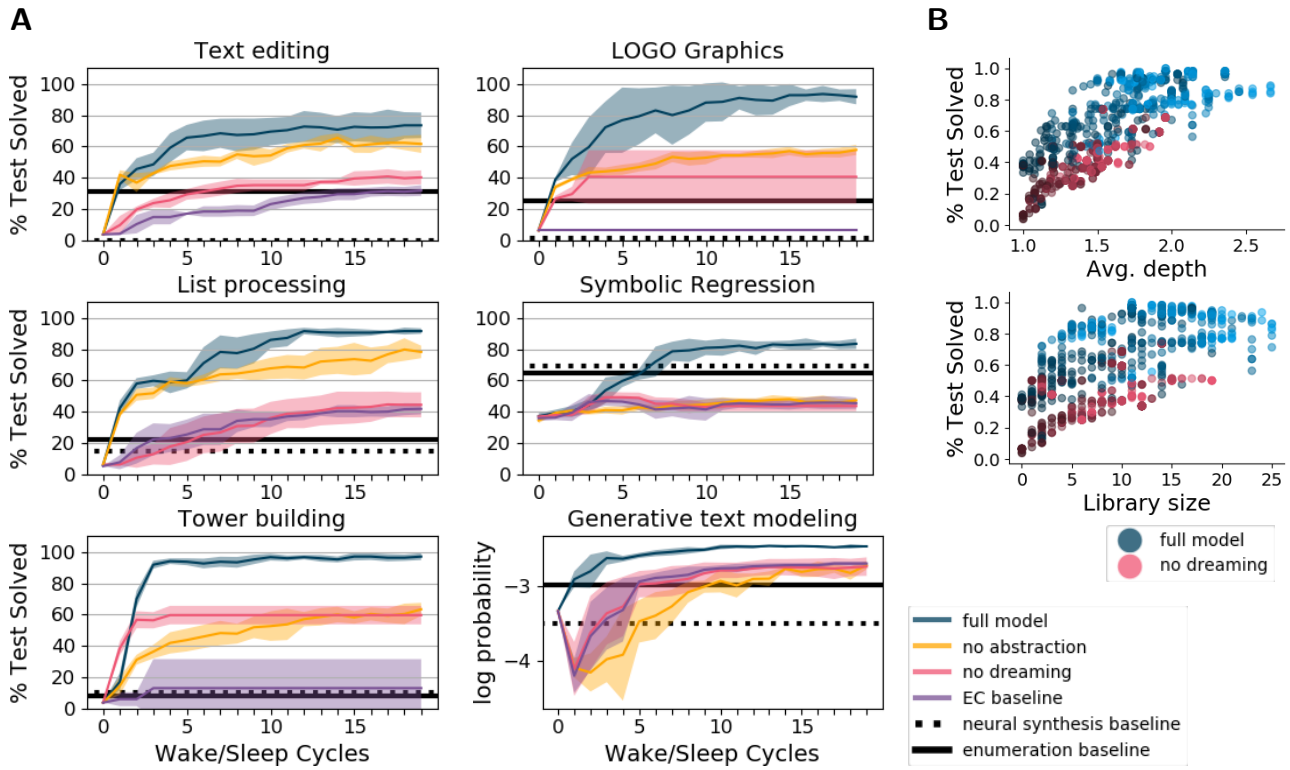


Figure 6: Quantitative comparisons of DreamCoder performance with ablations and baseline program induction methods; further baselines shown in Fig. S12. **(A)** Held-out test set accuracy, across 20 iterations of wake/sleep learning for six domains. Generative text modeling plots show posterior predictive 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.

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 problems in the activations of the recognition network guiding its search for solutions. 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. S3; $p < 10^{-4}$ 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. S2) – analogous to how human experts learn to classify problems by the underlying principles that govern their solution rather than superficial similarities (*9, 10*).

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

Motivated by classic work on inferring physical laws from experimental data (*36–38*), 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 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" (*39*). 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.

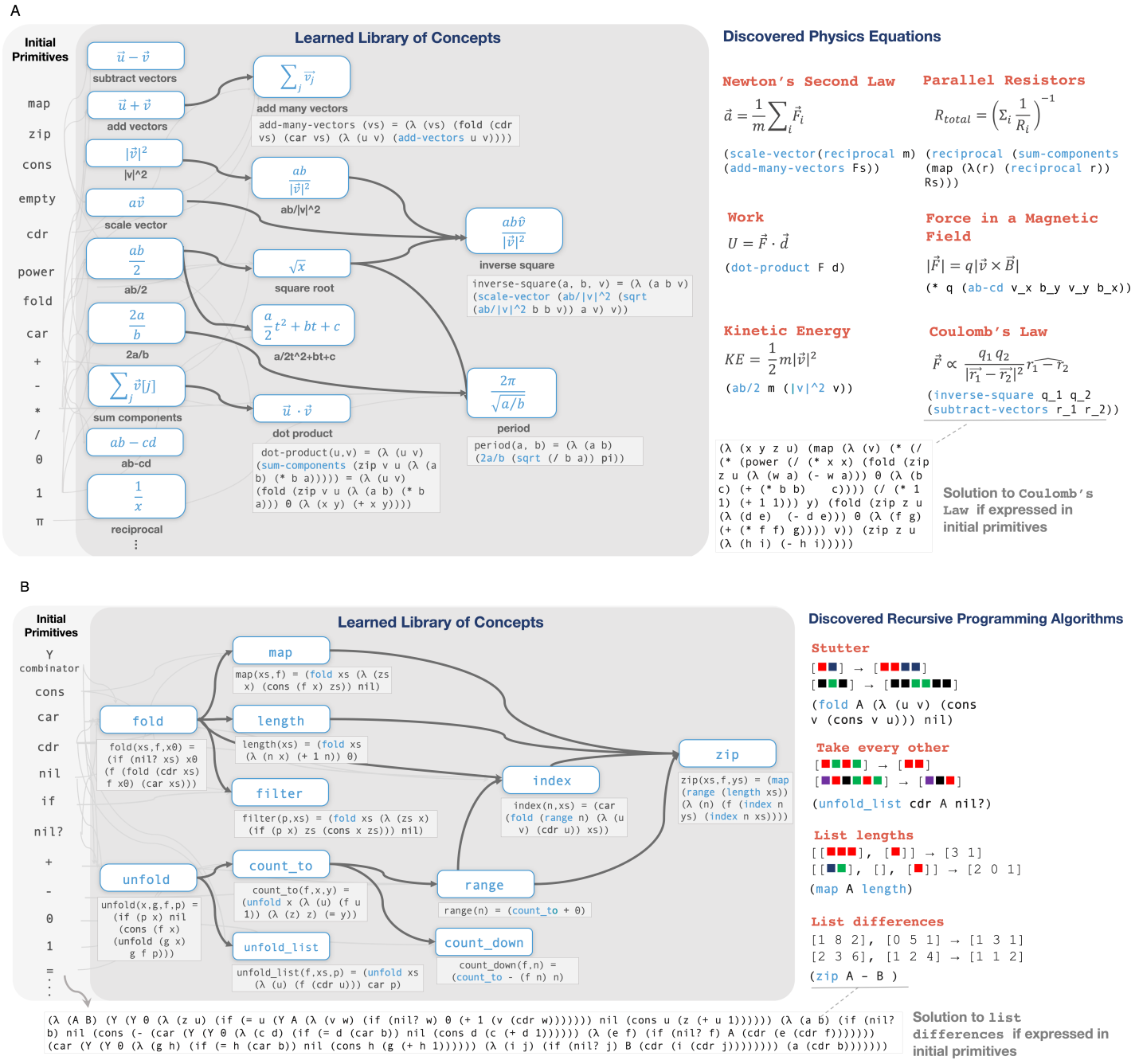


Figure 7: DreamCoder develops languages for physical laws (starting from recursive functions) and recursion patterns (starting from the Y-combinator, `cons`, `if`, etc.) **(A)** Learning a language for physical laws starting with recursive list routines such as `map` and `fold`. DreamCoder observes numerical 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.

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.

Interfaces with biological learning

DreamCoder's wake-sleep mechanics draw inspiration from the Helmholtz machine, which is itself loosely inspired by human sleep. A high-level difference separating DreamCoder from Helmholtz machines is our reliance on a pair of interleaved sleep cycles. Intriguingly, biological sleep similarly comes in various stages. Fast-wave REM sleep, or dream sleep, is associated with consolidation processes that give rise to implicit procedural skill, and engages both episodic replay and dreaming, analogous to our model's dream sleep phase. Slow-wave sleep is associated with the formation of new declarative abstractions, roughly mapping to our model's abstraction sleep phase. While neither DreamCoder nor the Helmholtz machine are intended as theories of biological sleep, we speculate that our approach brings wake-sleep learning algorithms closer to the structure of 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" (*40*) and "starting small" (*41*): Instead of solving increasingly difficult tasks ordered by a human teacher (the "curriculum"), DreamCoder moves through randomized batches 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 while solving easier ones. But human learners can select which tasks to solve, even without a human teacher, and can even generate their own tasks, either as steppingstones 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 a necessary next step.

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 (*42*), 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 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 (*1, 43–46*). We do not advocate such blank-slate learning as a route to AI —although, DreamCoder's ability to start from minimal bases and discover the vocabularies of functional programming, vector algebra, and physics could have been seen as a small step towards that goal. Children start from a rich endowment (*47–49*), and we strongly endorse learning that begins with the conceptual resources humans do. Rather than start from

a minimal basis, we believe the future lies with systems initialized with rich yet broadly applicable resources, such as those embodied by the standard libraries of modern programming languages. Indeed, while learning from scratch may be possible in theory, such approaches suffer from a notorious thirst for data–as in neural networks–or, if not data, then massive compute: to construct 'origami' functional programming, DreamCoder took approximately a year of total CPU time.

Rather than de novo acquisition of domain expertise, our approach shares similar philosophical motivations to the sketching approach to program synthesis (*21*). Sketching approaches consider single synthesis problems in isolation, and expect a human engineer to outline the skeleton of a solution. Analogously, we built in what we know how to build in: relatively spartan but generically powerful sets of control flow operators, higher-order functions, and types, and then used learning to grow a specialized language atop these bases. We do not anticipate a future where fully blank-slate program synthesis contributes broadly to AI: instead, we advocate richer systems of innate knowledge, coupled to program synthesis algorithms whose power stems from learning on top of this prior knowledge.

Limitations, and how to move beyond them

Our work focuses on problem-solving domains where the solution space is largely captured by crisp symbolic forms. This includes classic program synthesis areas as well as domains like LOGO graphics (where the task inputs are pixel images), generative text patterns (where the input data may contain exceptions and irregularities), and symbolic regression (where the solution space contains continuous parameters). Nonetheless, much real-world data is far messier than considered here. Program induction can scale to settings with heavier noise and more pervasive uncertainty (*4, 50, 51*), and does so by leaning more heavily on probabilistic and neural approaches to AI. We see deeper integration between these statistical learning methods and program synthesis as a productive path forward, as exemplified in the HOUDINI system (*52*), neural module networks (*53*), and others (*54, 55*).

Looking forward

Scaling up program induction beyond the tasks considered here to the full AI landscape — to commonsense reasoning, natural language understanding, or causal inference, for instance — demands innovation but holds great promise. Programs uniquely combine universal expressiveness with data-efficient learning and rich, abstract compositionality. As our work shows, we can now efficiently learn not just individual programs but whole systems of interrelated symbolic concepts as program libraries. Most importantly, the approach will need to be extended to learn not just one domain at a time, but to simultaneously develop expertise across many different classes of problems, autonomously carving out its own domains alongside its own domain-specific representations. We anticipate this will be enabled by metalearning a cross-domain library or "language-of-thought" (*56, 57*): growing a single generic programmatic substrate for learning and reasoning, as humans have built collectively through biological and cultural evolution, which can then differentiate itself into representations for unboundedly many different classes of problems.

References

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

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

3. 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.

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

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

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

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

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

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

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

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

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

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

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

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

16. 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.

17. 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.

18. 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.

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

20. 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.

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

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

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

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

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

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

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

28. 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.

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

30. 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.

31. 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.

32. 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.

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

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

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

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

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

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

39. Jeremy Gibbons. *Origami programming*. 2003.

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

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

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

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

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

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

46. Pedro Domingos. *The master algorithm: How the quest for the ultimate learning machine will remake our world*. Basic Books, 2015.

47. 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.

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

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

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

51. 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.

52. 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.

53. 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.

54. 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.

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

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

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

Acknowledgments. We thank L. Schulz, J. Andreas, T. Kulkarni, M. Kleiman-Weiner, J. M. Tenenbaum, 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. All code and data will be made available on a public github site upon publication. **Author contributions.** K.E., J.B.T., and A.S-L. conceived of the model. K.E., C.W., A.S-L., and J.B.T. created the algorithm. K.E., M.N., M.S-M., and C.W. ran experiments. K.E., L.C., C.W., M.N., L.M., and M.S-M. implemented the software. C.W., K.E., and J.B.T. designed illustrations. L.H., K.E., M.S-M., and L.M. collected task data sets. J.B.T. and A.S-L. advised the project. K.E., J.B.T., and C.W. wrote the paper. **Competing interests:** The authors declare no competing interests.

Supplementary Materials

We divide our supplementary materials into additional results (S1), material covering our experimental methods (S2), background and relation to prior work (S3), and our algorithmic techniques and software implementation (S4).

Contents

S1 Supplemental results

S1.1 Deep Representation Learning

Figure S1 (top) illustrates typical deep libraries learned across our domains, and this depth is correlated with % testing tasks solved. The recognition model leads both to deeper libraries and more solved tasks (Figure S1, bottom), which occurs because the recognition model allows more training tasks to be solved during waking, which are then compressed during the next sleep cycle.

S1.2 Neural Problem Representations

Figure S2 shows how DreamCoder organizes and represents its tasks over its learning trajectory for several domains. We visualize the inferred program representations (left columns) by calculating a feature vector for each task x, written $\phi(x)$, whose i^{th} component is the expected number of times that the i^{th} library component was used when solving the task:

$$\phi(x)_i = \mathrm{E}_{\mathrm{P}[\rho|L,x]}\left[\text{number of times } i \in L \text{ occurs in } \rho\right]$$

and then embed those feature vectors in a two dimensional space using tSNE (*26*). We can visualize the recognition model's task representations by flattening the neural output $Q_{ijk}(x)$ (see S4.6) to obtain a feature vector for each task; or, alternatively, we can interrogate the neural network's output by taking the above expectation over the approximate posterior $Q(\rho|x)$ rather than the true posterior $\mathrm{P}\left[\rho|L,x\right]$. We visualize these neural task representations also using tSNE both after the first wake/sleep cycle (left column) and after the final wake/sleep cycle (middle column). For list and text, we visualize the raw output $Q_{ijk}(x)$; for logo and towers, we visualize the above expectation taken over the recognition model's posterior $Q(\rho|x)$ rather than the true posterior, because this gave more interpretable clusterings.

S1.3 Generating new examples of text concepts

Figure S4 shows representative generative regex tasks, where DreamCoder infers a generative text-generating process (a probabilistic regex) from a small number of example strings, and contrasts the output of the full model with ablations that lesion either library learning or recognition model training. See S2.1.5 for methods.

S1.4 Dreams before and after learning

Figures S5-S6 show 150 randomly sampled dreams from an agent learning to draw pictures using LOGO graphics and learning to build towers, respectively, both prior to any learning (i.e., random programs drawn from the initial library) and from the final state of learning across five different runs.

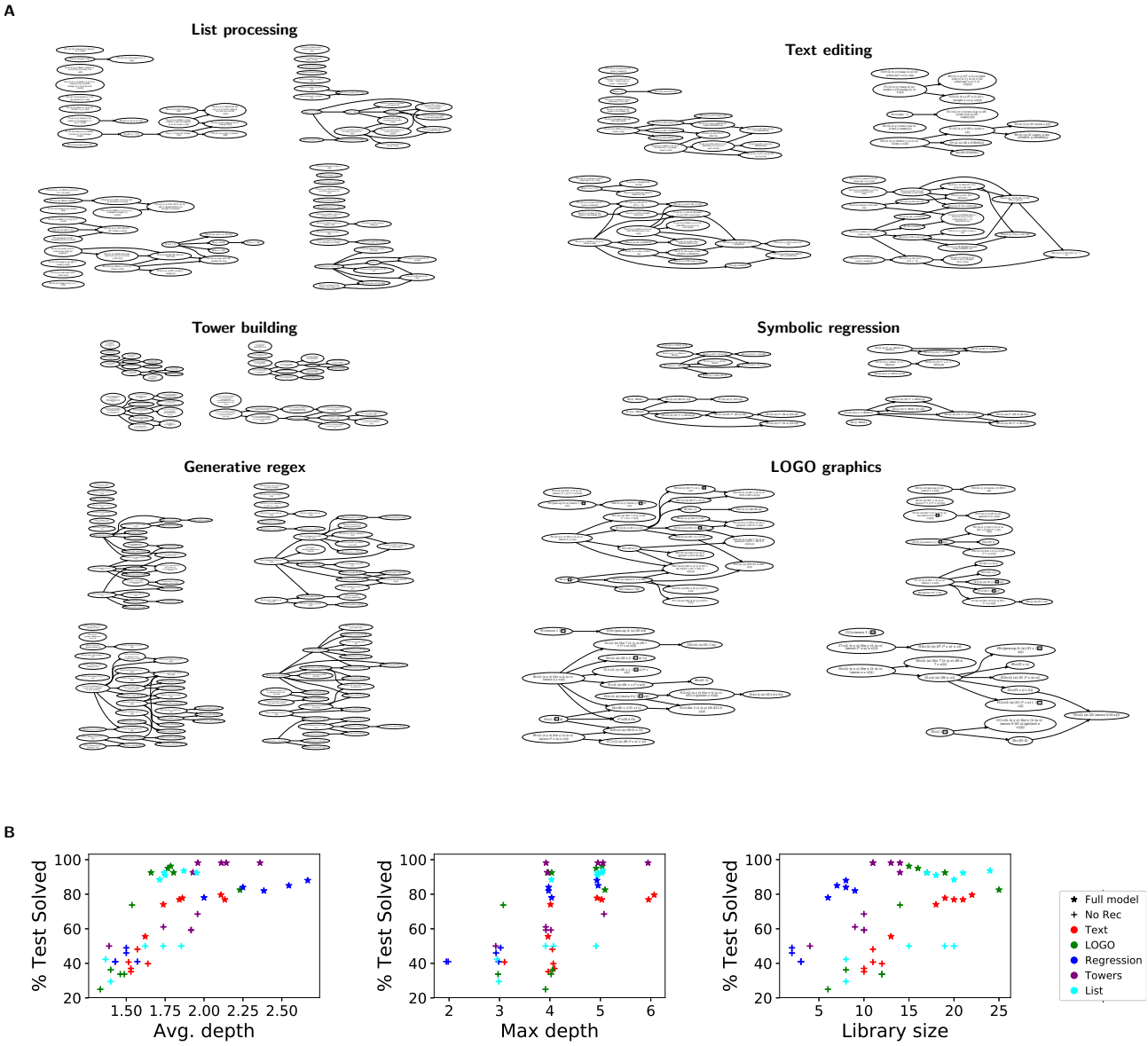


Figure S1: **(A)**: Learned libraries diagrammed as graphs. Lines (left-or-right) go from a learned concept to all other concepts that use it. For each domain we show four libraries from four different runs, showing the variability in learned library structure, which **(B)** plots in relation to held-out testing performance, for each random seed and for each domain, contrasting the full model with an ablation missing the recognition model (i.e. missing dreaming). Performance at the final iteration plotted. In **(A)** only learned library routines are shown; non-learned, built in primitives (depth 1) ellided. Similarly in **(B)** we compute library size ignoring the initial set of primitives.

