Algorithms for Learning to Induce Programs

by

Kevin Ellis

Submitted to the Department of Brain and Cognitive Sciences
in partial fulfillment of the requirements for the degree of

PhD in Cognitive Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2020

Author . . . . . .
Department of Brain and Cognitive Sciences
June 2020

Certified by . . . . . .
Joshua B. Tenenbaum
Professor
Thesis Supervisor

Certified by . . . . . .
Armando Solar-Lezama
Professor
Thesis Supervisor

Accepted by . . . . . .
Rebecca Saxe
John W. Jarve (1978) Professor of Brain and Cognitive Sciences
Associate Head, Department of Brain and Cognitive Sciences
Affiliate, McGovern Institute for Brain Science
Chairman, Department Committee on Graduate Theses

Algorithms for Learning to Induce Programs

by

Kevin Ellis

Submitted to the Department of Brain and Cognitive Sciences
on June 2020, in partial fulfillment of the
requirements for the degree of
PhD in Cognitive Science

Abstract

The future of machine learning should have a knowledge representation that supports, at a minimum, several features: Expressivity, interpretability, the potential for reuse by both humans and machines, while also enabling sample-efficient generalization. Here we argue that programs–i.e., source code–are a knowledge representation which can contribute to the project of capturing these elements of intelligence. This research direction however requires new program synthesis algorithms which can induce programs solving a range of AI tasks. This program induction challenge confronts two primary obstacles: the space of all programs is infinite, so we need a strong inductive bias or prior to steer us toward the correct programs; and even if we have that prior, effectively searching through the vast combinatorial space of all programs is generally intractable. We introduce algorithms that learn to induce programs, with the goal of addressing these two primary obstacles. Focusing on case studies in vision, computational linguistics, and learning-to-learn, we develop an algorithmic toolkit for learning inductive biases over programs as well as learning to search for programs, drawing on probabilistic, neural, and symbolic methods. Together this toolkit suggests ways in which program induction can contribute to AI, and how we can use learning to improve program synthesis technologies.

Thesis Supervisor: Joshua B. Tenenbaum
Title: Professor

Thesis Supervisor: Armando Solar-Lezama
Title: Professor

Acknowledgments

It's unfortunate that only a single name is attached to PhD theses, because the work in this thesis is a massive team effort.

Both my advisors, Armando Solar-Lezama and Josh Tenenbaum, helped me both perform this work and grow as a researcher. With Josh I had the privilege of getting to start out as an undergraduate in his class, and then transitioning into working in the lab as a graduate student. He helped me in many ways, from applying to grad school to applying for jobs, but especially in learning to think more scientifically, write more convincingly, design more compelling illustrations, give better talks, collaborate more effectively, and scope better problems. The intellectual environment of the lab is surprising both in its breadth and its depth, and I couldn't think of a better place to work between cognitive science and artificial intelligence. I'm incredibly grateful to have also worked under Armando, who helped teach me to be a computer scientist. Armando has the unparalleled ability to quickly see to the core of a problem—even having never seen it before—and both sketch a solution and intuit the more general case. He helped teach me how to present and explain my work more coherently; design and conduct the correct experiments, even if they are nonobvious; identify and move toward the most practical next steps; get a job after graduation; and seems to have a large collection of "life/research algorithms/heuristics" that I have frequently solicited and used. His lab is similarly vibrant and I would not have made the same kind of contact with the programming languages community were it not for him.

One of the best parts of grad school was the opportunity to do extended collaborations with many brilliant and interesting people. Interning under Sumit Gulwani grew my appreciation for practical problems forced me to really learn how to program. Later collaborating with Lucas Morales and Luc Cary made me a better engineer—but I'm still nowhere near as good as they are. Working with Mathias Sablé-Meyer was really fun: he

built one of the best examples of DreamCoder in action, and we became friends in the process. He's kept me up to date with what he has been doing since and also gave me the template for my webpage. Max Nye has given a lot of life advice, including during the job search, and is spearheading what I see as a key part of the future of program induction. Evan (Yewen) Pu is one of the most interesting people I've met, both personally and scientifically. I'm inspired by the creativity of the problems that he works on and the unique ways in which he pursues solutions and answers. I benefited from frequent discussions and hangouts with Evan and Max, as well as our collaborations. They also organize excellent dinner parties. Cathy Wong is another talented scientist who I benefited from getting to work with. Some of the best writing in Chapter 5, as well as much of the work, comes from her. Luke Hewitt was an important "fellow program inductor" whose memoised wake-sleep algorithm distills key ideas from Exploration-Compression in complementary and in many cases better ways than was done in DreamCoder. Our conversations contributed to me ideas about Bayesian inference and its relation to neural networks which can be seen in this thesis. Felix Sosa's ideas helped spur the development of the REPL project, and I'm grateful that Felix, Max, and I sat down at a table during CogSci and hashed out the beginnings of that project. Daniel Ritchie is a very helpful collaborator who is unusually generous with his time, and he helped me learn the modern deep learning toolkit. Kliment Serafimov was a undergrad who worked with me on DreamCoder applications, and I've found his enthusiasm, intelligence, and curiosity inspiring–he motivated one of the sections in the concluding chapter. Sam Tenka and I worked together briefly at the end of grad school and, although our work on ARC did not make it into the thesis, he contributed a key observation about DreamCoder's workings. Eyal Dechter, a graduate student at the time, worked with me while I was an undergrad; without him I would not have gone down this particular research direction. Eyal also helped me apply to grad school, and he showed me by example what a great talk looks like. He helped make this thesis possible. Collaborating with Tim

O'Donnell is the sole reason why I was able to do a project in computational linguistics. Every time I talk with him I learn two new words and three new facts about either language or computation. Collaborating with Lucas Tian and Marta Kryven taught me about how cognitive scientists think and work. Toward the end of grad school, working with them helped tether me to the mind and brain. They also throw really good dinner parties, together with Ronald, Essie, Tuan-Anh, and Julio.