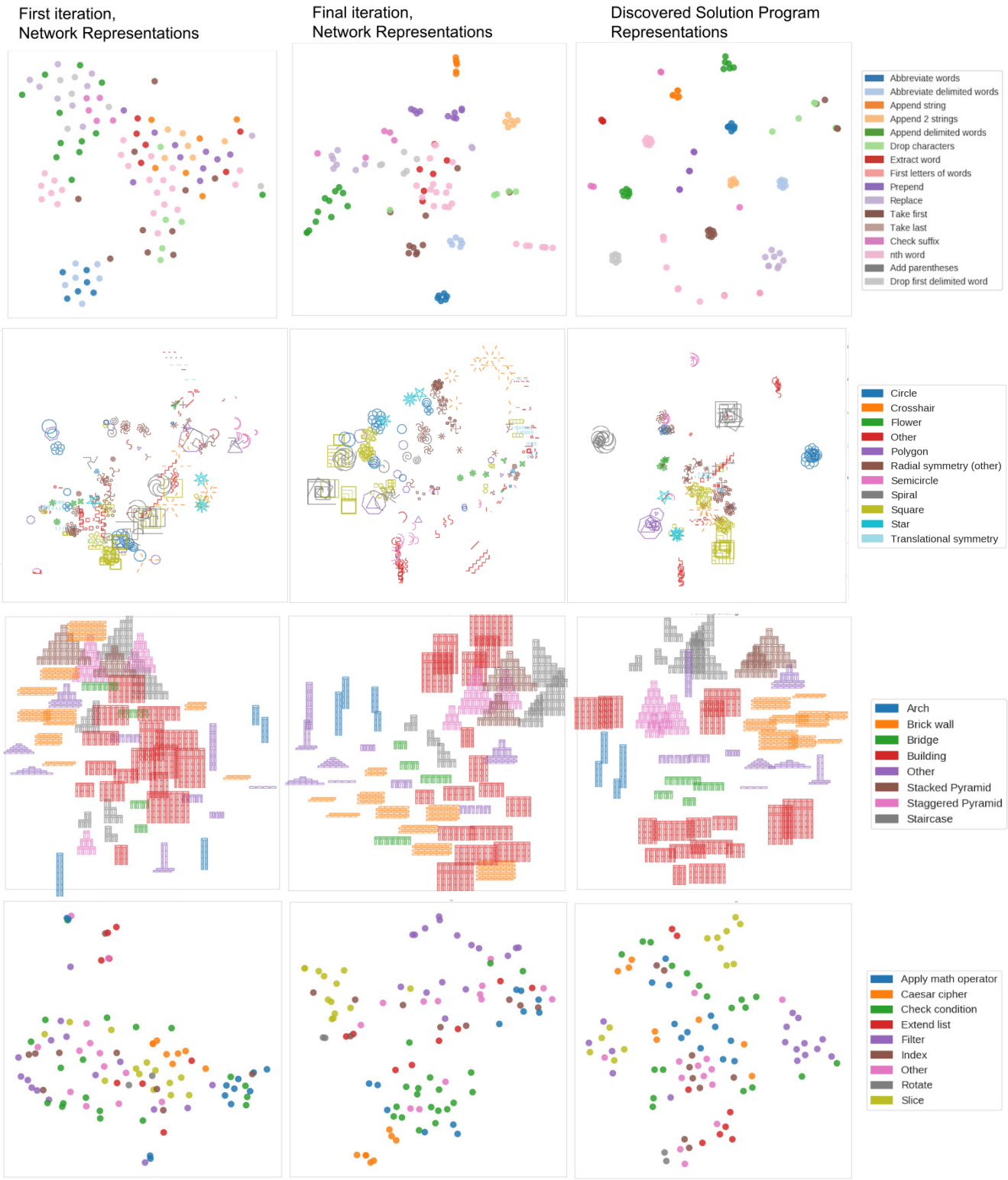


Figure S2: Left: TSNE embedding of model's 'at-a-glance' (before searching for solution) organization of tasks (neural net activations), after first wake/sleep cycle. Middle: at-a-glance organization after last wake/sleep cycle. Right: TSNE visualization of programs actually used to solve tasks; intuitively, how model organizes tasks after searching for a solution. We convert each program into a feature vector by counting the # times each library routine is used, and then embed those feature vectors.

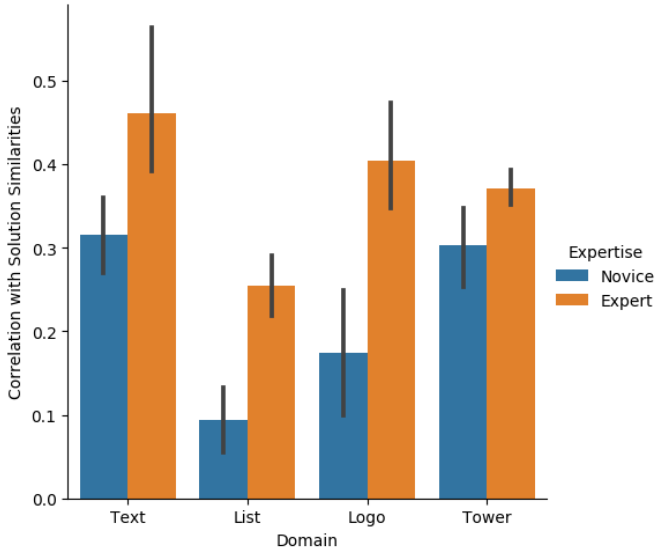


Figure S3: Correlation between similarity matrices of neural net activations and program solution feature vectors, both after the first wake/sleep cycle (Novice) and at the last cycle (Expert). Error bars represent SEM over 5 random seeds. Both correlations and similarities computed using cosine distance.

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

Input		MAP program	Samples
Y2015/1093 Y2013/1010 Y2014/1017 Y2015/1421 Y2017/1162	Full	Y201d/dddd	Y2010/3308 Y2010/1163
	No Library	Y201d/dddd	Y2011/1131 Y2015/7116
	No Rec	Y201(d)*d/(d)*	Y20127/20411 Y2011214/1
-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	-00:dd.dd.d	-00:16g22:5 -00:22.53\t2
	No Rec	(-00:)?(.)*dd.d	-00:i47.5 -00:r59.0
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	. - (d(.)*)*..	. - . tY - E5
	No Rec	u - (d)*(.)*	L - 5208s. S - 5533..

Figure S4: Results on held-out text generation tasks. Agent observes 5 strings ('Input') and infers a probabilistic regex ('MAP program'–regex primitives highlighted in red), from which it can imagine more examples of the text concept ('Samples'). No Library: Dreaming only (ablates library learning). No Rec: Abstraction only (ablates recognition model)

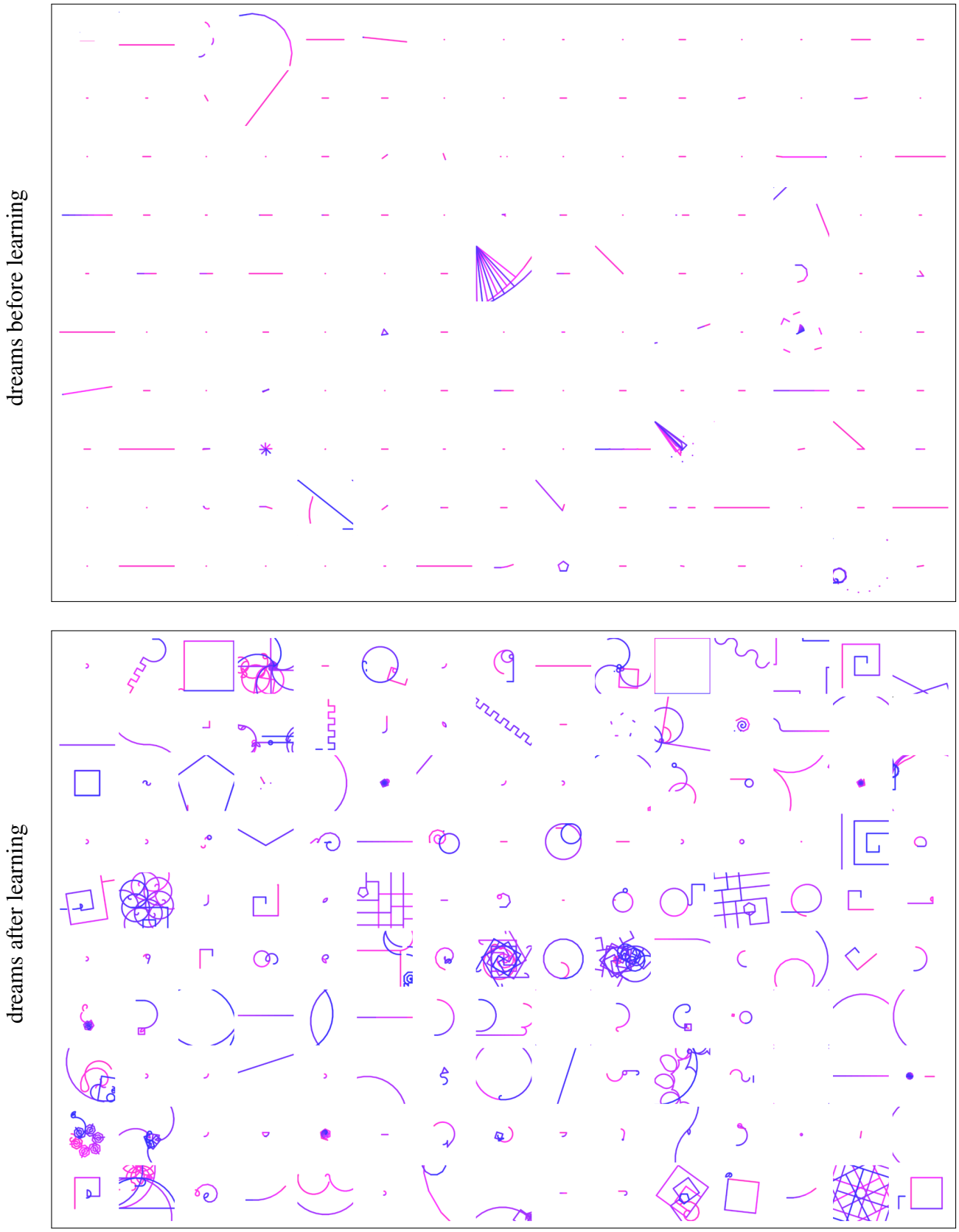


Figure S5: Dreams, or randomly sampled programs, built from the initial library (top) and the final learned library (bottom), drawn at random from 5 different runs.

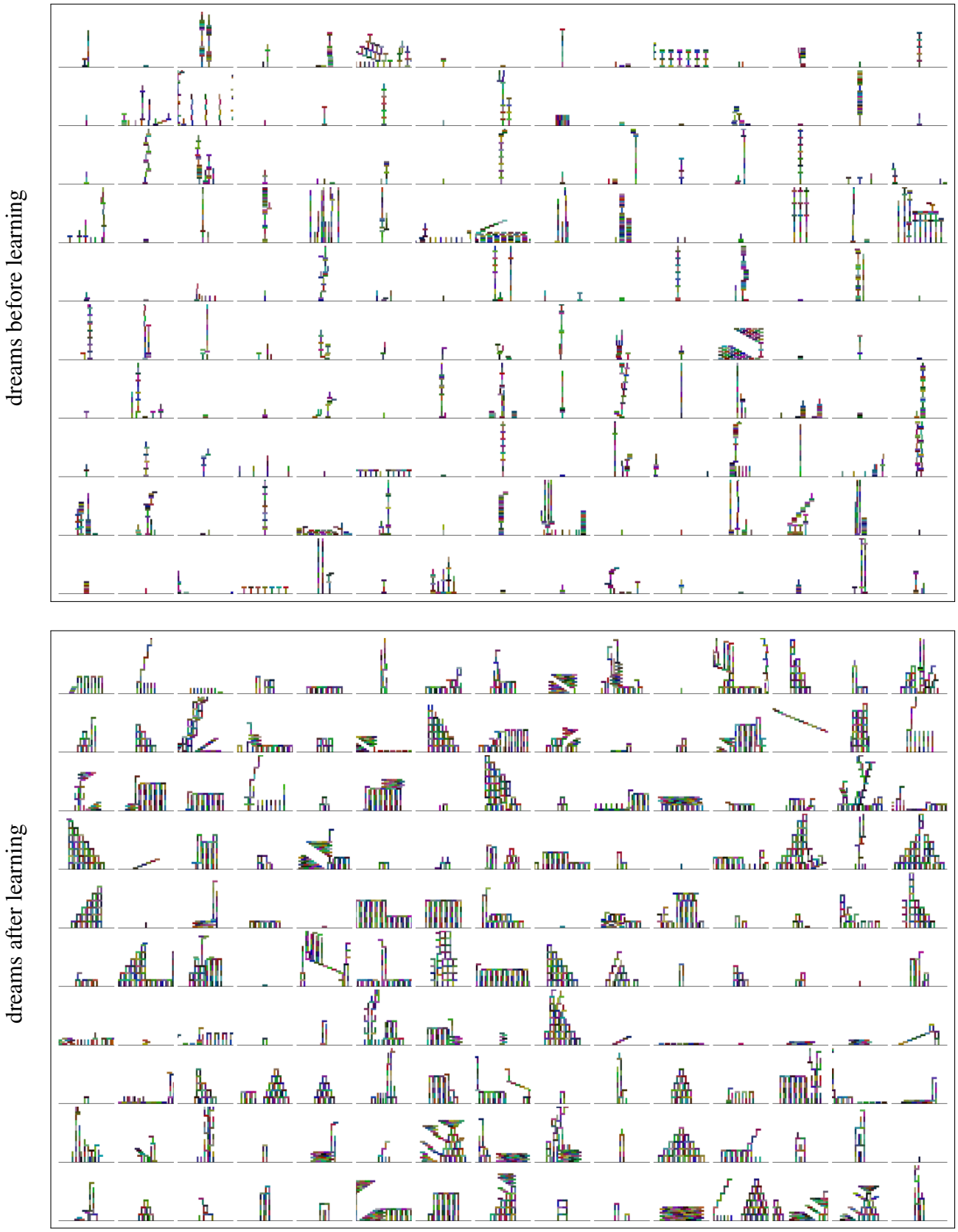


Figure S6: Dreams, or randomly sampled programs, built from the initial library (top) and the final learned library (bottom), drawn at random from 5 different runs.

S2 Experimental methods

S2.1 Problem Solving Domains

S2.1.1 List Processing

We took the 236 list processing problems from (*9*), removed 21 tasks that were equivalent to the identity function, randomly split the remaining tasks 50/50 train/test, and then made the testing set more difficult by excluding 31 test-set problems which were solvable within 10 minutes using the starting library and without using a recognition model. The system starts with the following sequence manipulation library routines: `fold`, `unfold`, `if`, `map`, `length`, `index`, `=`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`. The system additionally has the following numerical routines: `mod`, `*`, `>`, `is-square`, and `is-prime`.

S2.1.2 Text Editing

We trained on 128 synthetically generated tasks taken from (*9*) and tested on the 108 tasks from the 2017 SyGuS program synthesis competition in the "PBE" category. The system is initialized with the same sequence manipulation primitives we used for list processing, and we model strings as lists of characters. We built in an additional primitive, `string`, that represents an unknown string-valued parameter. During waking, we substitute occurrences of `string` for the longest common subsequence (LCS) occurring in the output examples. This allows DreamCoder to solve text editing problems requiring constant strings in the output.

S2.1.3 LOGO Graphics

We apply our system to 160 LOGO graphics tasks, split 50/50 test/train (Figure S7). For each task of the agent must write a program that moves a simulated pen across the canvas, with the goal of drawing the target image, as measured by pixel level similarity. We initially provide our agent with two control flow primitives: `for` (a 'for' loop), and `get/set`, which gets and saves the current state of the agent's pen, executes a block of code, and then restores the state to its previous value. We provide two different drawing primitives: `move`, which takes as input both a distance and angle, and moves the pen forward by that distance and rotates by that angle, and `pen-up`, which lifts up the pen and then executes a block of code *without* placing down ink on the canvas.

These programs are imperative (produce a sequence of actions) rather than purely functional (calculate a stateless mapping). The DreamCoder infrastructure only supports purely functional λ-calculus programs. To embed imperative routines within this framework, we model imperative routines using a state monad (*48*) and encode each imperative action in a continuation passing style where each imperative primitive takes as input the actions to execute afterwards (i.e., the continuation), and the program as a whole takes as input a final no-op continuation. This continuation passing set up allows the program synthesizer to not need the monadic 'bind' operator to sequence chains of commands. To be concrete, our imperative primitives have the following types:

```
move : distance→angle→State ()→State ()
pen-up : (State ()→State ())→State ()→State ()
for : int→(State ()→State ())→State ()→State ()
get/set : (State ()→State ())→State ()→State ()
```

Each synthesized program has type `State ()→State ()`. We additionally provide the agent with natural numbers (1 through 9), arithmetic operations (addition/subtraction/multiplication/division), unit distances and angles (1 meter and 2π radians), and special values ∞ (approximated as 20) and ϵ (approximated

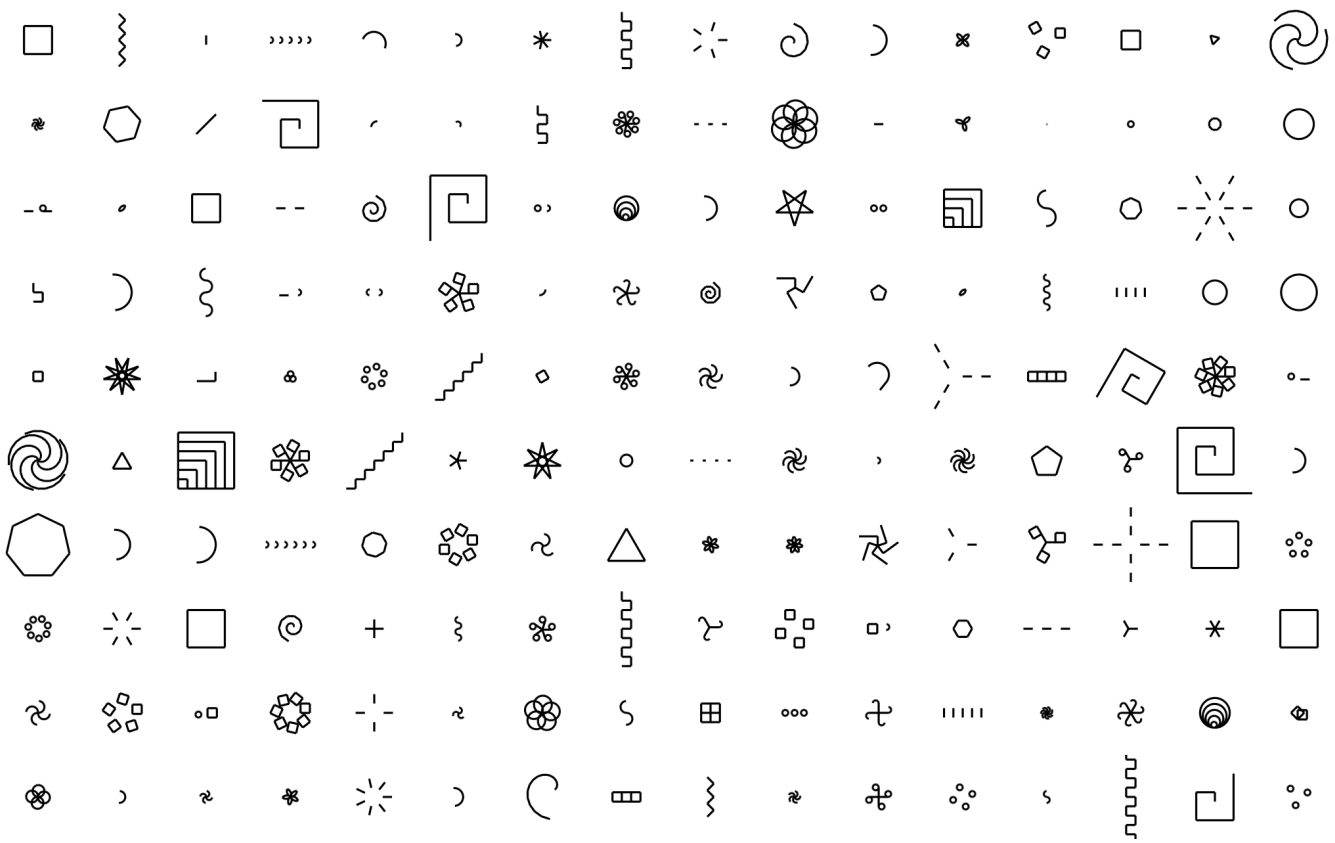


Figure S7: Full set of 160 LOGO graphics tasks that we apply our system to. Top 80 tasks: training set. Bottom 80 tasks: testing set.

as $1/20$). We distinguish distances and angles in the type system, only allowing a distance/angle to be added/subtracted with another object that is also a distance/angle, and only allowing multiplication/division by unitless scalars. This trick requires no modification of the underlying DreamCoder software, and is a standard way of embedding units and dimensions within a type system (*30*).

S2.1.4 Tower Building

We apply our system to 107 tower building tasks (Figure S8), split 50/50 test/train. We use the same monadic, continuation-passing encoding of imperative programs as we used for LOGO graphics, and include the exact same control flow primitives. Rather than moving a pen over a canvas, the agent here moves a simulated hand left/right over a 2D tower building stage, and has at its disposal an unlimited supply of horizontal and vertical blocks. All coordinates are discretized. Horizontal blocks have size 6x2 and vertical blocks have size 2x6. The state of the simulated hand maintains both its position (a single discrete scalar) and its orientation (a single boolean, either facing left are facing right). We include two domain specific primitives for adjusting the position of the hand: `move`, which takes as input a scalar d and moves the hand forward by distance d, and `reverse`, which flips the orientation of the hand. The agent has two additional domain specific primitives for dropping a horizontal/vertical block at the current position of the hand.