Last, but very importantly, I'd like to acknowledge my family, including my parents and sister, who provided support and encouragement during this process, and who were patient while I spent a decade at MIT. My grandparents, John Morse and Mary-Jo Morse, were fierce supporters of my education, which helped lay the foundation for the work here.

Contents

Chapter 1

Introduction

A long-standing the goal of computing has been to build machines that can program themselves: which autonomously construct useful software on the basis of experience, data, percepts, or formal specification. The practical utility of such technology is clear, because *programming is hard for humans.* Any progress toward automatic programming has the potential to spill over to advancing software engineering and computer science more broadly. Indeed, one can see the development of compilers as a basic first step toward automatic programming, and program synthesis as the logical evolution along this path.

But a slightly less obvious (though long appreciated) motivation for program synthesis is its application to artificial intelligence (AI). Viewed through an AI lens, source code is a kind of *knowledge representation* and program synthesis is a *learning algorithm.* As a representation of knowledge, programs bring several 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. Finally, programs are

universal, or Turing-complete: in principle, programs can represent solutions to the full spectrum of computational problems solvable by intelligence. Indeed, program synthesis serves as a foundation for most theoretical proposals for "universal AI" [126, 116, 63]. One should be skeptical of claims of universality: there is no "free lunch" [145], no truly generically applicable, bias-free learning algorithm. But program synthesis may prove to be the cheapest lunch on the menu for certain classes of AI problems, particularly when sample efficiency and interpretability are primary concerns.

The project of advancing automatic programming is far too large for a single thesis, and has received attention from multiple generations of computer scientists. Thus we must carefully delimit our scope. This thesis has two primary aims. The first aim is to advance the foundations of program synthesis, primarily by drawing on machine learning methods. This aim is accomplished by assembling a new toolkit of algorithms, framings, and reusable software implementations. The second aim is to expand the scope of program synthesis by illustrating how this toolkit can be deployed for artificial intelligence tasks, focusing on case studies in inverse graphics and theory induction. We view this research as "sandwiching" program synthesis between machine learning and artificial intelligence. Program synthesis restricted to inference and learning tasks is commonly called **program induction**[1]; using this terminology, the goal of this thesis is to use learning to advance program induction for AI tasks.

[1] A series of recent papers [33, 32] in the literature adopts a different definition of program induction, and contrasts it with "program synthesis." The definition of program induction used here–namely, as a particular subset of program synthesis–is older but still in common use [82, 30]. Unfortunately the literature now contains two different definitions of "program induction." In practice, I have fortunately not found my use of this phrase to cause confusion among researchers in artificial intelligence, machine learning, and programming languages.

1.1 What is a program?

A thesis which claims to introduce algorithms for learning programs must carefully carve out its actual scope: in a precise formal sense, every object within a computer corresponds to the output of some program, and so the project of learning programs must be, in the general case, at least as hard as learning anything else. But some objects are more program-*like* than others: while logistic regressors, 3D meshes, and neural networks are technically special cases of programs, we would not typically think of them that way. Other objects are more representative examples of programs: expressions in datalog, generative grammars, LaTeX source code, causal models, or equations in physics. It is difficult to formally pin down what makes these objects more program-*like* than the others, but from the perspective of artificial intelligence, they share several desirable properties, listed below. From the perspective of this thesis, a *program* is any object which satisfies these properties:

- A separation of syntax and semantics: Programs have *syntax*, the language in which they are expressed, and a *semantics*, the domain in which they operate. The semantics is a function of the syntax.

- Compositionality: The way in which the semantics is constructed from the syntax is by recursively computing the semantics of smaller pieces and then joining them in a manner which depends only on the way which they are joined, and not on the syntax of their constituents.

- Plausible interpretability: Not all syntactic expressions need be interpretable, but the computational substrate on which the program is built should at least offer a plausible route to interpretability. Not all programming languages satisfy this property: Python offers a plausible route to interpretability, but Turing machines do not.

We have not defined these desiderata formally: if we could define them formally,[2] then we could define formally what it meant for something to be program-*like*. A further property is *expressivity*, or the set of functions and relations which a representation can encode. For example, propositional logic is strictly less expressive than first-order logic; Turing machines and λ-calculus have the same expressivity. We do not see extreme expressivity as a requirement for program-*like*-ness, and will consider relatively inexpressive classes of programs such as those built from arithmetic operators, all the way up to maximally expressive classes of programs, such as arbitrary recursive programs built using the Y-combinator. The full spectrum of expressivity is practically useful for program induction systems: for instance, if we want to discover the equations governing a novel fluid, restricting ourselves to partial differential equations is both adequate and likely to be more effective than considering the space of models simulatable by Turing machines. On the other hand, if we have only weak prior knowledge about a domain, a highly expressive hypothesis class may be necessary.

These diagnostic properties of program-like representations are not exclusive to symbolic approaches. For example, neural module networks [7] can satisfy these properties. Nonetheless we will restrict our focus in this thesis to symbolic programs, a point we will return to in Section 2.4.