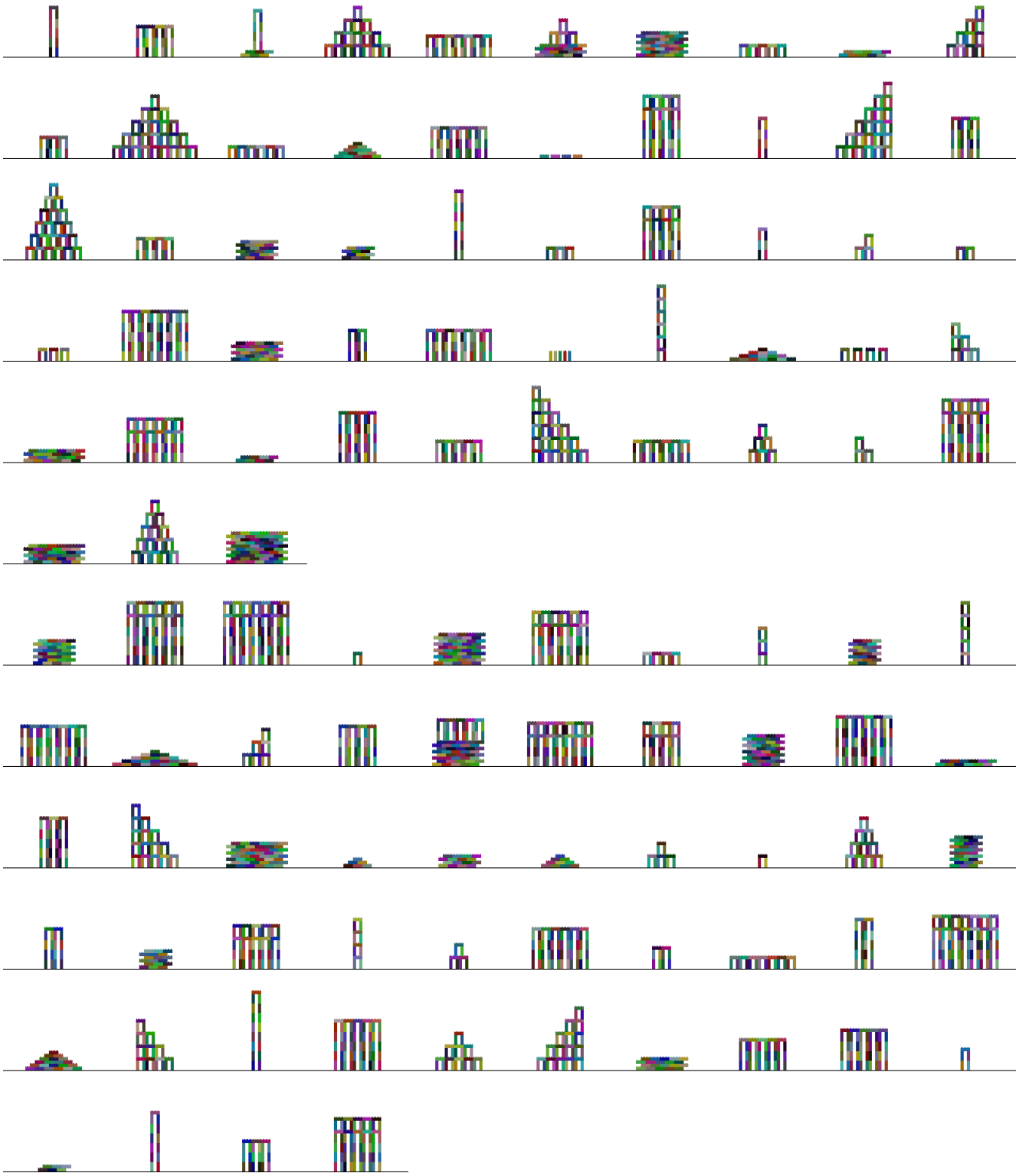


Figure S8: Full set of 107 tower building tasks that we apply our system to. Top 53 tasks: training set. Bottom 54 tasks: testing set.

S2.1.5 Probabilistic Generative Regexes

We took 256 CSV columns crawled from the web (drawn from (*15*), tasks split 50/50 test/train, 5 example strings per concept) and tasked our system, for each set of 5 strings, with inferring a regular expression matching that set of strings. For each regex primitive, e.g. `[0-9]`, `[a-z]`, etc., we define a probabilistic generative process that samples instances of that regex primitive. Then we define the likelihood model $\mathrm{P}[x|\rho]$, where x is a set of strings and ρ is a regular expression, as $\prod_{\sigma \in x} \mathrm{P}[\sigma|\rho]$. We include the following regex primitives: `[0-9]`, `[a-z]`, `[A-Z]`, any symbol, i.e. Σ or `.`, all printable ASCII characters, Kleene star, disjunction, and 'optional' (`?`).

For each task we have both 5 randomly sampled training strings (i.e. the task x), as well as a disjoint set of 5 randomly sampled testing strings (written test(x)). In Figure S11 we show both the posterior predictive likelihood of held-out strings on held out tasks, which is $\sum_{\rho \in \mathcal{B}_x} \frac{\mathrm{P}[x|\rho]\mathrm{P}[\rho|L]}{\sum_{\rho' \in \mathcal{B}_x} \mathrm{P}[x|\rho']\mathrm{P}[\rho'|L]} \prod_{\sigma \in \text{test}(x)} \mathrm{P}[\sigma|\rho]$, as well as the marginal likelihood of held out tasks, which is $\sum_{\rho \in \mathcal{B}_x} \mathrm{P}[x|\rho]\mathrm{P}[\rho|L]$, where in both cases x ranges over held out testing tasks, and $\mathcal{B}_x$ corresponds to the programs found during waking that assign positive likelihood to task x (see Sec. S4.1 for the definition of $\mathcal{B}_x$).

When generating new instances of a text concept, we fine-tune the continuous parameters of the generative regex by backpropagation through the calculation of $\mathrm{P}[x|\rho]$. Figure S4 shows representative results on held-out tasks. When training the recognition model, we used the 'unigram' parameterization of EC^2 rather than the 'bigram' parameterization introduced in S4.6, because the number of bigram parameters grows quadraticly with the number of primitives, and we have one primitive for each printable ASCII character.

S2.1.6 Symbolic regression: Programs for Smooth Trajectories

Many procedures that people learn, like as motor routines, can be thought of as some kind of smooth trajectory. Modeling these procedures as programs entails jointly inferring their combinatorial structure and their continuous parameters. We consider simple programs generating trajectories described by polynomials (up to degree 4) and rational functions, following (*9*). We equip our learner with a primitive for introducing a continuous parameter into the discrete structure of its programs, which an inner loop optimizes via gradient descent. We penalize the use of real-valued parameters using a BIC (*3*) likelihood model. We use a convolutional neural network as a 'visual' recognition model, which observes a graph of the trajectory over time, allowing the agent to 'eyeball' the trajectory prior to writing code that describes it (Figure S9). In solving 100 such tasks, the model learns to find programs minimizing the number of continuous parameters — this phenomenon arises from our Bayesian framing: both the generative model's bias toward shorter programs, and the likelihood model's BIC penalty.

S2.1.7 Learning a language for physical laws

We took equations from MCAT as well as AP Physics B & C cheat sheets (`http://mcat.prep101.com/wp-content/uploads/ES_MCATPhysics.pdf`; `https://secure-media.collegeboard.`

Figure S9: Sixteen smooth-trajectory tasks. Agent writes a program containing continuous real numbers that fits the points along the curve. Recognition model is a CNN that takes as input a graph of the trajectory over time (shown above). Task is 'solved' if program matches points on curve.

org/digitalServices/pdf/ap/ap-physics-1-equations-table.pdf;https://secure-media.collegeboard.org/digitalServices/pdf/ap/physics-c-tables-and-equations-list.pdf), choosing representative laws, but excluding expressions with trigonometric functions, multivariate differentials, and integrals. Table S1 shows the full set of tasks we apply DreamCoder to. Each equation was converted to a symbolic regression problem by sampling 10 random input/output examples, where the inputs correspond to free variables in the target expression. Physical constants were expressed in Planck units, which generally drops certain special coefficients from these expressions (e.g., the gravitational constant $G = 1$). We gave the system the following primitives: `0`, `1`, π, `power`, `+`, `*`, `/`, `-`, `map`, `fold`, `zip`, `cons`, `car`, `cdr`, and `nil`.

At each wake cycle, rather than sample a random minibatch of tasks, we gave the agent all tasks it had not solved in the past two wake cycles. This had the effect of the agent solving the easiest tasks first, and then reallocating its compute time to a smaller set of harder, unsolved tasks.

S2.1.8 Learning a language of recursive functions

We gave our system the following primitives: `if`, `=`, `>`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`, all of which are present in some form in McCarthy's 1959 Lisp (*28*).[1] We furthermore allowed functions to call themselves, which we modeled using the Y combinator. We did not use the recognition model for this experiment: a bottom-up pattern recognizer is of little use for acquiring this abstract knowledge from less than two dozen problems. Figure S10 shows the full set of tasks and the learned library.

S2.2 Performance on held out testing tasks over wake/sleep cycles

Figure S11 shows both solved testing tasks and average solved times as a function of wake/sleep cycles. For probabilistic generative regexes we show marginal likelihood of held out tasks; we do not show "time to solve" for the regex domain because there is no hard all-or-none discriminator between solving a task and failing to solve a task. Figure S12 shows solved tasks for a pair of baseline models which, rather than build libraries using our refactoring algorithm, instead simply memorize programs found during waking, incorporating them wholesale into the learned library.

[1] McCarthy's first version of Lisp used `cond` instead of `if`. Because we are using a typed language, we instead used `if`, because Lisp-style `cond` is unwieldy to express as a function in typed languages.

Freefall velocity	$\sqrt{2gh}$	Ballistic velocity	$v^2 = v_0^2 + 2a(x - x_0)$
Velocity magnitude	$v = \sqrt{v_x^2 + v_y^2}$	Angular acceleration	$a_r = v^2/R$
Mass-energy equivalence	$E = mc^2$	Center of mass	$\sum_i m_i x_i / \sum_i m_i$
Center of mass	$(m_1\vec{x_1} + m_2\vec{x_2})/(m_1 + m_2)$	Density	$\rho = m/v$
Pressure	F/A	Power	$P = I^2R$
Power	$P = V^2/R$	RMS voltage	$V/\sqrt{2}$
Energy in capacitor	$U = 1/2CV^2$	Energy in capacitor	$1/2QV$
Energy in capacitor	$1/2Q^2/C$	Optical power	$P = 1/f$
Focal length, curvature	$c = r/2$	Net force	$\vec{F}_{\text{net}} = \sum_i \vec{F}_i$
Newton's law	$\vec{a} = \frac{1}{m}\sum_i \vec{F}_i$	Work	$\vec{F} \cdot \vec{d}$
Work per time	$\vec{F} \cdot \vec{v}$	Lorentz force (3D)	$q\vec{v} \times \vec{B}$
Lorentz force (2D)	$q(v_x B_y - v_y B_x)$	Torque (3D)	$\vec{\tau} = \vec{r} \times \vec{F}$
Torque (2D)	$\lvert\vec{\tau}\rvert = r_x F_y - r_y F_x$	Ballistic velocity	$v(t) = v_0 + at$
Ballistic motion	$x(t) = x_0 + v_0 t + \frac{1}{2}at^2$	Momentum	$\vec{p} = m\vec{v}$
Impulse	$\Delta\vec{p} = \vec{F}\Delta t$	Kinetic energy	$\text{KE} = \frac{1}{2}m\lvert\vec{v}\rvert^2$
Kinetic energy (rotation)	$\text{KE} = \frac{1}{2}I\lvert\vec{\omega}\rvert^2$	Charge flux→Field	$\vec{E} = \rho\vec{J}$
Hook's law	$\vec{F}_{\text{spring}} = k\vec{x}$	Hook's law	$\vec{F}_{\text{spring}} = k(\vec{r}_1 - \vec{r}_2)$
Power	$P = \mathrm{d}E/\mathrm{d}t$	Angle over time	$\theta(t) = \theta_0 + \omega_0 t + \frac{1}{2}\alpha t^2$
Angular velocity over time	$\omega(t) = \omega_0 + \alpha t$	Rotation period	$T = \frac{2\pi}{\omega}$
Spring period	$T_{\text{spring}} = 2\pi\sqrt{\frac{m}{k}}$	Pendulum period	$T_{\text{pendulum}} = 2\pi\sqrt{l/g}$
Spring potential	$\text{E}_{\text{spring}} = kx^2$	Coulomb's law (scalar)	$C\frac{q_1 q_2}{\lvert\vec{r}\rvert^2}$
Ohm's law	$V = IR$	Power/Current/Voltage	$P = VI$
Gravitational potential energy	$9.8 \times mh$	Time/frequency relation	$f = 1/t$
Plank relation	$E = h\nu$	Capacitance	$C = V/Q$
Series resistors	$R_{\text{total}} = \sum_i R_i$	Parallel capacitors	$C_{\text{total}} = \sum_i C_i$
Series capacitors	$C_{\text{total}} = \frac{1}{\sum_i 1/C_i}$	Area of circle	$A = \pi r^2$
Pythagorean theorem	$c^2 = a^2 + b^2$	Vector addition (2)	$(\vec{a} + \vec{b})_i = \vec{a}_i + \vec{b}_i$
Vector addition (n)	$(\sum_n \vec{v}^{(n)})_i = \sum_n \vec{v}_i^{(n)}$	Vector norm	$\lvert\vec{v}\rvert = \sqrt{\vec{v} \cdot \vec{v}}$
Newtonian gravitation (2 objects)	$G\frac{m_1 m_2}{\lvert\vec{r}_1 - \vec{r}_2\rvert^2}\widehat{\vec{r}_1 - \vec{r}_2}$		
Newtonian gravitation (displacement)	$G\frac{m_1 m_2}{\lvert\vec{r}\rvert^2}\widehat{\vec{r}}$		
Newtonian gravitation (scalar)	$G\frac{m_1 m_2}{\lvert\vec{r}\rvert^2}$		
Coulomb's law (2 objects)	$C\frac{q_1 q_2}{\lvert\vec{r}_1 - \vec{r}_2\rvert^2}\widehat{\vec{r}_1 - \vec{r}_2}$		
Coulomb's law (displacement)	$C\frac{q_1 q_2}{\lvert\vec{r}\rvert^2}\widehat{\vec{r}}$		

Table S1: Physical laws given to DreamCoder as regression tasks. Expressions drawn from AP and MCAT physics "cheat sheets" and textbook equation guides.

Programs & Tasks

length
[1 9]→2
[5 3 8]→3
$f(\ell)$=(len ℓ)

lengths of lists
[[2 1] []]→[2 0]
[[] [] [9 8 9 9]]→[0 0 4]
$f(\ell)$=(map len ℓ)

keep positives
[0 1 1 0 0]→[1 1]
[9 0 8]→[9 8]
$f(\ell)$=(filter (eq? 0) ℓ)

remove negatives
[1 -1 0 2]→[1 2]
[9 -5 5 0 8]→[9 5 8]
$f(\ell)$=(filter (gt? 1) ℓ)

append zero
[2 1 4]→[2 1 4 0]
[9 8]→[9 8 0]
$f(\ell)$=(fold cons ℓ (cons 0 nil))

drop last element
[2 1 4]→[2 1]
[9 8]→[9]
$f(\ell)$=(unfold-list (λ (z) z)
ℓ (λ (z) (empty? (cdr z))))

sum elements
[2 5 6 0 6]→19
[9 2 7 6 3]→27
$f(\ell)$=(fold + ℓ 0)

double elements
[4 2 6 4]→[8 4 12 8]
[2 3 0 7]→[4 6 0 14]
$f(\ell)$=(map (λ (x) (+ x x)) ℓ)

negate elements
[4 2 6 4]→[-4 -2 -6 -4]
[2 3 0 7]→[-2 -3 -0 -7]
$f(\ell)$=(map (- 0) ℓ)

increment elements
[4 2 6 4]→[5 3 7 5]
[2 3 0 7]→[3 4 1 8]
$f(\ell)$=(map (+ 1) ℓ)

take every other
[1 5 2 9]→[1 2]
[3 8 1 3 1 2]→[3 1 1]
$f(\ell)$=(unfold-list cdr ℓ empty?)

range
3→[0 1 2]
2→[0 1]
$f(n)$=(range n)

inclusive range
3→[0 1 2 3]
2→[0 1 2]
$f(n)$=(range (+ 1 n))

stutter
[9 2]→[9 9 2 2]
[1 2 3 4]→[1 1 2 2 3 3 4 4]
$f(l)$=(fold (λ (a x) (cons x
(cons x a))) l nil)

0-index
0, [9 2 3]→9
3, [0 2 8 4 5 6]→4
f(n,l)=(index l n)

1-index
1, [9 2 3]→9
4, [0 2 8 1 5 6]→1
f(n,l)=(index l (+ 1 n))

count upward til zero
3→[-3 -2 -1]
4→[-4 -3 -2 -1]
f(n)=(count_down (λ (z) (- z n)) 0)

count downward til zero
2→[2 1]
4→[4 3 2 1]
f(n)=(count_down (λ (z) (+ z n)) 1)

add corresponding elements
[0 2 1], [1 2 0]→[1 4 1]
[9 6], [9 4]→[18 10]
f(a,b)=(zip a + b)

subtract corresponding elements
[1 1 9], [1 0 5]→[0 1 4]
[8 7], [7 8]→[1 -1]
f(a,b)=(zip b - a)

Figure S10: Bootstrapping a standard library of recursive functional programming routines, starting from recursion along with primitive operations found in 1959 Lisp. Complete set of tasks shown above w/ representative input/output examples. Learned library routines (map/fold/etc.) diagrammed in Figure 7.

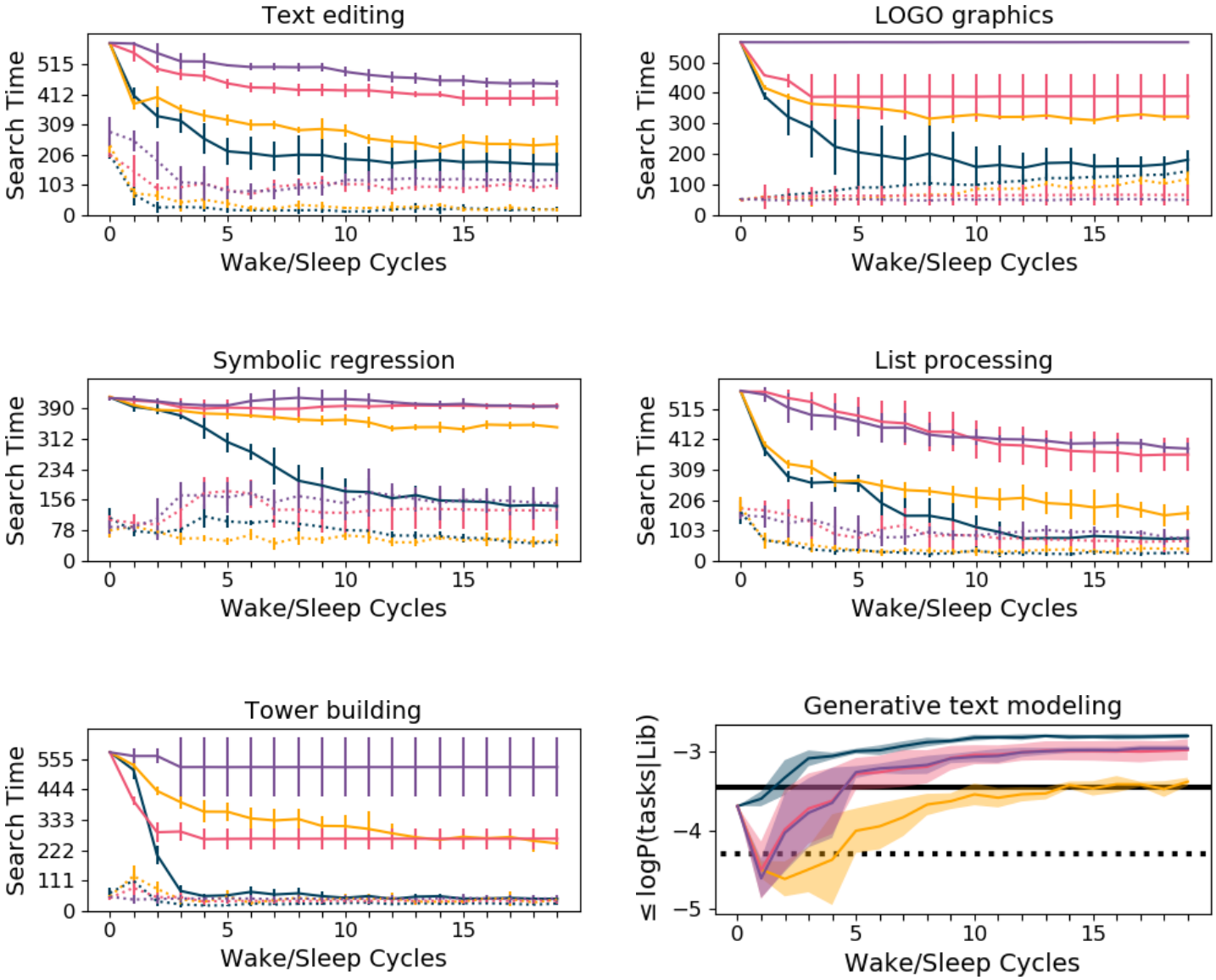


Figure S11: Test set performance across wake/sleep iterations. Error bars over five random seeds. Teal: Full model. Yellow: Dreaming only (no library learning). Pink: Abstraction only (no recognition model). Purple: EC baseline. Search time plots show solid lines (time averaged over all tasks) and dotted lines (time averaged over solved tasks). Generative text modeling plots show lower bound on marginal likelihood of held out tasks, averaged per character. Solid black line on generative text modeling is "enumeration" baseline (24 hours of enumeration with no learning). Dashed black line on generative text modeling is "neural synthesis" baseline (RobustFill trained for 24 hours).

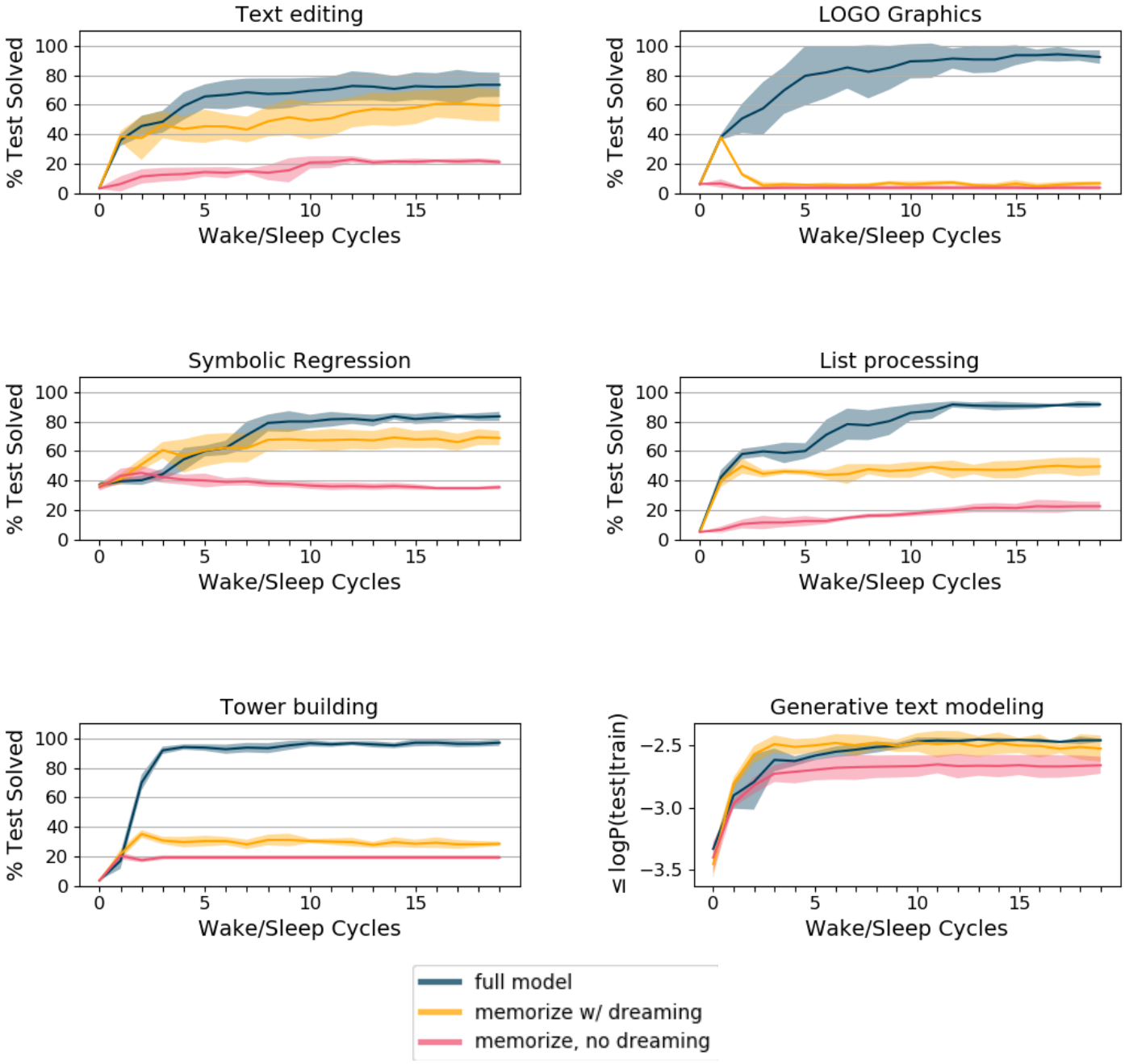


Figure S12: Comparing DreamCoder with approaches which build libraries by memorizing entire programs wholesale. Error bars over five random seeds. Teal: Full model. Yellow: Model which memorizes task solutions wholesale, and which trains a neural recognition model. Pink: Model which memorizes task solutions wholesale, and which does not train a recognition model.

S3 Technical background and relations to prior work

DreamCoder builds on several generations of research in AI, program synthesis, and cognitive science, with program induction being one of the oldest theoretical frameworks for concept learning within artificial intelligence (*42*), and the conceptually allied 'Language of Thought Hypothesis' being almost as old (*10*). Our technical contributions draw on recent neural network approaches for guiding program synthesis as well as symbolic methods for inducing new code subroutines. Our basic motivations and high-level framings are most closely allied with relatively older work on automatic program discovery (within AI) and computational language-of-thought models of learning (within cognitive science).

S3.1 Neurally-guided program synthesis

Recent neural program synthesis systems pair a fixed domain-specific programming language (a 'DSL') to a learned neural network that guides program search (*2, 7, 11*). Our recognition model builds on techniques developed for neurally-guided program synthesis, and is most similar to RobustFill and DeepCoder. DeepCoder works by learning to predict, conditioned on a set of input/output examples (i.e., a task), the probability that each DSL primitive is used in the shortest program consistent with those input/output examples – critically, a DeepCoder neural network only runs once per set of input/output examples, in contrast to systems like RobustFill (*7*), which repeatedly query an autoregressive neural network to predict each program token. Like DeepCoder, DreamCoder's neural recognition model runs once per task, which is advantageous because neural networks are slow compared to symbolic program enumeration. There are two important differences between DeepCoder and our recognition model. First, our recognition model outputs a (generative) distribution over programs. This is a prerequisite for a probabilistic wake/sleep framing. In contrast, DeepCoder does not predict the probability of a program, but instead (discriminatively) classifies whether or not each DSL primitive will be used in the program at least once. Second, as detailed in S4.6, our parameterization of the distribution over programs is not expressed in terms of the probability of using each DSL primitive; instead, the neural network outputs the probability of using each DSL primitive, conditioned on its local syntactic context. This allows the recognition model to provide fine-grained information about the structure of the target program, and also to break syntactic symmetries in the solution space (see S4.6). While the inference-time mechanics of our recognition model is most closely related to DeepCoder, our network's output, as well as its training objective, is more similar to work such as RobustFill (*7*), because, in both cases, the network outputs a distribution over programs, and is trained to maximize the likelihood of training programs.

One could replace our recognition model with any learned neural conditional distribution over programs, such as a Seq2Seq model (such as RobustFill (*7*)) or a Seq2Tree model (*37*). These are reasonable alternatives, and, in principle, such a recognition model could exactly solve the synthesis problem without any search. In practice, these approaches tend to fare poorly at out of sample generalization, which typically requires some kind of combinatorial tree search. Out-of-sample generalization is important for our setting because we assume the learner has access to at most a few hundred tasks. As an example of a hybrid model combining exhaustive search with a sophisticated Seq2Seq model, SketchAdapt (*32*) predicts 'sketches', or program-outlines with unspecified 'holes', which are then filled in through exhaustive enumeration. Such an approach could be combined with the rest of the DreamCoder architecture.

S3.2 Learning new code abstractions

Recent symbolic AI research has developed frameworks for learning DSLs or libraries by inferring reusable pieces of code (*6, 9, 24, 25*). These systems typically work through either memoization (caching reused

subtrees, i.e. (*6, 25*), or reused command sequences, as an (*22*)), or antiunification (caching reused tree-templates that unify with program syntax trees, i.e. (*9, 14, 16*)). Our abstraction sleep algorithm works through automatic refactoring, discovering and incorporating into the library syntax trees which are not present in the surface form of programs available to the learner. Our refactoring process is most similar to Liang et al. (*24*), which uses a tree-shaped datatype they call 'candidate structures' to stochastically explore a set of refactorings; 'candidate structures' are strongly reminiscent of version spaces, and in their work, were used to represent sets of observationally equivalent, yet semantically distinct, solutions to tasks, analogous to version spaces in FlashFill (*13*). We were strongly inspired by Liang et al., and build upon their work by using version spaces to exhaustively enumerate sets of semantically equivalent refactorings. We chose to use version spaces differently from Liang et al. because, for our setting, a stochastic, sampling-based search over refactorings would not scale: recall, from the main text, that we must explore on the order of 10^{14} or more refactorings before finding a single new library component. Thus, we created new algorithms based on version space algebras and equivalence graphs to exhaustively, rather than stochastically, explore the set of refactorings, where each refactoring is not just observationally equivalent but actually semantically identical to the original programs discovered during waking.

Within inductive logic programming, 'predicate invention' is a closely related technique where auxiliary predicates – analogous to subroutines – are automatically introduced while solving a single task (*31*); within genetic programming, 'Automatically Defined Functions' (ADF: (*36*)) play a similar role. Both ADF and predicate invention can be seen as the single-task analogues of the multitask library-learning techniques discussed above. Within *probabilistic* program synthesis, Bayesian program merging (*16*) is an approach for synthesizing probabilistic generative models specified as programs. Hwang et al. (*16*) investigates several refactoring search moves, and these refactorings are used to perform a beam search over the space of probabilistic programs, aiming to find a concise program assigning high likelihood to a data set. These search moves include antiunification — similar to our refactoring — as well as search moves specialized to probabilistic programs, such as 'deargumentation' (see (*16*)). Like ADF and predicate invention, these techniques have mechanics similar to library learning, but differ in their goal, which is to solve a single task with a concise program, rather than sharing code across many programs. Importing techniques from ADF, predicate invention, and Bayesian program merging is an avenue for future research in multitask library learning.

DreamCoder's deeply nested hierarchies of functions that are learned, yet human understandable, has antecedents in the literature on multitask library learning: more broadly, *any* library learning algorithm which iteratively defines new functions in terms of those learned earlier has the ability to construct a deep program representation. Recent examples include EC (*6*), dependent learning (*25*), and Playgol (*5*), which applies dependent learning to imagined problems, or "play problems," intriguingly similar to our dream learning, but applied to a symbolic library instead of a neural recognition network. What distinguishes DreamCoder's deep libraries is not their depth or size, but their interpretability, concision, and abstractness: by searching for the most compressive refactoring under a Bayesian criterion, we recover a modestly-sized set of generally useful routines, in contrast to the thousands of routines learned by e.g. dependent learning on large data sets; and by automatically refactoring, we recover functions which are useful, interpretable, and abstract.

While DreamCoder instantiates one variety of neurosymbolic program induction — in particular, where the programs are symbolic but the inference procedure integrates symbolic and neural components — another variety of neurosymbolic integration is to synthesize programs that themselves interleave learned neural modules (*1*) with symbolic programmatic structure, as recently proposed in the DeepProbLog (*27*) and HOUDINI (*47*). HOUDINI furthermore also reuses these neural modules in a multitask setting (in a manner roughly analogous to dependent learning (*25*)), implementing a neural-module analog of library learning. We see this reurosymbolic integration as complementary to the techniques developed here, and

we anticipate that the use of hybrid neurosymbolic programs will prove vital for synthesizing programs that directly interface with perceptual data.

S3.3 The Exploration-Compression family of algorithms

Our model draws primarily on ideas first elucidated by the Exploration-Compression (EC) family of algorithms (*6, 9, 22, 24, 25, 33, 39, 40, 43*), which alternate between searching, or 'exploring', a space of solutions, and then updating their search procedure by compressing solutions found during exploration. We incorporate several improvements to EC-style algorithms. Our abstraction sleep phase is analogous to a more sophisticated kind of compression that automatically refactors code, unlike EC (*6*) (which memorizes reused subexpressions), or dependent learning (*25*) (which memorizes all subexpressions), or EC^2 (*9*) (which memorizes reused fragments of subexpressions). Our wake phase is analogous to exploration, but is guided by a learned neural recognition model. This recognition model captures procedural aspects of domain knowledge, which human experts need, and which we show our algorithm needs as well.

DreamCoder directly descends from EC^2 (*9*), which in turn directly descended from EC (*6*). *Quantitatively*, DreamCoder outperforms EC^2 on EC^2's own set of example domains, both converging after less time spent searching for programs (typically $6\times$ faster on EC^2's benchmarks, analyzed below under differences in "waking"), while also solving more held-out test tasks: EC^2 solves 91.5% list processing problems vs. DreamCoder's median of 94%; EC^2 solves 74% text editing problems vs. DreamCoder solving 84.3%. *Qualitatively*, the differences between DreamCoder and EC^2 are:

- During waking, we sample a random minibatch of tasks, rather than trying to solve all of the tasks at once, which we found was important for scaling to large sets of tasks. For example, DreamCoder converges after 10 wake/sleep cycles for text editing and list processing, consuming 2 hours of program search time (on 64 CPUs), compared to 3 full-batch cycles for EC^2 (*9*), consuming 6 hours of program search time (on 128 CPUs), or using $6\times$ of the search time DreamCoder uses. For symbolic regression, DreamCoder also converges within 10 wake/sleep cycles with a time out of 2 minutes, consuming 200 minutes of program search time (on 20 CPUs), compared to 3 full-batch cycles for EC^2 (*9*), consuming 1500 minutes of program search time (on 40 CPUs), or using $15\times$ of the search time DreamCoder uses.

- When updating the library of code, DreamCoder uses a new algorithm to refactor the programs found during waking. The development of our new refactoring algorithm was motivated by the failure of EC^2 to learn libraries like those we describe in sections S2.1.8 and S2.1.7 of the main paper, as well as the failure of EC^2 to learn higher-order concepts like 'radial symmetry' when tasked with our LOGO graphics tasks.

- When training and using the neural recognition model, DreamCoder combines a new parameterization of its recognition model with a new training objective. Contrasting our neural-network-only ablation (Fig. 6 of main paper) with EC^2's neural-network-only ablation (see (*9*), Table 3), one sees that, for DreamCoder, the neural network by itself is always at least as effective as library learning by itself — yet for EC^2, the opposite is true.

S3.4 Automated program discovery

Automatically discovering programs, models, or theorems has a long history within classic AI; prominent examples include the Automated Mathematician (AM), Eurisko (*23*), HACKER (*44*), and BACON (*21*). In these works, an agent automatically explores a space of concepts expressed as programs, guided by

hand-coded heuristics, or by symbolic heuristics that the system itself discovers via heuristic search. One can see DreamCoder's recognition model as playing an analogous role to these search heuristics. Where these systems work best is when there is a tight match between the syntactic representation of the solution space and the semantic space of solutions. This principle is seen most clearly in the success of AM, and the relative lack of success in Eurisko: in his paper "Why AM and Eurisko Appear to Work," Lenat et al. (*23*) writes that AM's "chief source of power" comes from "exploiting the natural tie between Lisp and mathematics"; such a natural tie is not exist between out-of-the-box Lisp and most other domains, however, for which Lisp has "syntactic distinctions that don't reflect semantic distinctions" (*23*). DreamCoder's deep library learning can be seen as an answer to how a program-writing system can autonomously adapt its basis to the problem-solving domain.

S3.5 Computational models of the Language-of-Thought

The idea that mental representation of concepts assumes program-like forms—namely, that it is both symbolic and recursively compositional—is termed the 'language-of-thought' hypothesis (*10*). Motivated by this hypothesis, work in cognitive modeling has often exploited program-like representations for both concept (*12, 20, 34*) and intuitive theory learning (*17, 46*), and we view these models of concept learning as a kind of program induction, and see theory learning as similar to inducing a library. The chief advantage of a language-of-thought model, in contrast to a symbolic, yet nonrecursive or propositional representation, is expressivity: just as natural languages may express an infinite number of meanings through the combinatorial use of a finite inventory of atoms, a programmatic language-of-thought can encode an infinite number of concepts; in the limit, with a Turing-complete language-of-thought, any computable concept may be expressed. The advantage of a language-of-thought over distributed, connectionist representations—which in many cases are also computationally universal—is that a symbolic language of thought is systematic and modular, hence more interpretable and reusable (as shown in our library learning); and that symbolic program learning coupled to a description-length prior affords data efficient learning, an observation first made by Solomonoff (*42*). The disadvantage of a computational language-of-thought approach is that learning is made much more computationally difficult, a problem that we sought to address with the work here.

S4 Algorithmic and implementation details

S4.1 Probabilistic Formulation of DreamCoder

Our objective is to infer the maximum a posteriori L conditioned on tasks X while marginalizing over the program solving each task $\{\rho_x\}_{x\in X}$. The generative model over programs, L, consists of a set of λ-calculus expressions, written D, as well as a real-valued weight vector written θ, where the dimensionality of θ is $|D|+1$. These correspond to the discrete structure (D) and the continuous parameters (θ) of the prior over programs $L=(D,\theta)$. Writing J for the joint probability of the library and tasks, MAP inference corresponds to solving

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

$$D^* = \arg\max_D \int J(D,\theta)\,\mathrm{d}\theta \qquad \theta^* = \arg\max_\theta J(D^*,\theta) \tag{1}$$

where $\mathrm{P}[x|\rho]$ scores the likelihood of a task $x \in X$ given a program ρ.[2]

Evaluating Eq. 1 entails marginalizing over the infinite set of all programs – which is impossible. We make a particle-based approximation to Eq. 1 and instead marginalize over a finite **beam** of programs, with one beam per task, collectively written $\{\mathcal{B}_x\}_{x\in X}$. This particle-based approximation is written $\mathscr{L}(D,\theta,\{\mathcal{B}_x\})$ and acts as a lower bound on the joint density:[3]

$$J(D,\theta) \geq \mathscr{L}(D,\theta,\{\mathcal{B}_x\}) \triangleq \mathrm{P}[D,\theta] \prod_{x\in X} \sum_{\rho\in\mathcal{B}_x} \mathrm{P}[x|\rho]\mathrm{P}[\rho|D,\theta], \text{ where } |\mathcal{B}_x| \text{ is small} \tag{2}$$

In all of our experiments we set the maximum beam size $|\mathcal{B}_x|$ to 5.

Wake and sleep cycles correspond to alternate maximization of $\mathscr{L}$ w.r.t. $\{\mathcal{B}_x\}_{x\in X}$ (**Wake**) and (D,θ) (**Abstraction**):

Wake: Maxing $\mathscr{L}$ w.r.t. the beams. Here (D,θ) is fixed and we want to find new programs to add to the beams so that $\mathscr{L}$ increases the most. $\mathscr{L}$ most increases by finding programs where $\mathrm{P}[x|\rho]\mathrm{P}[\rho|D,\theta] \propto \mathrm{P}[\rho|x,D,\theta]$ is large, i.e., programs with high posterior probability, which is the search objective during waking.

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

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

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

S4.2 DreamCoder pseudocode

Algorithm 1 specifies how we integrate wake and sleep cycles.

S4.3 Generative model over programs

Algorithm 2 gives a stochastic procedure for drawing samples from $\mathrm{P}[\cdot|\mathcal{D},\theta]$. It takes as input the desired type of the unknown program, and performs type inference during sampling to ensure that the program has the desired type. It also maintains a *environment* mapping variables to types, which ensures that lexical scoping rules are obeyed.

[2]For example, for list processing, the likelihood is 1 if the program predicts the observed outputs on the observed inputs, and 0 otherwise; when learning a generative model or probabilistic program, the likelihood is the probability of the program sampling the observation.