1.2 Two primary obstacles to program induction

Program induction confronts two intertwined challenges, which we seek to address in this work:

- **Combinatorial search.** Synthesizing programs from data such as images, input/out-

[2] Syntax/semantics and compositionality can be defined formally; plausible interpretability can only be defined formally relative to a human oracle.

puts, etc. requires searching for a program consistent with that data. But the space of all programs is infinite and sharply discontinuous: tiny changes to the source code of a program can dramatically alter its semantics. These factors render naive search strategies unapplicable in all but the simplest of cases. How can we efficiently explore the vast space of all possible programs, and home in on programs that fit the data also satisfying other desiderata, such as parsimony?

- **Inductive bias.** When performing program induction—i.e., inferring programs from data rather than formal logical specifications—one intends for the program to *generalize*. For example, if the program maps inputs to outputs, then it should correctly transform held-out inputs; if the program implements a causal theory, then it should make correct predictions for new data. And if programs are viewed as a kind of knowledge representation, then the inferred program should actually represent knowledge that humans can understand. These concerns are not unique to programs, and essentially every learning system requires a prior or inductive bias to guide it toward good hypotheses. But this concern is especially relevant for highly expressive spaces of programs: for instance, if one searches over a Turing complete language, then *every* computable hypothesis is expressible. Furthermore, not all inductive biases lead to parsimonious, human-understandable programs, a central issue that we will dwell more on in Chapter 5.

In more technical terms, the challenge of combinatorial search corresponds to the **computational complexity** of program induction, while the challenge of the inductive bias corresponds to **sample complexity**. This thesis addresses both these obstacles through different kinds of learning algorithms: intuitively, we learn to search, and also learn an inductive bias. We build agents that learn these ingredients via experience solving a corpus of related program induction problems. Implicit in this approach is a concession that there

exists no truly generic search strategy over the space of Turing computable functions, nor does there exist a "universal prior" that is optimal for all domains (cf. [63, 116, 126]). But the flipside is a cautious optimism that there exist domain-general strategies for learning these domain-specific ingredients.

This optimistic goal is not yet fully realized—this thesis does *not* yet provide a general architecture for learning to induce programs—but the general principles developed here move us closer to such a system. This thesis is organized around a sequence of case studies illustrating computational principles that build progressively toward more general learning architectures (Figure 1-5) applied to a range of domains, some of which are shown in Figure 1-1. The next section previews techniques contributed by this thesis which move us toward this vision of learning to induce programs.

1.3 Contributions

This thesis makes two kinds of contributions: (1) Expanding the scope of program synthesis within artificial intelligence, by developing program synthesis systems which tackle a range of AI problems; and (2) developing the learning algorithms which make such expansion possible. We preview the application of our systems to AI problems in Section 1.3.1, and preview the principles behind these learning algorithms in Section 1.3.2.

1.3.1 Expanding the role of program synthesis

Program synthesis has a long and rich history [93, 56, 6], and has received increased interest following practical demonstrations of synthesizers for classic programming languages tasks such as synthesizing low-level arithmetic and bit-vector routines [125, 129, 2], assembly-code super-optimization [114], text manipulation [105, 55], and others. A core contribution

of this thesis is to expand the role of program synthesis within artificial intelligence domains, and to show that such expansion is practical. Although program induction is one of the oldest theories of general learning within AI, it is fair to say that it has largely remained at the margins of the field of AI [126, 95]. While the work here has not single-handedly reoriented the field, we hope that it points the way toward a future where program induction could be seen as a standard part of the AI toolkit.

We will cover new program synthesizers which recover visual graphics routines from voxel arrays and noisy images; program synthesizers which solve planning problems; systems which recover causal models of phenomena in natural language; systems which discover interpretable physics equations; and others (Figure 1-1,1-2). We believe these advances are practical. Our visual program induction systems (Chapter 3) can synthesize LaTeX code generating figures like those found in machine learning papers given natural hand-drawn input, as well as 3D CAD programs given voxel input. Our program induction system for natural language can solve the majority of the morphology and phonology problems across several linguistics textbooks (Chapter 4). The same system which learns to solve planning problems also learns to infer physics equations from simulated data, and draws on new general learning algorithms introduced in this thesis (Chapter 5).

1.3.2 Toward overcoming the two obstacles

This thesis contributes learning algorithms that draw on the following idea: Rather than solve problems in isolation, we can use learning to acquire the domain knowledge needed both for an inductive bias, and also for a search strategy, thereby overcoming the two primary obstacles laid out in Section 1.2. This is essentially a kind of meta-learning or learning-to-learn. The success of these learning-to-learn algorithms depends critically on a corpus of training problems: By solving the training program synthesis tasks, these