[3]One might be tempted to construct the ELBo bound by defining variational distributions $Q_x(\rho) \propto \mathbb{1}\left[\rho \in \mathcal{B}_x\right] \mathrm{P}\left[x,\rho|D,\theta\right]$ and maximize $\mathrm{ELBo} = \log \mathrm{P}[D,\theta] + \sum_x \mathrm{E}_{Q_x}\left[\log \mathrm{P}[x,\rho|D,\theta]\right]$. But the bound we have defined, $\mathscr{L}$, is tighter than this ELBo: $\log\mathscr{L} = \log \mathrm{P}[D,\theta] + \sum_x \log \mathrm{E}_{Q_x}\left[\mathrm{P}[x,\rho|D,\theta]/Q_x(\rho)\right] \geq \log \mathrm{P}[D,\theta] + \sum_x \mathrm{E}_{Q_x}\left[\log \mathrm{P}[x,\rho|D,\theta]/Q_x(\rho)\right]$ (by Jensen's inequality) which is $\log \mathrm{P}[D,\theta] + \sum_x \mathrm{E}_{Q_x}\left[\log \mathrm{P}[x,\rho|D,\theta]\right] + \mathrm{E}_{Q_x}\left[-\log Q_x\right] = \mathrm{ELBo} + \sum_x \mathrm{H}\left[Q_x\right] \geq \mathrm{ELBo}$ (by nonnegativity of entropy $H\left[\cdot\right]$).

Algorithm 1 Full DreamCoder algorithm

function DreamCoder(D, X):
Input: Initial library functions D, tasks X
Output: Infinite stream of libraries, recognition models, and beams
Hyperparameters: Batch size B, enumeration timeout T, maximum beam size M
$\theta \leftarrow$uniform distribution
$\mathcal{B}_x \leftarrow \varnothing, \forall x \in X$ ▷ Initialize beams to be empty
while true **do** ▷ Loop over epochs
    shuffle $\leftarrow$ random permutation of X ▷ Randomize minibatches
    while shuffle is not empty **do** ▷ Loop over minibatches
        batch $\leftarrow$ first B elements of shuffle ▷ Next minibatch of tasks
        shuffle $\leftarrow$ shuffle with first B elements removed
        $\forall x \in$ batch: $\mathcal{B}_x \leftarrow \mathcal{B}_x \cup \{\rho \mid \rho \in \text{enumerate}(\text{P}[\cdot|D,\theta], T) \text{ if } \text{P}[x|\rho] > 0\}$ ▷ Wake
        Train $Q(\cdot|\cdot)$ to minimize $\mathcal{L}^{\text{MAP}}$ across all $\{\mathcal{B}_x\}_{x\in X}$ ▷ Dream Sleep
        $\forall x \in$ batch: $\mathcal{B}_x \leftarrow \mathcal{B}_x \cup \{\rho \mid \rho \in \text{enumerate}(Q(\cdot|x), T) \text{ if } \text{P}[x|\rho] > 0\}$ ▷ Wake
        $\forall x \in$ batch: $\mathcal{B}_x \leftarrow$ top M elements of $\mathcal{B}_x$ as measured by $\text{P}[\cdot|x, D, \theta]$ ▷ Keep top M programs
        $D, \theta, \{\mathcal{B}_x\}_{x\in X} \leftarrow$ABSTRACTION$(D, \theta, \{\mathcal{B}_x\}_{x\in X})$ ▷ Abstraction Sleep
        yield (D, θ), Q, $\{\mathcal{B}_x\}_{x\in X}$ ▷ Yield the updated library, recognition model, and solutions found to tasks
    end while
end while

S4.4 Enumerative program search

Our implementation of DreamCoder takes the simple and generic strategy of enumerating programs in descending order of their probability under either (D, θ) or $Q(\rho|x)$. Algorithm 2 and 6 specify procedures for sampling from these distributions, but not for enumerating from them. We combine two different enumeration strategies, which allowed us to build a massively parallel program enumerator:

- **Best-first search:** Best-first search maintains a heap of partial programs ordered by their probability — here a partial program means a program whose syntax tree may contain unspecified 'holes'. Best-first search is guaranteed to enumerate programs in decreasing order of their probability, and has memory requirements that in general grow exponentially as a function of the description length of programs in the heap (thus linearly as a function of run time).

- **Depth-first search:** Depth first search recursively explores the space of execution traces through Algorithm 2 and 6, equivalently maintaining a stack of partial programs. In general it does not enumerate programs in decreasing order of probability, but has memory requirements that grow linearly as a function of the description length of the programs in the stack (thus logarithmically as a function of run time).

Our parallel enumeration algorithm (Algorithm 3) first performs a best-first search until the best-first heap is much larger than the number of CPUs. At this point, it switches to performing many depth-first searches in parallel, initializing a depth first search with one of the entries in the best-first heap. Because depth-first search does not produce programs in decreasing order of their probability, we wrap this entire procedure up into an outer loop that first enumerates programs whose description length is between 0 to Δ, then programs with description length between Δ and 2Δ, then 2Δ to 3Δ, etc., until a timeout is reached. This is similar in spirit to iterative deepening depth first search (*38*).

Algorithm 2 Generative model over programs

function sample(D, θ, τ):
Input: Library (D, θ), type τ
Output: a program whose type unifies with τ
return sample$'(D, \theta, \varnothing, \tau)$

function sample$'(D, \theta, \mathcal{E}, \tau)$:
Input: Library (D, θ), environment $\mathcal{E}$, type τ ▷ Environment $\mathcal{E}$ starts out as $\varnothing$
Output: a program whose type unifies with τ
if $\tau = \alpha \to \beta$ **then** ▷ Function type — start with a lambda
    var $\leftarrow$ an unused variable name
    body $\sim$ sample$'(D, \theta, \{\text{var} : \alpha\} \cup \mathcal{E}, \beta)$ ▷ Recursively sample function body
    return `(lambda (var) body)`
else ▷ Build an application to give something w/ type τ
    primitives $\leftarrow \{\rho | \rho : \tau' \in D \cup \mathcal{E}$ if τ can unify with yield$(\tau')\}$ ▷ Everything in scope w/ type τ
    variables $\leftarrow \{\rho \mid \rho \in$ primitives and ρ a variable$\}$
    Draw $e \sim$ primitives, w.p. $\propto \begin{cases} \theta_e & \text{if } e \in D \\ \theta_{var}/|\text{variables}| & \text{if } e \in \mathcal{E} \end{cases}$
    Unify τ with yield(τ'). ▷ Ensure well-typed program
    $\{\alpha_k\}_{k=1}^{K} \leftarrow \text{args}(\tau')$
    for $k = 1$ **to** K **do** ▷ Recursively sample arguments
        $a_k \sim$ sample$'(D, \theta, \mathcal{E}, \alpha_k)$
    end for
    return $(e\ a_1\ a_2\ \cdots\ a_K)$
end if

where:

yield$(\tau) = \begin{cases} \text{yield}(\beta) & \text{if } \tau = \alpha \to \beta \\ \tau & \text{otherwise.} \end{cases}$ ▷ Final return type of τ

args$(\tau) = \begin{cases} [\alpha] + \text{args}(\beta) & \text{if } \tau = \alpha \to \beta \\ [] & \text{otherwise.} \end{cases}$ ▷ Types of arguments needed to get something w/ type τ

Algorithm 3 Parallel enumerative program search algorithm

function enumerate($\mu, T,$ CPUs):
Input: Distribution over programs μ, timeout T, CPU count
Output: stream of programs in approximately descending order of probability under μ
Hyperparameter: nat increase rate Δ ▷ We set $\Delta = 1.5$
lowerBound$\leftarrow 0$
while total elapsed time $< T$ **do**
    heap$\leftarrow$newMaxHeap() ▷ Heap for best-first search
    heap.insert(priority $= 0$, value $=$ empty syntax tree) ▷ Initialize heap with start state of search space
    while $0 <$ |heap| $\leq 10 \times$ CPUs **do** ▷ Each CPU gets approximately 10 jobs (a partial program)
        priority, partialProgram $\leftarrow$ heap.popMaximum()
        if partialProgram is finished **then** ▷ Nothing more to fill in in the syntax tree
            if lowerBound $\leq -$priority $<$ lowerBound $+ \Delta$ **then**
                yield partialProgram
            end if
        else
            for child$\in$children(partialProgram) **do** ▷ children$(\cdot)$ fills in next 'hole' in syntax tree
                if $-\log \mu$(child) $<$ lowerBound $+ \Delta$ **then** ▷ Child's description length small enough
                    heap.insert(priority $= \log \mu$(child), value $=$ child)
                end if
            end for
        end if
    end while
    yield from ParallelMap$_{\text{CPUs}}$(depthFirst($\mu, T -$ elapsed time, lowerBound, $\cdot$), heap.values())
    lowerBound $\leftarrow$ lowerBound $+ \Delta$ ▷ Push up lower bound on MDL by Δ
end while

function depthFirst($\mu, T,$ lowerBound, partialProgram): ▷ Each worker does a depth first search. Enumerates completions of partialProgram whose MDL is between lowerBound and lowerBound $+ \Delta$
stack$\leftarrow$[partialProgram]
while total elapsed time $< T$ and stack is not empty **do**
    partialProgram$\leftarrow$stack.pop()
    if partialProgram is finished **then**
        if lowerBound $\leq -\log \mu$(partialProgram) $<$ lowerBound $+ \Delta$ **then**
            yield partialProgram
        end if
    else
        for child $\in$ children(partialProgram) **do**
            if $-\log \mu$(child) $<$ lowerBound $+ \Delta$ **then** ▷ Child's description length small enough
                stack.push(child)
            end if
        end for
    end if
end while

S4.5 Abstraction sleep (library learning)

Algorithm 4 specifies our library learning procedure. This integrates two toolkits: the machinery of version spaces and equivalence graphs (Appendix S4.5.1) along with a probabilistic objective favoring compressive libraries. The functions $I\beta(\cdot)$ and REFACTOR construct a version space from a program and extract the shortest program from a version space, respectively (Algorithm 4, lines 5-6, 14; Appendix S4.5.1). To define the prior distribution over (D, θ) (Algorithm 4, lines 7-8), we penalize the syntactic complexity of the λ-calculus expressions in the library, defining $\mathrm{P}[D] \propto \exp(-\lambda \sum_{\rho \in D} \mathrm{size}(\rho))$ where $\mathrm{size}(\rho)$ measures the size of the syntax tree of program p, and λ controls how strongly we regularize the size of the library. We place a symmetric Dirichlet prior over the weight vector θ.

To appropriately score each proposed D we must reestimate the weight vector θ (Algorithm 4, line 7). Although this may seem very similar to estimating the parameters of a probabilistic context free grammar, for which we have effective approaches like the Inside/Outside algorithm (*19*), our distribution over programs is context-sensitive due to the presence of variables in the programs and also due to the polymorphic typing system. Appendix S4.5.4 derives a tractable MAP estimator for θ.

Algorithm 4 Library Induction Algorithm

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

S4.5.1 Refactoring code with version spaces

Intuitively, a version space is a tree-shaped data structure that compactly represents a large set of programs and supports efficient set operations like union, intersection, and membership checking. Figure S13 (left) diagrams a subset of a version space containing refactorings of a program calculating $1 + 1$, a representation exponentially more efficient than explicitly enumerating the set of refactorings (Figure S13 right).

Formally, a version space is either:

- A deBuijn[4] index: written $\$i$, where i is a natural number

[4]deBuijn indices are an alternative way of naming variables in λ-calculus. When using deBuijn indices, λ-abstractions are written *without* a variable name, and variables are written as the count of the number of λ-abstractions up in the syntax tree the variable is bound to. For example, $\lambda x.\lambda y.(x\ y)$ is written $\lambda\lambda(\$1\ \$0)$ using deBuijn indices. See (*35*) for more details.

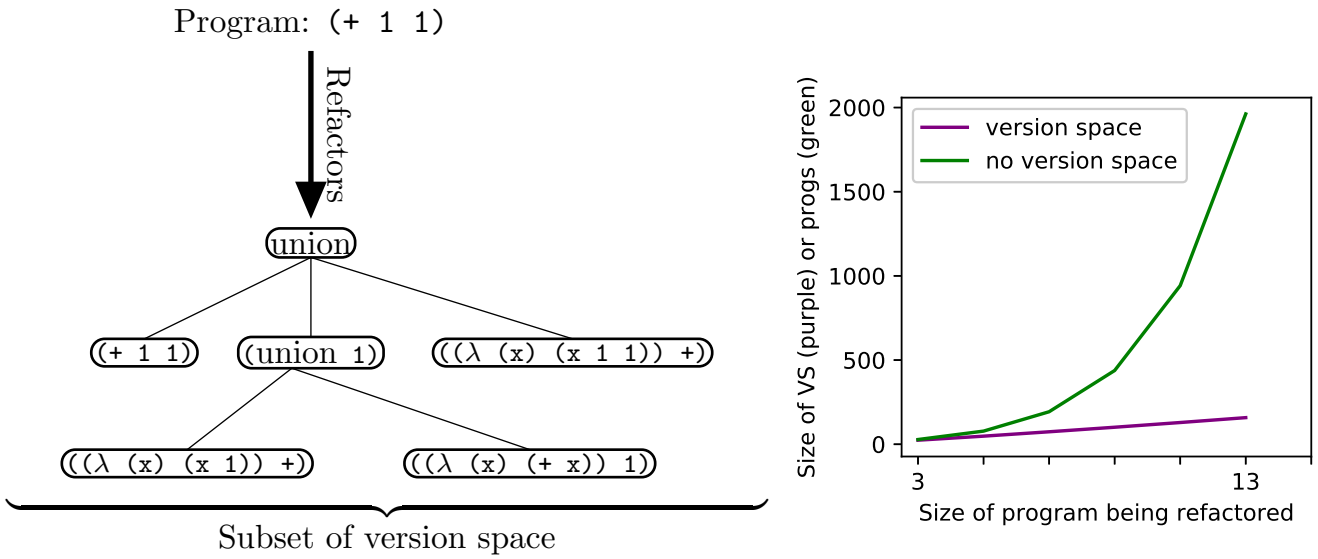


Figure S13: # of possible refactorings grows exponentially with program size, so we represent refactorings using version spaces, which augment syntax trees with a *union* operator whose children are themselves version spaces. Right graph: version spaces are exponentially more efficient than explicitly constructing set of refactorings. In this graph, refactored programs are of the form $1 + 1 + \cdots + 1$.

- An abstraction: written λv, where v is a version space
- An application: written $(f\ x)$, where both f and x are version spaces
- A union: $\uplus V$, where V is a set of version spaces
- The empty set, $\varnothing$
- The set of all λ-calculus expressions, Λ

The purpose of a version space to compactly represent a set of programs. We refer to this set as the **extension** of the version space:

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

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

Version spaces also support efficient membership checking, which we write as $e \in [\![v]\!]$. Important for our purposes, it is also efficient to refactor the members of a version space's extension in terms of a new library. We define $\text{REFACTOR}(v|D)$ inductively as:

$$\text{REFACTOR}(v|D) = \begin{cases} e, \text{ if } e \in D \text{ and } e \in [\![v]\!]. \text{ Exploits the fact that } [\![e]\!] \in v \text{ can be efficiently computed.} \\ \text{REFACTOR}'(v|D), \text{ otherwise.} \end{cases}$$

$$\text{REFACTOR}'(e|D) = e, \text{ if } e \text{ is a leaf} \qquad \text{REFACTOR}'(\lambda b|D) = \lambda \text{REFACTOR}(b|D)$$
$$\text{REFACTOR}'(f\ x|D) = \text{REFACTOR}(f|D)\ \text{REFACTOR}(x|D) \qquad \text{REFACTOR}'(\uplus V|D) = \underset{e \in \{\text{REFACTOR}(v|D)\ :\ v \in V\}}{\arg\min} \text{size}(e|D)$$

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

Recall that our goal is to define an operator over version spaces, $I\beta_n$, which calculates the set of n-step refactorings of a program p, e.g., the set of all programs p' where $p' \underbrace{\longrightarrow q \longrightarrow \cdots \longrightarrow}_{\leq n \text{ times}} p$, where $a \longrightarrow b$ is the standard notation for a rewriting to b according to the standard rewrite rules of λ-calculus (*35*).

We define this operator in terms of another operator, $I\beta'$, which performs a single step of refactoring:

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

where

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

where we have defined $I\beta'$ in terms of another operator, $S_k : \text{VS} \to 2^{\text{VS} \times \text{VS}}$, whose purpose is to construct the set of substitutions that are refactorings of a program in a version space. We define S as:

$$S_k(v) = \left\{\downarrow_0^k v \mapsto \$k\right\} \cup \begin{cases} \text{if } v \text{ is primitive:} & \{\Lambda \mapsto v\} \\ \text{if } v = \$i \text{ and } i < k\text{:} & \{\Lambda \mapsto \$i\} \\ \text{if } v = \$i \text{ and } i \geq k\text{:} & \{\Lambda \mapsto \$(i+1)\} \\ \text{if } v = \lambda b\text{:} & \{v' \mapsto \lambda b' \ : \ v' \mapsto b' \in S_{k+1}(b)\} \\ \text{if } v = (f\ x)\text{:} & \{v_1 \cap v_2 \mapsto (f'\ x') \ : \ v_1 \mapsto f' \in S_k(f),\ v_2 \mapsto x' \in S_k(x)\} \\ \text{if } v = \uplus V\text{:} & \bigcup_{v' \in V} S_n(v') \\ \text{if } v \text{ is } \varnothing\text{:} & \varnothing \\ \text{if } v \text{ is } \Lambda\text{:} & \{\Lambda \mapsto \Lambda\} \end{cases}$$

$$\downarrow_c^k \$i = \$i, \text{ when } i < c$$
$$\downarrow_c^k \$i = \$(i-k), \text{ when } i \geq c+k$$
$$\downarrow_c^k \$i = \varnothing, \text{ when } c \leq i < c+k$$
$$\downarrow_c^k \lambda b = \lambda \downarrow_{c+1}^k b$$
$$\downarrow_c^k (f\ x) = (\downarrow_c^k f\ \downarrow_c^k x)$$
$$\downarrow_c^k \uplus V = \uplus \left\{\downarrow_c^k v \mid v \in V\right\}$$
$$\downarrow_c^k v = v, \text{ when } v \text{ is a primitive or } \varnothing \text{ or } \Lambda$$

where $\uparrow^k$ is the shifting operator (*35*), which adds k to all of the free variables in a λ-expression or version space, and we have defined a new operator, $\downarrow$, whose purpose is to undo the action of $\uparrow$. We have written definitions recursively, but implement them using a dynamic program: we hash cons each version space, and only calculate the operators $I\beta_n$, $I\beta'$, and S_k once per each version space.

We now formally prove that $I\beta_n$ exhaustively enumerates the space of possible refactorings. Our approach is to first prove that S_k exhaustively enumerates the space of possible substitutions that could give

rise to a program. The following pair of technical lemmas are useful; both are easily proven by structural induction.

Lemma 1. *Let e be a program or version space and n, c be natural numbers.*
Then $\uparrow_{n+c}^{-1}\uparrow_c^{n+1} e = \uparrow_c^n e$, and in particular $\uparrow_n^{-1}\uparrow^{n+1} e = \uparrow^n e$.

Lemma 2. *Let e be a program or version space and n, c be natural numbers.*
Then $\downarrow_c^n\uparrow_c^n e = e$, and in particular $\downarrow^n\uparrow^n e = e$.

Theorem 1. ***Consistency of*** S_n.
If $(v \mapsto b) \in S_n(u)$ then for every $v' \in v$ and $b' \in b$ we have $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']b' \in u$.

Proof. Suppose $b = \$n$ and therefore, by the definition of S_n, also $v = \downarrow_0^n u$. Invoking Lemmas 1 and 2 we know that $u = \uparrow_n^{-1}\uparrow^{n+1} v$ and so for every $v' \in v$ we have $\uparrow_n^{-1}\uparrow^{n+1} v' \in u$. Because $b = \$n = b'$ we can rewrite this to $\uparrow_n^{-1} [\$n \mapsto \uparrow^{n+1} v']b' \in u$.

Otherwise assume $b \neq \$n$ and proceed by structural induction on u:

- If $u = \$i < n$ then we have to consider the case that $v = \Lambda$ and $b = u = \$i = b'$. Pick $v' \in \Lambda$ arbitrarily. Then $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']b' = \uparrow_n^{-1} \$i = \$i \in u$.

- If $u = \$i \geq n$ then we have consider the case that $v = \Lambda$ and $b = \$(i+1) = b'$. Pick $v' \in \Lambda$ arbitrarily. Then $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']b' = \uparrow_n^{-1} \$(i+1) = \$i \in u$.

- If u is primitive then we have to consider the case that $v = \Lambda$ and $b = u = b'$. Pick $v' \in \Lambda$ arbitrarily. Then $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']b' = \uparrow_n^{-1} u = u \in u$.

- If u is of the form λa, then $S_n(u) \subset \{v \mapsto \lambda b \mid (v \mapsto b) \in S_{n+1}(a)\}$. Let $v \mapsto \lambda b \in S_n(u)$. By induction for every $v' \in v$ and $b' \in b$ we have $\uparrow_{n+1}^{-1} [\$n \mapsto \uparrow^{2+n} v']b' \in a$, which we can rewrite to $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']\lambda b' \in \lambda a = u$.