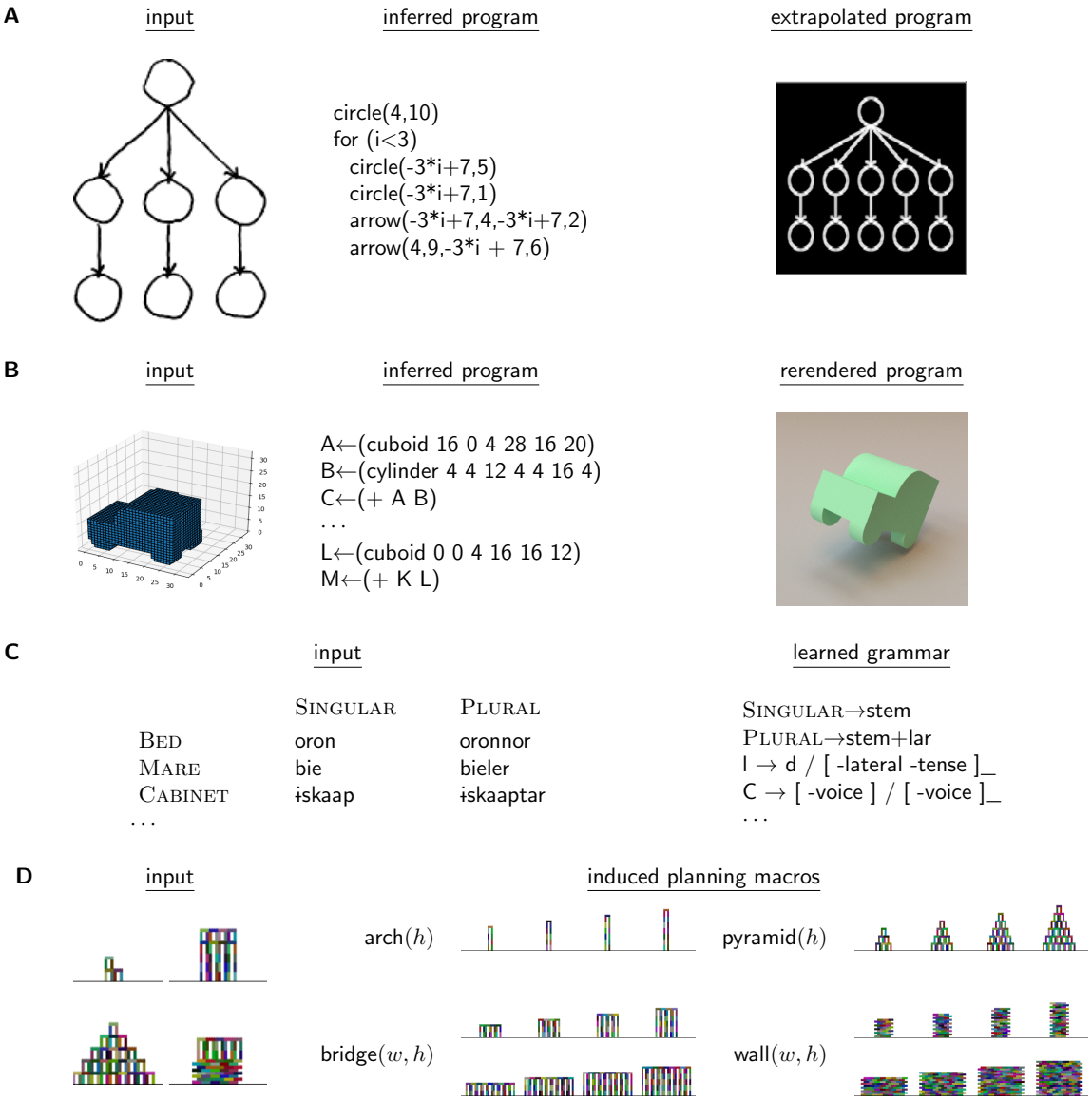


Figure 1-1: **(A-B)** Chapter 3, program induction from visual input. In both cases the input is a rasterization (**A**, 2D pixels; **B**, 3D voxels) and the system learns to infer a graphics routine, which may be extrapolated to produce an analogous but larger image **(A)** or rerendered from novel viewpoints **(B)**. Both these systems illustrate approaches for learning to *search* for programs. **(C)** Chapter 4, program induction for human-understandable model discovery, focusing on natural language grammar (morphophonology). The system illustrates principles of learning an *inductive bias* over programs by shared statistical strength across different languages. **(D)** Chapter 5, progress toward a general algorithm for learning to induced programs, illustrated here within a blocks-world planning domain. This algorithm combines learning inductive biases with learning search strategies. The learned inductive bias is built from a library of reusable abstractions, which here correspond to macros or "options" useful for planning.

algorithms learn inductive biases and search strategies which generalize to new induction problems.

In engineering terms, we will situate this learning-to-learn framework in a hierarchical Bayesian formalism. We will model the inductive bias as a probabilistic prior over program source code, and treat a corpus of training problems as the observed data, which will be explained by synthesizing a latent program solving each training problem. By learning the mapping from observed data to program, we will learn to search. By doing inference that the level of the prior over source code, we will estimate an inductive bias over problem solutions.

Algorithms for learning an inductive bias

The programming language given to a program synthesizer strongly influences the kinds of inductive biases it will exhibit. For example, given a list processing language such as Lisp, it will be straightforward to express recursive patterns on sequences, but awkward to express regular expressions; Given a markup language such as LaTeX it will be trivial to express simple diagrams built out of lines and circles, but effectively impossible to write a sorting algorithm. The fact that some programming languages make certain tasks easy to express–and hence easier to learn–while leaving others effectively inexpressible is sometimes called a **language bias**. Practical, real-world program synthesizers exclusively work within restricted **Domain Specific Languages** (DSLs) [56, 105]: non-Turing complete specialized languages, equipped with functions, data types, control flow patterns, and macros that are tuned to a specific class of problems. Could we learn something like a DSL — the most foundational kind of inductive bias within program induction?