- If u is of the form $(f\ x)$ then then $S_n(u) \subset \{v_f \cap v_x \mapsto (b_f\ b_x) \mid (v_f \mapsto b_f) \in S_n(f),\ (v_x \mapsto b_x) \in S_n(x)\}$. Pick $v' \in v_f \cap v_x$ arbitrarily. By induction for every $v'_f \in v_f$, $v'_x \in v_x$, $b'_f \in b_f$, $b'_x \in b_x$ we have $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v'_f]b'_f \in f$ and $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v'_x]b'_x \in x$. Combining these facts gives $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v'](b'_f\ b'_x) \in (f\ x) = u$.

- If u is of the form $\uplus U$ then pick $(v \mapsto b) \in S_n(u)$ arbitrarily. By the definition of S_n there is a z such that $(v \mapsto b) \in S_n(z)$, and the theorem holds immediately by induction.

- If u is $\varnothing$ or Λ then the theorem holds vacuously.

□

Theorem 2. ***Completeness of*** S_n.
If there exists programs v' and b', and a version space u, such that $\uparrow_n^{-1} [\$n \mapsto \uparrow^{1+n} v']b' \in u$, then there also exists $(v \mapsto b) \in S_n(u)$ such that $v' \in v$ and $b' \in b$.

Proof. As before we first consider the case that $b' = \$n$. If so then $\uparrow_n^{-1}\uparrow^{1+n} v' \in u$ or (invoking Lemma 1) that $\uparrow^n v' \in u$ and (invoking Lemma 2) that $v' \in \downarrow^n u$. From the definition of S_n we know that $(\downarrow^n u \mapsto \$n) \in S_n(u)$ which is what was to be shown.

Otherwise assume that $b' \neq \$n$. Proceeding by structural induction on u:

- If $u = \$i$ then, because b' is not $\$n$, we have $\uparrow_n^{-1} b' = \$i$. Let $b' = \$j$, and so

$$i = \begin{cases} j & \text{if } j < n \\ j - 1 & \text{if } j > n \end{cases}$$

  where $j = n$ is impossible because by assumption $b' \neq \$n$.

  If $j < n$ then $i = j$ and so $u = b'$. By the definition of S_n we have $(\Lambda \mapsto \$i) \in S_n(u)$, completing this inductive step because $v' \in \Lambda$ and $b' \in \$i$. Otherwise assume $j > n$ and so $\$i = \$(j-1) = u$. By the definition of S_n we have $(\Lambda \mapsto \$(i+1)) \in S_n(u)$, completing this inductive step because $v' \in \Lambda$ and $b' = \$j = \$(i+1)$.

- If u is a primitive then, because b' is not $\$n$, we have $\uparrow_n^{-1} b' = u$, and so $b' = u$. By the definition of S_n we have $(\Lambda \mapsto u) \in S_n(u)$ completing this inductive step because $v' \in \Lambda$ and $b' = u$.

- If u is of the form λa then, because of the assumption that $b' \neq \$n$, we know that b' is of the form $\lambda c'$ and that $\lambda \uparrow_{n+1}^{-1} [\$(n+1) \mapsto\uparrow^{2+n} v']c' \in \lambda a$. By induction this means that there is a $(v \mapsto c) \in S_{n+1}(a)$ satisfying $v' \in v$ and $c' \in c$. By the definition of S_n we also know that $(v \mapsto \lambda c) \in S_n(u)$, completing this inductive step because $b' = \lambda c' \in \lambda c$.

- If u is of the form $(f\ x)$ then, because of the assumption that $b' \neq \$n$, we know that b' is of the form $(b'_f\ b'_x)$ and that both $\uparrow_n^{-1} [\$n \mapsto\uparrow^{1+n} v']b'_f \in f$ and $\uparrow_n^{-1} [\$n \mapsto\uparrow^{1+n} v']b'_x \in x$. Invoking the inductive hypothesis twice gives a $(v_f \mapsto b_f) \in S_n(f)$ satisfying $v' \in v_f$, $b'_f \in b_f$ and a $(v_x \mapsto b_x) \in S_n(x)$ satisfying $v' \in v_x$, $b'_x \in b_x$. By the definition of S_n we know that $(v_f \cap v_x \mapsto b_f\ b_x) \in S_n(u)$ completing the inductive step because v' is guaranteed to be in both v_f and v_x and we know that $b' = (b'_f\ b'_x) \in (b_f\ b_x)$.

- If u is of the form $\uplus U$ then there must be a $z \in U$ such that $\uparrow_n^{-1} [\$n \mapsto\uparrow^{1+n} v']b' \in z$. By induction there is a $(v \mapsto b) \in S_n(z)$ such that $v' \in v$ and $b' \in v$. By the definition of S_n we know that $(v \mapsto b)$ is also in $S_n(u)$ completing the inductive step.

- If u is $\varnothing$ or Λ then the theorem holds vacuously.

□

From these results the consistency and completeness of $I\beta_n$ follows:

Theorem 3. ***Consistency of*** $I\beta'$**.**
If $p \in [\![I\beta'(u)]\!]$ *then there exists* $p' \in [\![u]\!]$ *such that* $p \longrightarrow p'$.

Proof. Proceed by structural induction on u. If $p \in [\![I\beta'(u)]\!]$ then, from the definition of $I\beta'$ and $[\![\cdot]\!]$, at least one of the following holds:

- Case $p = (\lambda b')v'$ where $v' \in v$, $b' \in b$, and $v \mapsto b \in S_0(u)$: From the definition of β-reduction we know that $p \longrightarrow\uparrow^{-1} [\$0 \mapsto\uparrow^1 v']b'$. From the consistency of S_n we know that $\uparrow^{-1} [\$0 \mapsto\uparrow^1 v']b' \in u$. Identify $p' =\uparrow^{-1} [\$0 \mapsto\uparrow^1 v']b'$.

- Case $u = \lambda b$ and $p = \lambda b'$ where $b' \in [\![I\beta'(b)]\!]$: By induction there exists $b'' \in [\![b]\!]$ such that $b' \longrightarrow b''$. So $p \longrightarrow \lambda b''$. But $\lambda b'' \in [\![\lambda b]\!] = [\![u]\!]$, so identify $p' = \lambda b''$.

- Case $u = (f\ x)$ and $p = (f'\ x')$ where $f' \in [\![I\beta'(f)]\!]$ and $x' \in [\![x]\!]$: By induction there exists $f'' \in [\![f]\!]$ such that $f' \longrightarrow f''$. So $(f'\ x') \longrightarrow (f''\ x')$. But $(f''\ x') \in [\![(f\ x)]\!] = [\![u]\!]$, so identify $p' = (f''\ x')$.

- Case $u = (f\ x)$ and $p = (f'\ x')$ where $x' \in [\![I\beta'(x)]\!]$ and $f' \in [\![f]\!]$: Symmetric to the previous case.
- Case $u = \uplus U$ and $p \in [\![I\beta'(u')]\!]$ where $u' \in U$: By induction there is a $p' \in [\![u']\!]$ satisfying $p' \longrightarrow p$. But $[\![u']\!] \subseteq [\![u]\!]$, so also $p' \in [\![u]\!]$.
- Case u is an index, primitive, $\varnothing$, or Λ: The theorem holds vacuously.

□

Theorem 4. ***Completeness of*** $I\beta'$**.**
Let $p \longrightarrow p'$ *and* $p' \in [\![u]\!]$*. Then* $p \in [\![I\beta'(u)]\!]$.

Proof. Structural induction on u. If $u = \uplus V$ then there is a $v \in V$ such that $p' \in [\![v]\!]$; by induction on v combined with the definition of $I\beta'$ we have $p \in [\![I\beta'(v)]\!] \subseteq [\![I\beta'(u)]\!]$, which is what we were to show. Otherwise assume that $u \neq \uplus V$.

From the definition of $p \longrightarrow p'$ at least one of these cases must hold:

- Case $p = (\lambda b')v'$ and $p' = \uparrow^{-1} [\$0 \mapsto \uparrow^1 v']b'$: Using the fact that $\uparrow^{-1} [\$0 \mapsto \uparrow^1 v']b' \in [\![u]\!]$, we can invoke the completeness of S_n to construct a $(v \mapsto b) \in S_0(u)$ such that $v' \in [\![v]\!]$ and $b' \in [\![b]\!]$. Combine these facts with the definition of $I\beta'$ to get $p = (\lambda b')v' \in [\![(\lambda b)v]\!] \subseteq I\beta'(u)$.
- Case $p = \lambda b$ and $p' = \lambda b'$ where $b \longrightarrow b'$: Because $p' = \lambda b' \in [\![u]\!]$ and by assumption $u \neq \uplus V$, we know that $u = \lambda v$ and $b' \in [\![v]\!]$. By induction $b \in [\![I\beta'(v)]\!]$. Combine with the definition of $I\beta'$ to get $p = \lambda b \in [\![\lambda I\beta'(v)]\!] \subseteq [\![I\beta'(u)]\!]$.
- Case $p = (f\ x)$ and $p' = (f'\ x)$ where $f \longrightarrow f'$: Because $p' = (f'\ x) \in [\![u]\!]$ and by assumption $u \neq \uplus V$ we know that $u = (a\ b)$ where $f' \in [\![a]\!]$ and $x \in [\![b]\!]$. By induction on a we know $f \in [\![I\beta'(a)]\!]$. Therefore $p = (f\ x) \in [\![(I\beta'(a)\ b)]\!] \subseteq [\![I\beta'((a\ b))]\!] \subseteq [\![I\beta'(u)]\!]$.
- Case $p = (f\ x)$ and $p' = (f\ x')$ where $x \longrightarrow x'$: Symmetric to the previous case.

□

Finally we have our main result:

Theorem 5. ***Consistency and completeness of*** $I\beta_n$**.** *Let* p *and* p' *be programs. Then* $p \underbrace{\longrightarrow q \longrightarrow \cdots \longrightarrow}_{\leq n \text{ times}} p'$ *if and only if* $p \in [\![I\beta_n(p')]\!]$.

Proof. Induction on n.

If $n = 0$ then $[\![I\beta_n(p')]\!] = \{p\}$ and $p = p'$; the theorem holds immediately. Assume $n > 0$.

If $p \underbrace{\longrightarrow q \longrightarrow \cdots \longrightarrow}_{\leq n \text{ times}} p'$ then $q \underbrace{\longrightarrow \cdots \longrightarrow}_{\leq n-1 \text{ times}} p'$; induction on n gives $q \in [\![I\beta_{n-1}(p')]\!]$. Combined with $p \longrightarrow q$ we can invoke the completeness of $I\beta'$ to get $p \in [\![I\beta'(I\beta_{n-1}(p'))]\!] \subset [\![I\beta_n(p')]\!]$.

If $p \in [\![I\beta_n(p')]\!]$ then there exists a $i \leq n$ such that $p \in [\![\underbrace{I\beta'(I\beta'(I\beta'(\cdots}_{i \text{ times}} p')))]\!]$. If $i = 0$ then $p = p'$ and p reduces to p' in $0 \leq n$ steps. Otherwise $i > 0$ and $p \in [\![I\beta'(\underbrace{I\beta'(I\beta'(\cdots}_{i-1 \text{ times}} p'))]\!]$. Invoking the consistency of $I\beta'$ we know that $p \longrightarrow q$ for a program $q \in [\![\underbrace{I\beta'(I\beta'(\cdots}_{i-1 \text{ times}} p'))]\!] \subseteq [\![I\beta_{i-1}(p')]\!]$. By induction $q \underbrace{\longrightarrow \cdots \longrightarrow}_{\leq i-1 \text{ times}} p'$, which combined with $p \longrightarrow q$ gives $p \underbrace{\longrightarrow q \longrightarrow \cdots \longrightarrow}_{\leq i \leq n \text{ times}} p'$. □

S4.5.2 Tracking equivalences

Having introduced the version-space machinery, we now are in a position to define how these version spaces aggregate into a single data structure, one for each program, tracking every equivalence revealed by $I\beta_n$. Observe that every time we calculate $I\beta_n(p)$, we obtain a version space containing expressions semantically equivalent to program p. In line with prior work in program analysis (*45*), we track these equivalences, and finally return a single structure for each program compiling all of these equivalences. Concretely, Algorithm 4 calculates (line 5), for each program p, a version space $I\beta(p)$ defined as

$$I\beta(p) = I\beta_n(p) \uplus \begin{cases} \text{if } p = (f\ x)\text{: } I\beta(f)\ I\beta(x) \\ \text{if } p = \lambda b\text{: } \lambda I\beta(b) \\ \text{if } p \text{ is an index or primitive: } \varnothing \end{cases}$$

where n, the amount of refactoring, is a hyper parameter. We set n to 3 for all experiments with the exception of learning recursive functions from basic Lisp primitives (Appendix S2.1.8), where more involved refactoring was required; for that experiment we set $n = 4$.

S4.5.3 Computational complexity of library learning

How long does each update to the library in Algorithm 4 take? In the worst case, each of the n computations of $I\beta'$ will intersect every version space node constructed so far (from the definition of $S_k(f\ x)$),[5] giving worst-case quadratic growth in the size and time complexity of constructing the version space data structure with each refactoring step, and hence polynomial in program size. Thus, constructing all the version spaces takes time linear in the number of programs (written P) in the beams (Algorithm 4, line 5), and, in the worst case, exponential time as a function of the number of refactoring steps n — but we bound the number of steps to be a small number (in particular $n = 3$; see Figure S14). Writing V for the number of version spaces, this means that V is $O(P2^n)$. The number of proposals (line 10) is linear in the number of distinct version spaces, so is $O(V)$. For each proposal we have to refactor every program (line 6), so this means we spend $O(V^2) = O(P^2 2^n)$ per library update. In practice this quadratic dependence on P (the number of programs) is prohibitively slow. We now describe a linear time approximation to the refactor step in Algorithm 4 based on beam search.

For each version space v we calculate a *beam*, which is a function from a library D to a shortest program in $[\![v]\!]$ using primitives in D. Our strategy will be to only maintain the top B shortest programs in the beam; throughout all of the experiments in this paper, we set $B = 10^6$, and in the limit $B \to \infty$ we recover the exact behavior of REFACTOR. The following recursive equations define how we calculate these beams; the set 'proposals' is defined in line 10 of Algorithm 4, and D is the current library:

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

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

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

[5] In practice, most version spaces will not be intersected, and of those that we do intersect, most will yield $\varnothing$. Here we are just deriving a coarse upper bound on the amount of computation.

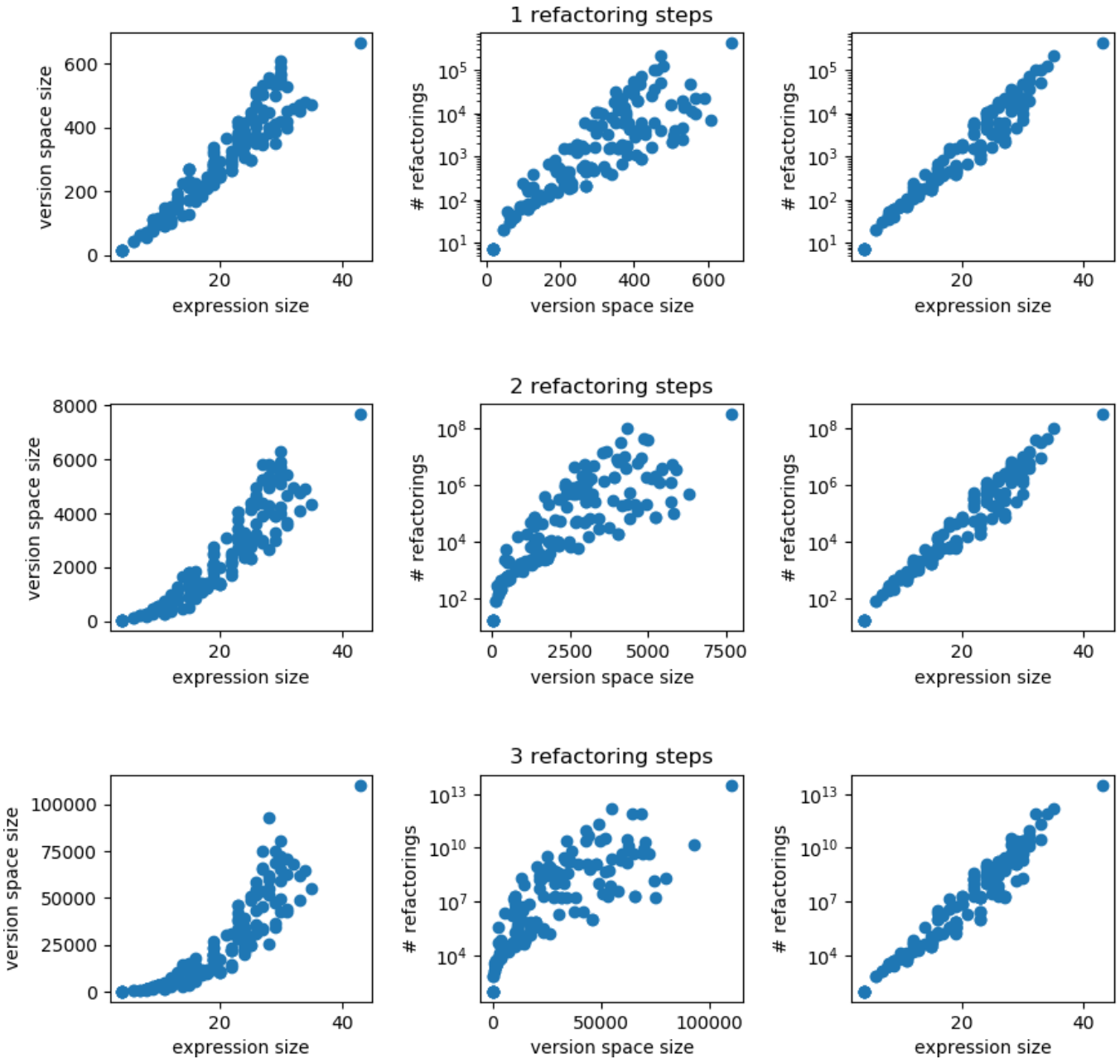


Figure S14: Library learning involves constructing a version space of refactorings of each program. Time and space complexity is proportional to # version spaces, which grows polynomialy with program size and exponentially on the bound on the number of refactoring steps ($\text{size}^{\text{bound}}$). Above we show # version spaces constructed for our list processing programs during the first sleep cycle, as the bound is varied from 1-3 (rows of scatterplots), showing exponential growth in # refactorings (right column), the exponential savings afforded by version spaces (middle column), and polynomial dependence of version space representation on program size (left column). Note log scale on # refactorings.

We calculate $\text{beam}_v(\cdot)$ for each version space using dynamic programming. Using a minheap to represent $\text{beam}_v(\cdot)$, this takes time $O(VB \log B)$, replacing the quadratic dependence on V (and therefore the number of programs, P) with a $B \log B$ term, where the parameter B can be chosen freely, but at the cost of a less accurate beam search.

After performing this beam search, we take only the top I proposals as measured by $-\sum_x \min_{p \in \mathcal{B}_x} \text{beam}_{v_p}(D')$. We set $I = 300$ in all of our experiments, so $I \ll B$. The reason why we only take the top I proposals (rather than take the top B) is because parameter estimation (estimating θ for each proposal) is much more expensive than performing the beam search — so we perform a very wide beam search and then at the very end tim the beam down to only $I = 300$ proposals. Next, we describe our MAP estimator for the continuous parameters (θ) of the generative model.

S4.5.4 Estimating the continuous parameters of the generative model

We use an EM algorithm to estimate the continuous parameters of the generative model, i.e. θ. Suppressing dependencies on D, the EM updates are

$$\theta = \arg\max_{\theta} \log P(\theta) + \sum_x \mathrm{E}_{q_x} \left[\log \mathrm{P}\left[\rho|\theta\right]\right] \tag{3}$$

$$q_x(\rho) \propto \mathrm{P}[x|\rho]\mathrm{P}[\rho|\theta]\mathbb{1}\left[\rho \in \mathcal{B}_x\right] \tag{4}$$

In the M step of EM we will update θ by instead maximizing a lower bound on $\log \mathrm{P}[\rho|\theta]$, making our approach an instance of Generalized EM.

We write $c(e, \rho)$ to mean the number of times that primitive e was used in program ρ; $c(\rho) = \sum_{e \in D} c(e, \rho)$ to mean the total number of primitives used in program ρ; $c(\tau, \rho)$ to mean the number of times that type τ was the input to sample in Algorithm 2 while sampling program ρ. Jensen's inequality gives a lower bound on the likelihood:

$$\sum_x \mathrm{E}_{q_x} \left[\log \mathrm{P}[\rho|\theta]\right] =$$

$$\sum_{e \in D} \log \theta_e \sum_x \mathrm{E}_{q_x}\left[c(e, \rho_x)\right] - \sum_\tau \mathrm{E}_{q_x}\left[\sum_x c(\tau, \rho_x)\right] \log \sum_{\substack{e:\tau' \in D \\ \text{unify}(\tau,\tau')}} \theta_e$$

$$= \sum_e C(e) \log \theta_e - \beta \sum_\tau \frac{\mathrm{E}_{q_x}\left[\sum_x c(\tau, \rho_x)\right]}{\beta} \log \sum_{\substack{e:\tau' \in D \\ \text{unify}(\tau,\tau')}} \theta_e$$

$$\geq \sum_e C(e) \log \theta_e - \beta \log \sum_\tau \frac{\mathrm{E}_{q_x}\left[\sum_x c(\tau, \rho_x)\right]}{\beta} \sum_{\substack{e:\tau' \in D \\ \text{unify}(\tau,\tau')}} \theta_e$$

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

where we have defined

$$C(e) \triangleq \sum_x \mathrm{E}_{q_x}\left[c(e, \rho_x)\right]$$

$$R(\tau) \triangleq \mathrm{E}_{q_x}\left[\sum_x c(\tau, \rho_x)\right]$$

$$\beta \triangleq \sum_\tau \mathrm{E}_{q_x}\left[\sum_x c(\tau, \rho_x)\right]$$

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

$$\sum_e (C(e) + \alpha) \log \theta_e - \beta \log \sum_\tau \frac{R(\tau)}{\beta} \sum_{\substack{e:\tau' \in D \\ \text{unify}(\tau,\tau')}} \theta_e \tag{5}$$

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

$$\frac{C(e) + \alpha}{\theta_e} \propto \sum_{\tau'} \mathbb{1}\left[\text{unify}(\tau, \tau')\right] R(\tau') \tag{6}$$

$$\theta_e \propto \frac{C(e) + \alpha}{\sum_{\tau'} \mathbb{1}\left[\text{unify}(\tau, \tau')\right] R(\tau')} \tag{7}$$

The above is our estimator for θ_e. The above estimator has an intuitive interpretation. The quantity $C(e)$ is the expected number of times that we used e. The quantity $\sum_{\tau'} \mathbb{1}\left[\text{unify}(\tau, \tau')\right] R(\tau')$ is the expected number of times that we *could have* used e. The hyperparameter α acts as pseudocounts that are added to the number of times that we used each primitive, and are not added to the number of times that we could have used each primitive.

We are only maximizing a lower bound on the log posterior; when is this lower bound tight? This lower bound is tight whenever all of the types of the expressions in the library are not polymorphic, in which case our library is equivalent to a PCFG and this estimator is equivalent to the inside/outside algorithm. Polymorphism introduces context-sensitivity to the library, and exactly maximizing the likelihood with respect to θ becomes intractable, which is why we derived the above estimator.

S4.6 Dream sleep (recognition model training)

We define a pair of alternative objectives for the recognition model, $\mathcal{L}^{\text{posterior}}$ and $\mathcal{L}^{\text{MAP}}$, which either train Q to perform full posterior inference or MAP inference, respectively. These objectives combine replays and fantasies:

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

$$\mathcal{L}^{\text{posterior}}_{\text{Replay}} = \mathrm{E}_{x\sim X}\left[\sum_{\rho\in\mathcal{B}_x} \frac{\mathrm{P}\left[x, \rho | D, \theta\right] \log Q(\rho|x)}{\sum_{\rho'\in\mathcal{B}_x} \mathrm{P}\left[x, \rho' | D, \theta\right]}\right] \qquad \mathcal{L}^{\text{MAP}}_{\text{Replay}} = \mathrm{E}_{x\sim X}\left[\max_{\substack{\rho\in\mathcal{B}_x \\ \rho \text{ maxing } \mathrm{P}[\cdot|x,D,\theta]}} \log Q(\rho|x)\right]$$

$$\mathcal{L}^{\text{posterior}}_{\text{Fantasy}} = \mathrm{E}_{(\rho,x)\sim(D,\theta)}\left[\log Q(\rho|x)\right] \qquad \mathcal{L}^{\text{MAP}}_{\text{Fantasy}} = \mathrm{E}_{x\sim(D,\theta)}\left[\max_{\substack{\rho \\ \rho \text{ maxing } \mathrm{P}[\cdot|x,D,\theta]}} \log Q(\rho)\right]$$

The fantasy objectives are essential for data efficiency: all of our experiments train DreamCoder on only a few hundred tasks, which is too little for a high-capacity neural network. Once we bootstrap a (D, θ), we can draw unlimited samples from (D, θ) and train Q on those samples. But, evaluating $\mathcal{L}_{\text{Fantasy}}$ involves drawing programs from the current library, running them to get their outputs, and then training Q to regress from the input/outputs to the program. Since these programs map inputs to outputs, we need to sample the inputs as well. Our solution is to sample the inputs from the empirical observed distribution of inputs in X.

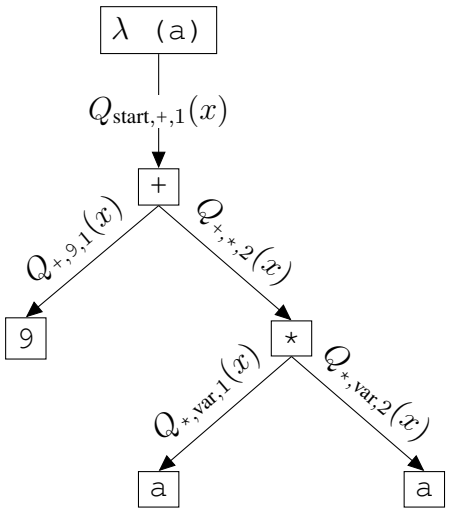
The $\mathcal{L}^{\text{MAP}}_{\text{Fantasy}}$ objective involves finding the MAP program solving a task drawn from the library. To make this tractable, rather than *sample* programs as training data for $\mathcal{L}^{\text{MAP}}_{\text{Fantasy}}$, we *enumerate* programs in decreasing order of their prior probability, tracking, for each dreamed task x, the set of enumerated programs maximizing $\text{P}[x, p|D, \theta]$. Algorithm 5 precisely specifies this process.

Algorithm 5 Dream generation for MAP training objective $\mathcal{L}^{\text{MAP}}$

function generateDreams(L, X, T)
Input: Prior over programs L, training tasks X, timeout T ▷ T set to the same timeout as waking
Output: function which returns dreamed (task, program) pairs
inputs $\leftarrow \{\text{input} \mid (\text{input}, \text{output}) \in x, x \in X\}$ ▷ All attested inputs in training tasks
observedBehaviors $\leftarrow$makeHashTable() ▷ Maps from set of (input,output) to set of programs
for $\rho \in$enumerate$(\text{P}[\cdot|L], T, \text{CPUs} = 1)$ **do** ▷ Draw programs from prior
    b $\leftarrow \{\langle \text{input}, \rho(\text{input})\rangle \mid \text{input} \in \text{inputs}\}$ ▷ Convert program to its observational behavior
    if b $\notin$ observedBehaviors **then** ▷ Is dreamed task new?
        observedBehaviors[b] $\leftarrow \{\rho\}$ ▷ Record new dreamed task
    else if $\forall \rho' \in$ observedBehaviors [b] : $\text{P}[\rho|L] > \text{P}[\rho'|L]$ **then** ▷ Dream supersedes old ones?
        observedBehaviors[b] $\leftarrow \{\rho\}$ ▷ Record new MDL program for task
    else if $\forall \rho' \in$ observedBehaviors [b] : $\text{P}[\rho|L] = \text{P}[\rho'|L]$ **then** ▷ Dream is equivalent MDL solution
        observedBehaviors[b] $\leftarrow \{\rho\} \cup$ observedBehaviors [b]
    else if $\forall \rho' \in$ observedBehaviors [b] : $\text{P}[\rho|L] < \text{P}[\rho'|L]$ **then** ▷ Dream is not MDL solution
        continue ▷ Discard program and go on to the next one
    end if
end for
function sampler() ▷ closure that samples dreams from observedBehaviors
    draw (x, P) from observedBehaviors
    draw ρ from P
    $x' \leftarrow$random subset of input/outputs in x
    return (x', ρ)
end function
return sampler

We maximize $\mathcal{L}^{\text{MAP}}$ rather than $\mathcal{L}^{\text{posterior}}$ for two reasons: $\mathcal{L}^{\text{MAP}}$ prioritizes the shortest program solving a task, thus more strongly accelerating enumerative search during waking; and, combined with our parameterization of Q, described next, we will show that $\mathcal{L}^{\text{MAP}}$ forces the recognition model to break symmetries in the space of programs.

Parameterizing Q. The recognition model predicts a fixed-dimensional tensor encoding a distribution over routines in the library, conditioned on the local context in the syntax tree of the program. This local context consists of the parent node in the syntax tree, as well as which argument is being generated, functioning as a kind of 'bigram' model over trees. Figure S15 (left) diagrams this generative process. This parameterization confers three main advantages: (1) it supports fast enumeration and sampling of programs, because the recognition model only runs once per task, like in (*2, 9, 29*) – thus we can fall back on fast enumeration if the



	Unigram	Bigram

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

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

Figure S15: **Left:** Bigram parameterization of distribution over programs predicted by recognition model. Here the program (syntax tree shown above) is `(λ (a) (+ 9 (* a a )))`. Each conditional distribution predicted by the recognition model is written $Q_{\text{parent,child,argument index}}(x)$, where x is a task. **Right:** Agent learns to break symmetries in program space only when using both bigram parameterization and $\mathcal{L}^{\text{MAP}}$ objective, associating addition to the right and avoiding adding zero. % right-associative calculated by drawing 500 samples from Q. $\mathcal{L}^{\text{MAP}}$/Unigram agent incorrectly learns to never generate programs with `0`'s, while $\mathcal{L}^{\text{MAP}}$/Bigram agent correctly learns that `0` should only be disallowed as an argument of addition. Tasked with building programs from `+`, `1`, and `0`.

target program is unlike the training programs; (2) the recognition model provides fine-grained information about the structure of the target program, similar to (*8, 49*); and (3) in conjunction with $\mathcal{L}^{\text{MAP}}$ the recognition model learns to break symmetries in the space of programs.

Symmetry breaking. Effective domain-specific representations not only exposes high-level building blocks, but also carefully restrict the ways in which those building blocks are allowed to compose. For example, when searching over arithmetic expressions, one could disallow adding zero, and force right-associative addition. A bigram parameterization of the recognition model, combined with the $\mathcal{L}^{\text{MAP}}$ training objective, interact in a way that breaks symmetries like these, allowing the agent to more efficiently explore the solution space. This interaction occurs because the bigram parameterization can disallow library routines depending on their local syntactic context, while the $\mathcal{L}^{\text{MAP}}$ objective forces all probability mass onto a single member of a set of syntactically distinct but semantically equivalent expressions (Appendix S4.6). We experimentally confirm this symmetry-breaking behavior by training recognition models that minimize either $\mathcal{L}^{\text{MAP}}$/$\mathcal{L}^{\text{posterior}}$ and which use either a bigram parameterization/unigram[6] parameterization. Figure S15 (right) shows the result of training Q in these four regimes and then sampling programs. On this particular run, the combination of bigrams and $\mathcal{L}^{\text{MAP}}$ learns to avoid adding zero and associate addition to the right — different random initializations lead to either right or left association.

In formal terms, Q predicts a $(|D|+2) \times (|D|+1) \times A$-dimensional tensor, where A is the maximum arity[7] of any primitive in the library. Slightly abusing notation, we write this tensor as $Q_{ijk}(x)$, where x

[6]In the unigram variant Q predicts a $|D|+1$-dimensional vector: $Q(\rho|x) = \text{P}[\rho|D, \theta_i = Q_i(x)]$, and was used in EC^2

[7]The arity of a function is the number of arguments that it takes as input.

is a task, $i \in D \cup \{\text{start}, \text{var}\}$, $j \in D \cup \{\text{var}\}$, and $k \in \{1, 2, \cdots, A\}$. The output $Q_{ijk}(x)$ controls the probability of sampling primitive j given that i is the parent node in the syntax tree and we are sampling the k^{th} argument. Algorithm 6 specifies a procedure for drawing samples from $Q(\cdot|X)$.

Algorithm 6 Drawing from distribution over programs predicted by recognition model. Compare w/ Algorithm 2

function recognitionSample(Q, x, D, τ):
Input: recognition model Q, task x, library D, type τ
Output: a program whose type unifies with τ
return recognitionSample$'(Q, x, \text{start}, 1, D, \varnothing, \tau)$

function recognitionSample$'(Q, x, \text{parent}, \text{argumentIndex}, D, \mathcal{E}, \tau)$:
Input: recognition model Q, task x, library D, parent $\in D \cup \{\text{start}, \text{var}\}$, argumentIndex $\in \mathbb{N}$, environment $\mathcal{E}$, type τ
Output: a program whose type unifies with τ
if $\tau = \alpha \to \beta$ **then** ▷ Function type — start with a lambda
    var $\leftarrow$ an unused variable name
    body $\sim$ recognitionSample$'(Q, x, \text{parent}, \text{argumentIndex}, D, \{\text{var} : \alpha\} \cup \mathcal{E}, \beta)$
    return `(lambda (`var`)` body`)`
else ▷ Build an application to give something w/ type τ
    primitives $\leftarrow \{\rho | \rho : \tau' \in D \cup \mathcal{E}$ if τ can unify with yield$(\tau')\}$ ▷ Everything in scope w/ type τ
    variables $\leftarrow \{\rho \mid \rho \in$ primitives and ρ a variable$\}$
    Draw $e \sim$ primitives, w.p. $\propto \begin{cases} Q_{\text{parent},e,\text{argumentIndex}}(x) & \text{if } e \in D \\ Q_{\text{parent,var,argumentIndex}}(x)/|\text{variables}| & \text{if } e \in \mathcal{E} \end{cases}$
    Unify τ with yield(τ'). ▷ Ensure well-typed program
    newParent$\leftarrow \begin{cases} e & \text{if } e \in D \\ \text{var} & \text{if } e \in \mathcal{E} \end{cases}$
    $\{\alpha_k\}_{k=1}^{K} \leftarrow \text{args}(\tau')$
    for $k = 1$ **to** K **do** ▷ Recursively sample arguments
        $a_k \sim$ recognitionSample$'(Q, x, \text{newParent}, k, D, \mathcal{E}, \alpha_k)$
    end for
    return $(e\ a_1\ a_2\ \cdots\ a_K)$
end if

We next prove (Theorem 6) that any global optimizer of $\mathcal{L}^{\text{MAP}}$ breaks symmetries, and then give a concrete worked out example contrasting the behavior of $\mathcal{L}^{\text{MAP}}$ and $\mathcal{L}^{\text{posterior}}$.

Theorem 6. *Let $\mu(\cdot)$ be a distribution over tasks and let $Q^*(\cdot|\cdot)$ be a task-conditional distribution over programs satisfying*

$$Q^* = \arg\max_Q \; E_\mu \left[\max_{p \text{ maxing } P[\cdot|x,\mathcal{D},\theta]} \log Q(p|x) \right]$$

where $(\mathcal{D}, \theta)$ is a generative model over programs. Pick a task x where $\mu(x) > 0$. Partition Λ into expressions that are observationally equivalent under x:

$$\Lambda = \bigcup_i \mathcal{E}_i^x \text{ where for any } p_1 \in \mathcal{E}_i^x \text{ and } p_2 \in \mathcal{E}_j^x\text{: } P[x|p_1] = P[x|p_2] \iff i = j$$

Then there exists an equivalence class $\mathcal{E}_i^x$ that gets all the probability mass of Q^* – e.g., $Q^*(p|x) = 0$ whenever $p \notin \mathcal{E}_i^x$ – and there exists a program in that equivalence class which gets all of the probability mass assigned by $Q^*(\cdot|x)$ – e.g., there is a $p \in \mathcal{E}_i^x$ such that $Q^*(p|x) = 1$ – and that program maximizes $P[\cdot|x, \mathcal{D}, \theta]$.

Proof. We proceed by defining the set of "best programs" – programs maximizing the posterior $\mathrm{P}[\cdot|x, \mathcal{D}, \theta]$ – and then showing that a best program satisfies $Q^*(p|x) = 1$. Define the set of best programs $\mathcal{B}_x$ for the task x by

$$\mathcal{B}_x = \left\{ p \mid \mathrm{P}[p|x, \mathcal{D}, \theta] = \max_{p' \in \Lambda} \mathrm{P}[p'|x, \mathcal{D}, \theta] \right\}$$

For convenience define

$$f(Q) = \mathrm{E}_\mu \left[\max_{p \in \mathcal{B}_x} \log Q(p|x) \right]$$

and observe that $Q^* = \arg\max_Q f(Q)$.

Suppose by way of contradiction that there is a $q \notin \mathcal{B}_x$ where $Q^*(q|x) = \epsilon > 0$. Let $p^* = \arg\max_{p \in \mathcal{B}_x} \log Q^*(p|x)$. Define

$$Q'(p|x) = \begin{cases} 0 & \text{if } p = q \\ Q^*(p|x) + \epsilon & \text{if } p = p^* \\ Q^*(p|x) & \text{otherwise.} \end{cases}$$

Then

$$f(Q') - f(Q^*) = \mu(x) \left(\max_{p \in \mathcal{B}_x} \log Q'(p|x) - \max_{p \in \mathcal{B}_x} \log Q^*(p|x) \right) = \mu(x) \left(\log \left(Q^*(p^*|x) + \epsilon \right) - \log Q^*(p^*|x) \right) > 0$$

which contradicts the assumption that Q^* maximizes $f(\cdot)$. Therefore for any $p \notin \mathcal{B}_x$ we have $Q^*(p|x) = 0$.

Suppose by way of contradiction that there are two distinct programs, q and r, both members of $\mathcal{B}_x$, where $Q^*(q|x) = \alpha > 0$ and $Q^*(r|x) = \beta > 0$. Let $p^* = \arg\max_{p \in \mathcal{B}_x} \log Q^*(p|x)$. If $p^* \notin \{q, r\}$ then define

$$Q'(p|x) = \begin{cases} 0 & \text{if } p \in \{q, r\} \\ Q^*(p|x) + \alpha + \beta & \text{if } p = p^* \\ Q^*(p|x) & \text{otherwise.} \end{cases}$$

Then

$$\begin{aligned} f(Q') - f(Q^*) &= \mu(x) \left(\max_{p \in \mathcal{B}_x} \log Q'(p|x) - \max_{p \in \mathcal{B}_x} \log Q^*(p|x) \right) \\ &= \mu(x) \left(\log \left(Q^*(p^*|x) + \alpha + \beta \right) - \log Q^*(p^*|x) \right) > 0 \end{aligned}$$

which contradicts the assumption that Q^* maximizes $f(\cdot)$. Otherwise assume $p^* \in \{q, r\}$. Without loss of generality let $p^* = q$. Define

$$Q'(p|x) = \begin{cases} 0 & \text{if } p = r \\ Q^*(p|x) + \beta & \text{if } p = p^* \\ Q^*(p|x) & \text{otherwise.} \end{cases}$$

Then

$$f(Q') - f(Q^*) = \mu(x) \left(\max_{p \in \mathcal{B}_x} \log Q'(p|x) - \max_{p \in \mathcal{B}_x} \log Q^*(p|x) \right) = \mu(x) \left(\log \left(Q^*(p^*|x) + \beta \right) - \log Q^*(p^*|x) \right) > 0$$

which contradicts the assumption that Q^* maximizes $f(\cdot)$. Therefore $Q^*(p|x) > 0$ for at most one $p \in \mathcal{B}_x$. But we already know that $Q^*(p|x) = 0$ for any $p \notin \mathcal{B}_x$, so it must be the case that $Q^*(\cdot|x)$ places all of its probability mass on exactly one $p \in \mathcal{B}_x$. Call that program p^*.

Because the equivalence classes $\{\mathcal{E}_i^x\}$ form a partition of Λ we know that p^* is a member of exactly one equivalence class; call it $\mathcal{E}_i^x$. Let $q \in \mathcal{E}_j^x \neq \mathcal{E}_i^x$. Then because the equivalence classes form a partition we know that $q \neq p^*$ and so $Q^*(q|x) = 0$, which was our first goal: *any* program not in $\mathcal{E}_i^x$ gets no probability mass.

Our second goal — that there is a member of $\mathcal{E}_i^x$ which gets all the probability mass assigned by $Q^*(\cdot|x)$ — is immediate from $Q^*(p^*|x) = 1$.

Our final goal — that p^* maximizes $\mathrm{P}[\cdot|x, \mathcal{D}, \theta]$ — follows from the fact that $p^* \in \mathcal{B}_x$. □

Notice that Theorem 6 makes no guarantees as to the cross-task systematicity of the symmetry breaking; for example, an optimal recognition model could associate addition to the right for one task and associate addition to the left on another task. *Systematic* breaking of symmetries must arise only as a consequence as the network architecture (i.e., it is more parsimonious to break symmetries the same way for every task than it is to break them differently for each task).