While the goal of learning an entire DSL is far-off, this thesis contributes algorithms for augmenting existing DSLs by adding new reusable functions to the language. This

approach is roughly analogous to human programmers who enrich their programming language by writing libraries of reusable functions. A library implements a kind of learned language bias, but where foundational elements of language are held fixed, such as the primitive datatypes and control flow schemas. Imagine how importing a library changes the kinds of programs that are easy to express, and how it biases us as human engineers toward certain kinds of code: the long-term goal of the library-learning algorithms contributed here is to allow program synthesizers to grow their own libraries for new domains of problems, thereby biasing them toward effective domain-specific solutions.

At a high level, the algorithms introduced in Sections 4 & 5.1.2 learn libraries by alternating between using the current library to solve a corpus of program synthesis problems, and then updating that library by "compressing out" reused code schemas common across multiple problem solutions. Figure 1-2 illustrates the end result of such an approach when synthesizing physics equations from simulated data: each learned library routine builds on top of those learned earlier, growing out a network of reusable functions. We will see that this strategy falls out naturally from a Bayesian formulation, which prefers compressive joint representations of programs and the libraries in which they are expressed. These library learning algorithms also estimate a probability associated with each library subroutine, and even if we hold the symbolic structure of the library fixed, simply estimating these probabilities is another, simpler way of learning an inductive bias. In Section 3.1.3 we will explore how learning these probabilities can be used to more reliably induce programs from perceptual (visual) data, in essence modeling the interaction between bottom-up perceptual biases and the top-down inductive bias over the space of programs.

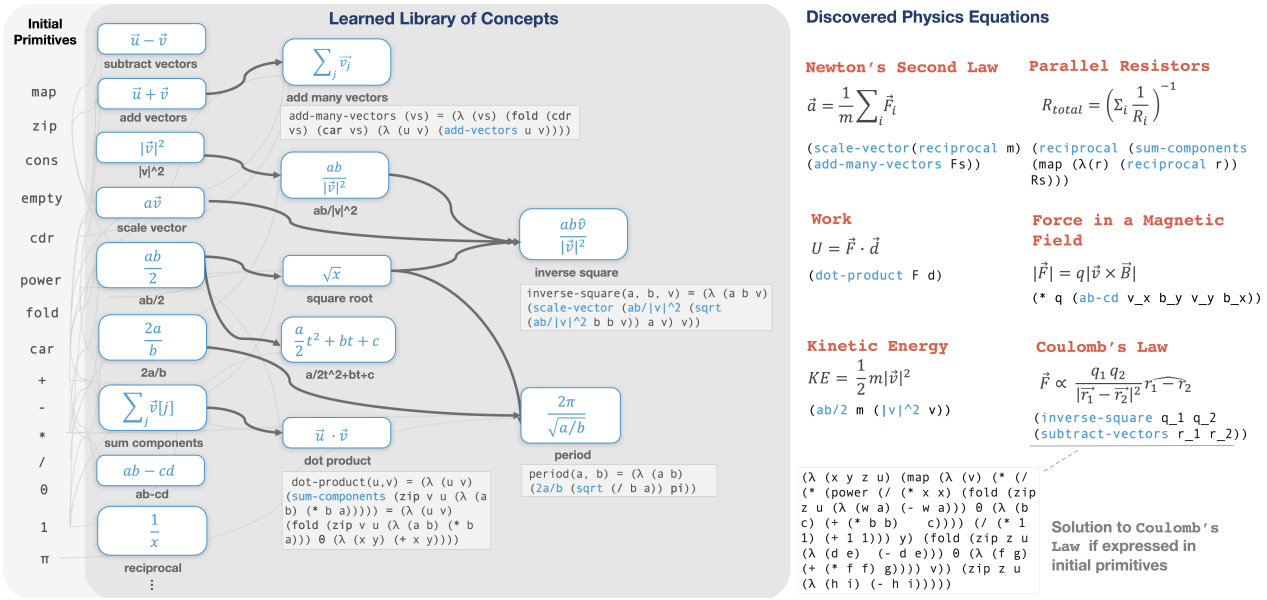


Figure 1-2: Learning an inductive bias by inducing a library of reusable functions. The program induction algorithm, described in Chapter 5, begins with an initial set of primitives (leftmost) in its library, and solves learning tasks such as equations shown on the right by writing a program (below each equation) which fits simulated data drawn from that physics equation. Learned library routines can call each other, notated via arrows. In the first layer of the learned library, the system rediscovers basic vector operations by building them out of `map` and `fold`, which are given to the system in its initial library. Moving deeper into the learned library, we see routines which capture common patterns across physics equations, such as inverse square laws. Each program is typically interpretable when expressed in terms of the learned library, but may be much longer and less interpretable when reexpressed in terms of the initial primitives, which we illustrate for Coulombs law.

Algorithms for learning to search

Even given an inductive bias that is finely tuned to a particular class of problems, program induction may remain intractable. Consider a typical 3D CAD program, or a program in Perl which edits lines in a file: In each case the language is specialized to the domain (graphics and text, respectively) but in general these programs may be very large, and schemes such as enumeration, random search, or even constraint solving scale worst-case exponentially with program size. Just as we will address inductive biases using learning methods, we will now view *search* as a learning problem: given a corpus of program induction problems, how can we learn to search in a manner which will generalize to new related problems?