As a concrete example of symmetry breaking, consider an agent tasked with writing programs built from addition and the constants zero and one. A bigram parameterization of Q allows it to represent the fact that it should never add zero ($Q_{+,0,0} = Q_{+,0,1} = 0$) or that addition should always associate to the right ($Q_{+,+,0} = 0$). The $\mathcal{L}^{\text{MAP}}$ training objective encourages learning these canonical forms. Consider two recognition models, Q_1 and Q_2, and two programs in beam $\mathcal{B}_x$, $p_1 =$ `(+ (+ 1 1) 1)` and $p_2 =$ `(+ 1 (+ 1 1))`, where

$$\begin{aligned} Q_1(p_1|x) &= \frac{\epsilon}{2} & Q_1(p_2|x) &= \frac{\epsilon}{2} \\ Q_2(p_1|x) &= 0 & Q_2(p_2|x) &= \epsilon \end{aligned}$$

i.e., Q_2 breaks a symmetry by forcing right associative addition, but Q_1 does not, instead splitting its probability mass equally between p_1 and p_2. Now because $\mathrm{P}[p_1|\mathcal{D}, \theta] = \mathrm{P}[p_2|\mathcal{D}, \theta]$ (Algorithm 2), we have

$$\begin{aligned} \mathcal{L}^{\text{posterior}}_{\text{real}}(Q_1) &= \frac{\mathrm{P}[p_1|\mathcal{D}, \theta] \log \frac{\epsilon}{2} + \mathrm{P}[p_2|\mathcal{D}, \theta] \log \frac{\epsilon}{2}}{\mathrm{P}[p_1|\mathcal{D}, \theta] + \mathrm{P}[p_2|\mathcal{D}, \theta]} = \log \frac{\epsilon}{2} \\ \mathcal{L}^{\text{posterior}}_{\text{real}}(Q_2) &= \frac{\mathrm{P}[p_1|\mathcal{D}, \theta] \log 0 + \mathrm{P}[p_2|\mathcal{D}, \theta] \log \epsilon}{\mathrm{P}[p_1|\mathcal{D}, \theta] + \mathrm{P}[p_2|\mathcal{D}, \theta]} = +\infty \\ \mathcal{L}^{\text{MAP}}_{\text{real}}(Q_1) &= \log Q_1(p_1) = \log Q_1(p_2) = \log \frac{\epsilon}{2} \\ \mathcal{L}^{\text{MAP}}_{\text{real}}(Q_2) &= \log Q_2(p_2) = \log \epsilon \end{aligned}$$

So $\mathcal{L}^{\text{MAP}}$ prefers Q_2 (the symmetry breaking recognition model), while $\mathcal{L}^{\text{posterior}}$ reverses this preference.

How would this example work out if we did not have a bigram parameterization of Q? With a unigram parameterization, Q_2 would be impossible to express, because it depends on local context within the syntax tree of a program. So even though the objective function would prefer symmetry breaking, a simple unigram model lacks the expressive power to encode it.

To be clear, our recognition model does not learn to break *every* possible symmetry in every possible library or DSL. But in practice we found that a bigrams combined with $\mathcal{L}^{\text{MAP}}$ works well, and we use with this combination throughout the paper.

S4.7 Hyperparameters and training details

Neural net architecture The recognition model for domains with sequential structure (list processing, text editing, regular expressions) is a recurrent neural network. We use a bidirectional GRU (*4*) with 64 hidden units that reads each input/output pair; we concatenate the input and output along with a special delimiter symbol between them. We use a 64-dimensional vectors to embed symbols in the input/output. We MaxPool the final hidden unit activations in the GRU along both passes of the bidirectional GRU.

The recognition model for domains with 2D visual structure (LOGO graphics, tower building, and symbolic regression) is a convolutional neural network. We take our convolutional architecture from (*41*).

We follow the RNN/CNN by an MLP with 128 hidden units and a ReLU activation which then outputs the Q_{ijk} matrix described in S4.6.

Neural net training We train our recognition models using Adam (*18*) with a learning rate of 0.001 for 5×10^3 gradient steps or 1 hour, whichever comes first.

Hyperparameters Due to the high computational cost we performed only an informal coarse hyperparameter search. The most important parameter is the enumeration timeout during the wake phase; domains that present more challenging program synthesis problems require either longer timeouts, more CPUs, or both.

Domain	Timeout	CPUs	Batch size	λ (S4.5)	α (S4.5.4)	Beam size (S4.2)
Symbolic regression	2m	20	10	1	30	5
Lists	12m	64	10	1.5	30	5
Text	12m	64	10	1.5	30	5
Regexes	12m	64	10	1.5	30	5
Graphics	1h	96	50	1.5	30	5
Towers	1h	64	50	1.5	30	5
Physical laws	4h	64	batch all unsolved (S2.1.7)	1	30	5
Recursive functions	4h	64	batch all unsolved (S2.1.8)	1	30	5

S4.8 Software Architecture

This section is meant to assist readers who want to use and interpret the DreamCoder software. It should serve as a guide for those who wish to understand, install and run the code for their own experiments, or reuse specific components of the code in future research. Please note that class names and module names are subject to change, but the primary relationships between the different components and a description of the data types are documented below as accurately as possible.

The source code for DreamCoder is available on GitHub at `https://github.com/ellisk42/ec` and technical documentation including installation details exists in the codebase in Markdown files. The system is made up of a hybrid of Python and OCaml modules in which the OCaml code functions as a

performant backend and the Python code functions as a high-level frontend that interfaces with the backend. One notable exception to this rule is the neural network code of the recognition model, which is written in Python and uses the PyTorch deep learning library (`https://pytorch.org/`).

The code contains experimental Python and Rust implementations of several of the modules described below, which can be toggled by command-line flags, but it is simplest to describe the default workflow where the OCaml backend is used for enumeration (waking), compression (abstraction), and dreaming. Advanced usage of DreamCoder using the different implementations is covered by the help options of the DreamCoder command-line interface.

In order to understand how the Python and OCaml parts of the codebase fit together, it is useful to outline the primary modules within the DreamCoder system, the different types of data passed between the modules, and the relationships between the data types. The next section will cover these three items. Then, the fundamental sequence of events and exchange of data at each of the major steps of DreamCoder's execution will be described below.

Data Types and Modules: The primary data types, modules, and their relationships are depicted in Figure S16. Data classes are shared between Python and OCaml, since both parts of the codebase require the ability to interpret and modify each of the primary data types. Data objects are serialized to and deserialized from JSON when the backend and frontend communicate. The data classes in the code approximately match the terms used in this paper.

`Program` objects represent the programs that DreamCoder generates to solve tasks. These programs are generated by the enumeration module `enumeration.ml` according to Algorithm 3 above. Programs are represented as λ-calculus expressions using deBuijn indices.The `Primitive` program type is used to specify programs that make up the initial library and are composed into larger, more complex programs during enumeration.

`Program` objects specify a type attribute that corresponds to the `Type` class. The `Type` can be either a ground type (e.g. `int, bool`), a type variable (e.g. α, β, γ) or a type built from type constructors (e.g. `list[int]`, `char→bool`, α`→list[`α`]`). Type objects are used to represent a type system capable of describing the expected input and output types of a program, as well as to ensure that a generated program is well-typed during enumeration. The type module `type.ml` contains routines for creating and transforming types, such as the `instantiate`, `unify`, and `apply` methods. These routines are imported by the enumeration module and called when building candidate programs during enumeration, as seen in Algorithm 2 in Appendix S4.4. The `Task` data that specifies input-output examples also references a `Type` class used to match tasks with suitable programs during enumeration.

`Grammar` objects contain the initial and learned library of code mentioned throughout this paper. A `Grammar` object is composed of a set of `Program` objects, namely `Primitive` and `Invented` programs, as well as a numerical weight used to calculate the probability of a given program being used when creating new programs (see Algorithm 2 above, where $\log\theta$ corresponds to the weight in the source code). The `Grammar` data serves as input to the enumeration module, because the enumeration routines create programs from the current library. This `Grammar` data is also input to the dream module `helmholtz.ml`, which generates training data for the recognition model (trained in `recognition.py`), and the `compression.ml` module updates the library as explained in Algorithm 4 of Appendix S4.5.1. In this way, the DreamCoder system uses `Grammar` objects to iteratively create and update a library in addition to generating programs from that changing library.

This web of relationships between the modules and data of the system is described from a sequential perspective of the program's workflow steps in the following section.

Program Workflow: DreamCoder's workflow is outlined in Figure S17. This figure is a sequential diagram

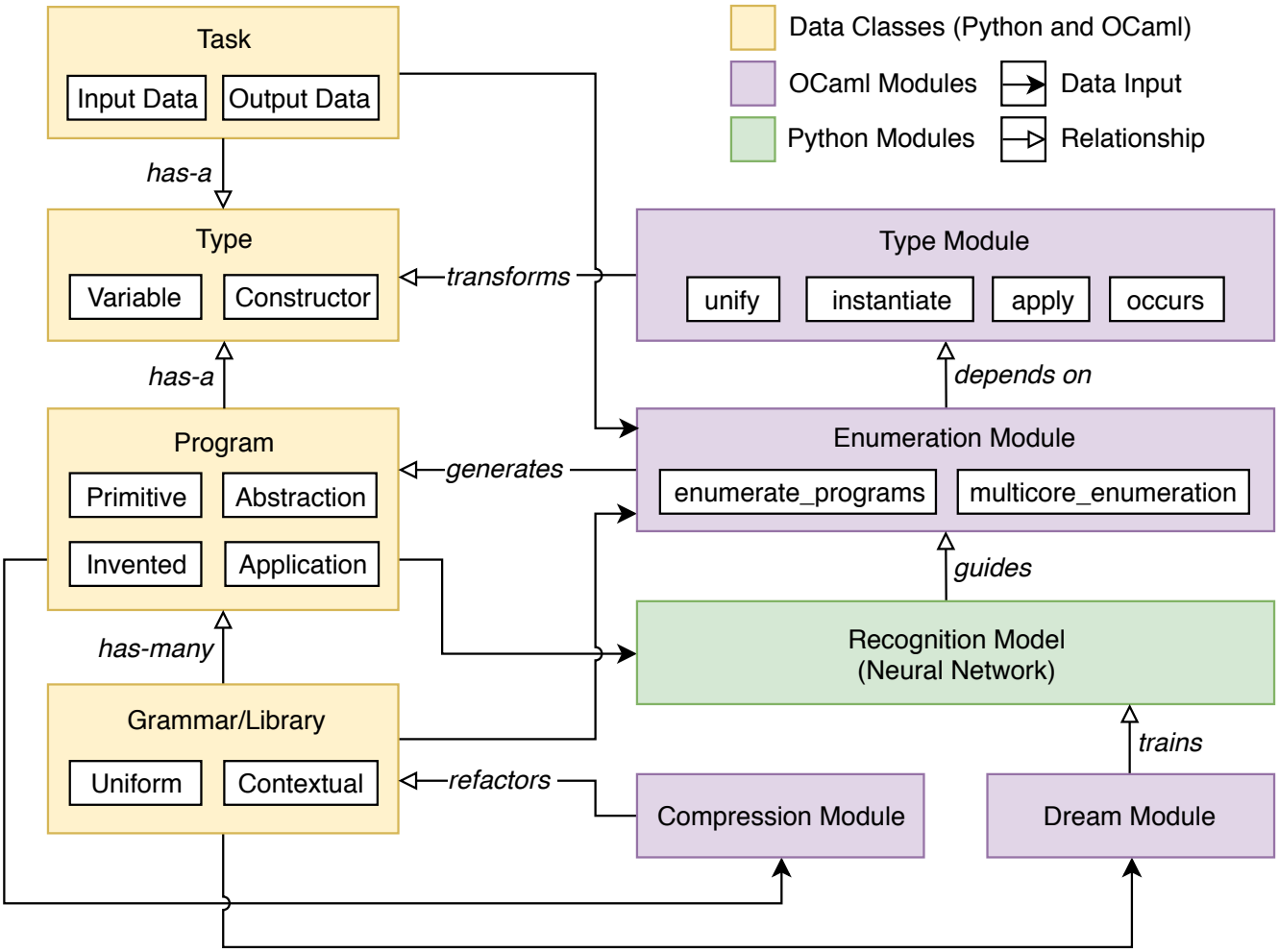


Figure S16: Relationships between primary data classes and source code modules.

of each of the major steps that occur during the system's life-cycle. These steps include: (1) dreaming, which occurs in parallel with (2) program enumeration from the current library, then sequentially (3) recognition model training, (4) program enumeration guided by the recognition model, (5) abstraction or compression, and (6) library visualization.

The entry point to the system is the `dreamcoder.py` module, which processes the command-line arguments specified by the user and iteratively invokes each of the following steps. The system optionally takes a checkpoint file as input to resume training from a previous state. Neural network state and metadata are serialized to or deserialized from binary files using the `pickle` module from Python's standard library. Checkpointing occurs at the end of each iteration of DreamCoder's execution. In addition to allowing the system to restart from a previous state, checkpoint files can be used to graph the system's performance during training and testing via the `bin/graphs.py` script. This script is used to create the graphs of the performance featured elsewhere in the paper, such as in Figure 6.

In Step 1, the `dreamcoder.py` module calls `dreaming.py` module, which creates multiple background workers that run OCaml processes in parallel. These processes generate dreams described by the Dream Sleep phase of Algorithm 1. The Python frontend communicates a request to the OCaml backend in JSON format after serializing the current library to JSON. The OCaml processes return responses asynchronously after enumerating sets of different programs that solve the requested tasks. The response data contains these sets of programs which are later used to train the recognition model in Step 3.

In Step 2, in parallel with Step 1, the system begins enumerating programs built from the current library, which is isomorphic to the dreaming process and, like dreaming, is launched by the `enumeration.py` module. The Python frontend spawns a new child process to run the `solver.ml` OCaml module which executes enumeration in parallel as detailed by Algorithm 3 above. The degree of parallelization is controlled by command-line flags, causing the algorithm to run on a parameterizable number of CPUs. Generated programs that successfully solve the input tasks are represented in a lambda calculus and returned in JSON format to the parent Python process.

In Step 3, the `recognition.py` module trains a neural network with PyTorch as a backend. The network is trained to predict the most likely program solving each task, both actual training tasks and dreamed tasks. During training, programs are sampled from the dreaming response data of Step 1. The specifics of this step are summarized in Appendix S4.6.

In Step 4, we enumerate programs, *for each task*, from the current library, guided by the recognition model's output for each task.

In Step 5, the `compression.py` frontend module spawns a child process to invoke the OCaml `compression.ml` module, which implements the compression procedure from Appendix S4.5. The OCaml module receives a request with the current programs produced by the previous wake cycles, and returns a new library in the response JSON.

In Step 6, the `primitiveGraph.py` module creates a snapshot of the current primitives in the library. This snapshot is saved as a PDF file, allowing for later inspection of the state of DreamCoder's library at any iteration in the program's life-cycle. Examples of the learned library routines shown in the figures of this paper, such as Figure 4, are drawn from these primitive snapshots.

After all steps have been completed, a final checkpoint file is created for the iteration. The above steps are iterated over in a loop limited by hyperparameters set by command-line flags (e.g. iteration count, enumeration timeout, limit on recognition model's training time and # gradient steps).

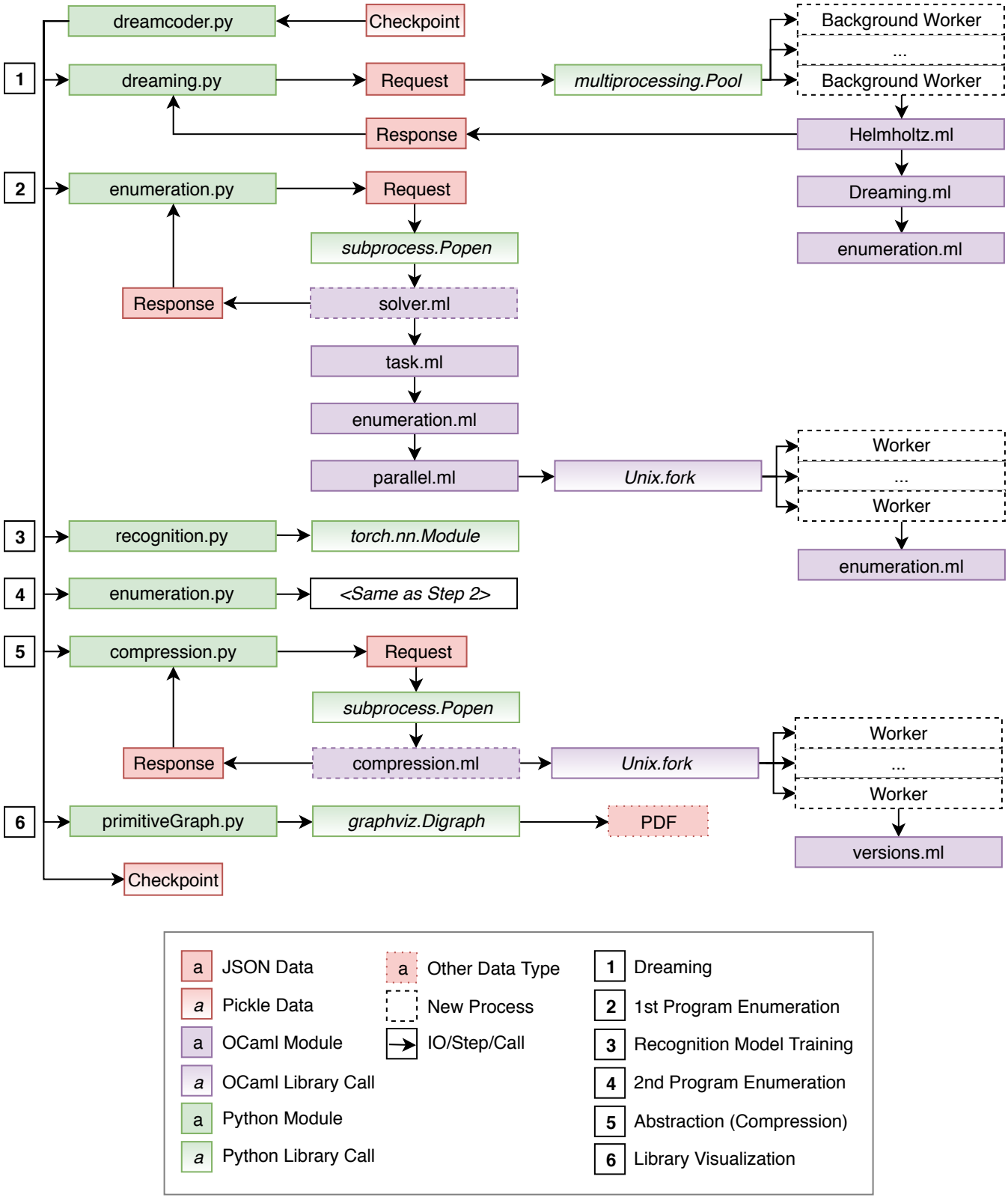


Figure S17: Program flowchart. Steps 1 & 2 occur in parallel.

References

1. 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.

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

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

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

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

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

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

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

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

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

11. Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S. M. Ali Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. *ICML*, 2018.

12. Noah D Goodman, Joshua B Tenenbaum, and T Gerstenberg. *Concepts in a probabilistic language of thought*. MIT Press, 2015.

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

14. Robert John Henderson. *Cumulative learning in the lambda calculus*. PhD thesis, Imperial College London, 2013.

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

16. Irvin Hwang, Andreas Stuhlmüller, and Noah D Goodman. Inducing probabilistic programs by bayesian program merging. *arXiv preprint arXiv:1110.5667*, 2011.

17. Yarden Katz, Noah D. Goodman, Kristian Kersting, Charles Kemp, and Joshua B. Tenenbaum. Modeling semantic cognition as logical dimensionality reduction. In *CogSci*, pages 71–76, 2008.

18. Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

19. J.D. Lafferty. *A Derivation of the Inside-outside Algorithm from the EM Algorithm*. Research report. IBM T.J. Watson Research Center, 2000.

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

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

22. 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.

23. Douglas Lenat and John Seely Brown. Why am and eurisko appear to work. volume 23, pages 236–240, 01 1983.

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

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

26. Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

27. 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.

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

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

30. Microsoft. F# guide: Units of measure, 2016.

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

32. Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. *ICML*, 2019.

33. Eray Özkural. Teraflop-scale incremental machine learning. *arXiv preprint arXiv:1103.1003*, 2011.

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

35. Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

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

37. Illia Polosukhin and Alexander Skidanov. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335*, 2018.

38. Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

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

40. Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Oleksandr Polozov. Program synthesis and semantic parsing with learned code idioms. *CoRR*, abs/1906.10816, 2019.

41. Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In *Advances in Neural Information Processing Systems*, 2017.

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

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

44. Gerald J Sussman. *A computational model of skill acquisition*. PhD thesis, MIT, 1973.

45. 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.

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

47. 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.

48. Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78. ACM, 1990.

49. Maksym Zavershynskyi, Alex Skidanov, and Illia Polosukhin. Naps: Natural program synthesis dataset. In *ICML*, 2018.