This thesis contributes three algorithms for learning to search for programs. Sections 3.1.2 & 5.1.3 introduce methods then draw primarily on two ideas. The first idea is that even if writing an entire program is difficult, it may be possible to easily recognize broad features of the desired program just from the observed data. For example, if we are inducing a program that draws an image, and we observe that the image contains circles, then we should bias our search toward programs that call out to a circle-drawing routine, even if we cannot immediately work out the exact positions, sizes, and geometric layouts of the circles. The second idea is that we can still leverage the powerful symbolic program synthesis methods developed by the programming manages community over the past two decades, such as constraint solving combined with sketching [124]. Putting these two ideas together, we arrive at a family of approaches for learning to search where a learned model inputs the data for a synthesis problem and outputs hyperparameters for a downstream, non-learned program synthesizer. Concretely, Section 3.1.2 trains a learned model to bias the sketch program synthesizer toward spending more time in regions of the search space likely to contain satisfying programs, while Section 5.1.3 trains a neural network to output

probabilities guiding an enumerative type-directed solver [44].

Approaches such as these suffer from several drawbacks: they still expect a solver to synthesize the entire program at once, and learning plays only a relatively small role in the search process. Section 3.2 (Figure 1-3) introduces an approach that we see as a more scalable way to integrate learning with symbolic search, motivated by the observation that human programmers write long, complicated programs piece-by-piece. This section introduces an approach where a neural network is trained to output small pieces of code at a time, conditioned on the execution of code-so-far as well as the desired program behavior. This has the advantage of more deeply integrating learning into the search process, because the learned model is repeatedly queried online during inference. It also more closely mirrors how we as human engineers build long programs: often interactively, within an interpreter, executing code as we go along.

How should we get the data to train these systems to search for programs? Assuming we have a probabilistic prior over program source code, then we can generate an endless supply of synthetic programs to train on. Note that we cannot pull the same trick with inductive biases: if we tried to estimate an inductive bias from samples from a prior, we would only recover that exact same prior.

Synergy between search strategies and inductive biases

At its most technical, this thesis aims to identify and address two particular algorithmic problems connected to program induction. Surprisingly, solutions to each of these obstacles–inductive bias and search–can interact synergistically: a system which can more effectively solve search problems can also more effectively learns inductive biases, and vice versa. This synergy is a main theme of Chapter 5.

Briefly, this synergistic interaction between learning inductive biases and learning

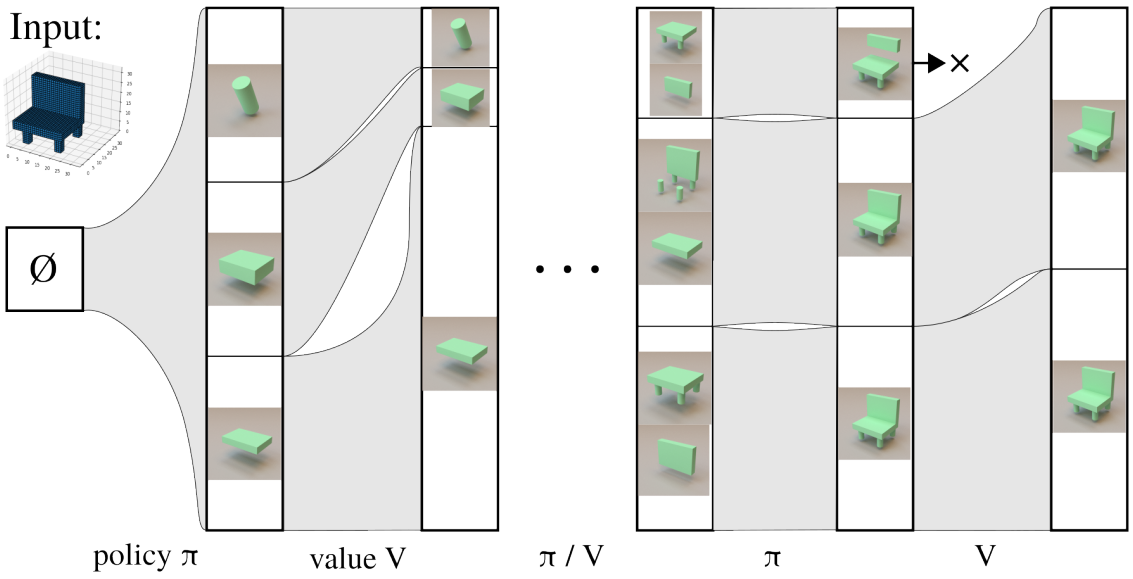


Figure 1-3: An illustration of a neurally-guided program search algorithm introduced in Chapter 3.2 applied to 3-D graphics programs. Given an input voxel field, the algorithm begins with the empty program (notated via the empty set, $\varnothing$) and explores a search tree of possible programs by alternatingly proposing lines of code via a neural policy π and then reweighting branches of the search tree deemed more promising by a neural value function V. This is roughly modeled after systems such as AlphaGo [119] which interleave policy and value networks with an explicit symbolic tree search. We visualize intermediate states in the search tree by rendering the corresponding intermediate program. These renderers are shown to the neural networks, effectively executing code as it is written in a manner similar to interacting with an interpreter or REPL.

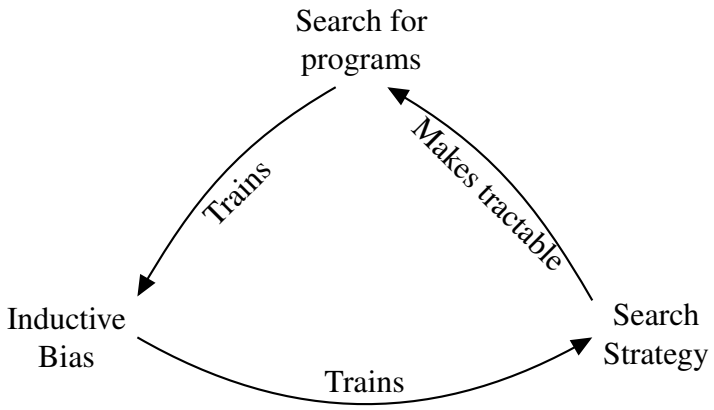


Figure 1-4: Synergy between learning an inductive bias and learning a search strategy. As we solve more problems (top) we get more data from which to estimate a prior or inductive bias over programs. If we train our search strategy on samples from the prior, then improving the prior also improves the search strategy. But a better search strategy solves more problems, which then feeds back into the inductive bias, creating a positive feedback loop.

search strategies works as follows (Figure 1-4): We assume that we have a training corpus of program synthesis problems to solve, but without any ground truth program solutions. We will learn our inductive bias by estimating a probability distribution over programs solving problems in the corpus, and that we will learn our search strategy both on the basis of solutions to training problems, but also on the unlimited synthetic data generated by sampling from the learned prior. As our inductive bias improves, this synthetic data will come to more closely resemble actual problems to solve, and so will be more useful for training the search strategy. But as our search strategy improves, we will solve more problems, giving more data from which to estimate our prior over programs. Therefore we have a positive feedback loop: when the inductive bias improves, the search strategy improves, and vice versa, causing the system to improve in a bootstrapping fashion.

Although program induction meets other challenges apart from inductive bias and search, our choice of these two technical obstacles isn't arbitrary: we chose these not

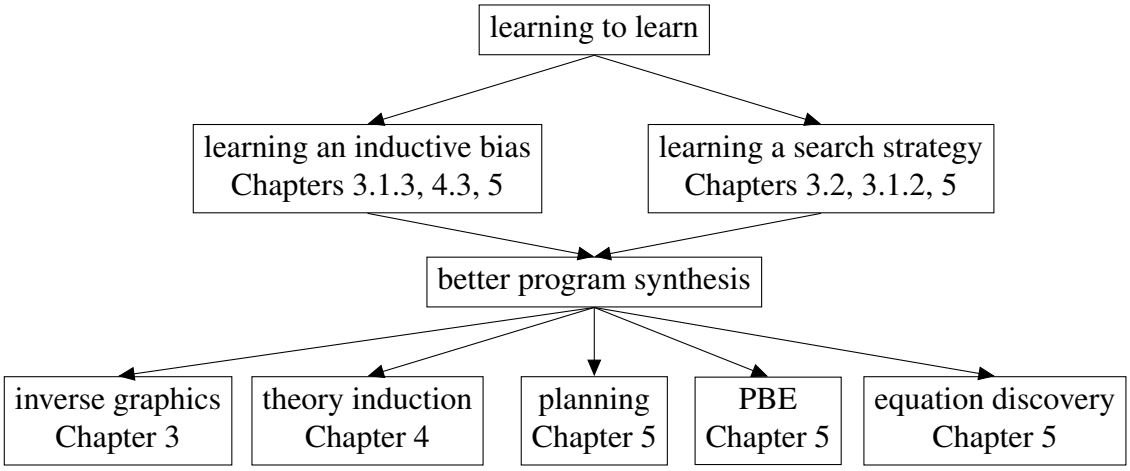


Figure 1-5: Thesis overview. This work contributes new methods for learning to learn programs which work by jointly learning inductive biases over program spaces, and learning to effectively deploy that inductive bias by searching over that space of programs. These methods are applicable to a range of learning and inference tasks. "PBE" is short for programming-by-examples, of which [55] is the most famous example.

only because they are central, but because solutions to one aids the other. We suspect that this bootstrapping, learning-to-learn approach is one of our best bets for getting program induction to work at scale, and we hope that, in reading this thesis, you will come to share this perspective.

Chapter 2

The Technical Landscape

Here we survey the prior art viewed through the lens of what we see as the two primary obstacles to program induction: *inductive bias* and *combinatorial search*, with an emphasis on learning-based methods. At the same time we will introduce a technical vocabulary and a Bayesian framework that will be threaded throughout the thesis, and offer previews of how this vocabulary and framework provide a foundation for the work described in later sections.

Much of this landscape is, in a sense, neurosymbolic: it uses neural network learning methods to assist the synthesis of symbolic programs. In the Section 2.4 we will discuss a different sense of neurosymbolic program induction, where the programs themselves are neither fully symbolic nor fully neural.

2.1 A probabilistic framework for learning to induce programs

Throughout this thesis we adopt a Bayesian view of program induction. This probabilistic approach gives a unified framework for noise, ambiguity, mixed discrete/continuous parameters, and learning-to-learn (through hierarchical Bayes). It also allows us to write down a single objective function jointly capturing these different kinds of learning and inference, and from this objective function we can then derive different algorithms.

Figure 2-2 diagrams this general approach as a graphical model. An inductive bias specifies a prior probability distribution over programs, such as a probabilistic context free grammar over syntax trees. Each latent program generates observed data, such as images, plans, or input/output examples. This setup is, in general, multitask: we might have many programs to induce. This graphical model has the joint distribution:

$$\mathrm{P}[\text{inductive bias}] \prod_{n=1}^{N} \mathrm{P}\left[\text{program}_n | \text{inductive bias}\right] \mathrm{P}\left[\text{data}_n | \text{program}_n\right] \tag{2.1}$$

Variations of this Bayesian framework underlie a number of program induction systems, such as Lake et al.'s model of hand-drawn characters [73], Liang et al.'s hierarchical Bayesian approach [81], Dechter et al.'s EC algorithm [30], dependent learning within inductive logic programming [83], and others [39, 36, 66, 59, 135, 40, 140].

Focusing on the work described here, this generative model will be used as follows:

- Graphics programs (Chapter 3): Here each observed datum is a raster image, and each program is a graphics routine which deterministically generates observed pixels. These pixels might be noisy, a fact which is folded into the program→observation likelihood model. Especially because of this noise there may be ambiguity as to the correct program, and by estimating a prior over programs we can learn to infer more

plausible program-parses of each image.

- Natural language grammar (Chapter 4): Here each observed datum is a corpus of words from a natural language, and the goal of the system is to infer a program implementing a grammar for each language. Each program nondeterministically generates possible words of the language, and we seek programs which have high likelihood of generating the observed corpora. The inductive bias corresponds to a "universal grammar," i.e. a distribution over grammars which models cross-language typological tendencies.

Chapter 5 will consider more general algorithms for inference in this graphical model which are not tied to any particular domain.

We now break apart this equation to illustrate different elements of learning and inference that can be captured in this setup.

Modeling noise

Different kinds of noise and cost models may be folded into the likelihood term P[data|program]. For example, when generating images via graphics programs, we could use a L_2 norm to penalize deviations from the observed image (hence a Gaussian likelihood function):

$$\mathrm{P}\left[\text{data}|\text{program}\right] \propto \prod_{\vec{r}} \exp\left(-\frac{(\llbracket\text{program}\rrbracket(\vec{r}) - \text{data}(\vec{r}))^2}{2\sigma^2}\right) \tag{2.2}$$

where data is an image (a function from position to light intensity), $\llbracket\cdot\rrbracket$ is an interpreter that maps programs to images, $\vec{r}$ ranges over positions in the observed image, and σ is the standard deviation of the likelihood noise model.

When inferring programs from input/output examples, we typically assume no noise, and the likelihood function is just an indicator which is one iff the program is consistent

with the data (input/output examples):

$$\mathrm{P}\left[\text{data}|\text{program}\right] = \prod_{(x,y)\in\text{data}} \mathbb{1}\left[[\![\text{program}]\!](x) = y\right] \tag{2.3}$$

where (x, y) ranges over input/output examples in the observed data.[1]

Modeling nondeterminism

When inducing probabilistic programs, we can fold a marginalization over random choices into the likelihood function; for instance, if we add a "noise" random variable, ϵ_n for data_n, then we can define

$$\mathrm{P}\left[\text{data}_n|\text{program}_n\right] = \int \mathrm{d}\epsilon_n \; \mathrm{P}\left[\text{data}_n|\text{program}_n, \epsilon_n\right] \mathrm{P}\left[\epsilon_n|\text{program}_n\right]$$

Probabilistic program induction—where one marginalizes over random choices made by the synthesized program–can be seen in works such as [68, 139, 113]. A key obstacle is that, in order to even score how well a candidate program matches the data, we have to perform an integral that is, in general, intractable: i.e., it is no longer easy to check the quality of a candidate solution. This is analogous to popping up one level higher in the polynomial hierarchy [122]. For certain classes of programs, such as Gaussian process kernels, this integral may be performed in closed form, a property exploited by [113]. In general, one may have to rely upon enumeration or approximate Monte Carlo methods, such as the inner-loop Gibbs sampling performed in [68, 139]. In Section 5.2 we consider probabilistic programs that generate text, but where the marginalization over random choices may be performed efficiently via dynamic programming.

[1] Eagle eyed readers may notice that there is a bug in this likelihood function: really, it should *condition* on the inputs, rather than calculate the probability of the inputs, in order to ensure that summing over the space of data's for a given program comes out to 1. For uniformity of notation, we can ignore this notational bug.

Generalizing to continuous parameters

When inducing programs with continuous parameters, such as for symbolic regression [37] or graphics routines [39], one can treat these continuous parameters as random draws from a probabilistic (nondeterministic) program exactly as above, and employ the same marginalization scheme within the likelihood calculation. Here however, it may be more appropriate to approximate the marginal likelihood with a point estimate obtained via continuous optimization, as done by Lake et al. [73].

Resolving ambiguity

In general, there may be many programs consistent with the data. To represent this ambiguity we construct approximations to the posterior, $\mathrm{P}\left[\text{program}_n | \text{data}_n, \text{inductive bias}\right]$. Example approximations include population of samples [40], a beam of the top-k most likely programs [37], or version spaces [36]. One can integrate over this posterior when making predictions: for instance, if these programs map inputs X to outputs Y, then

$$\mathrm{P}\left[Y | X, \text{data}_n\right] = \sum_{\text{program}_n} \mathrm{P}\left[\text{program}_n | \text{data}_n, \text{inductive bias}\right] \mathbb{1}\left[\text{program}_n(X) = Y\right]$$

or, if the program implements a probabilistic generative model over members of a set X, then we can compute the posterior predictive distribution (where $x \in X$):

$$\mathrm{P}\left[x | \text{data}_n\right] = \sum_{\text{program}_n} \mathrm{P}\left[\text{program}_n | \text{data}_n, \text{inductive bias}\right] \mathrm{P}\left[x | \text{program}_n\right]$$

2.2 Learning an inductive bias

A good inductive bias, i.e. prior, over the space of programs is crucial when the data leave the synthesis problem ill posed. However, an inductive bias has other uses as well: by

learning the inductive bias, and then deploying it on new problems, we can implement a kind of transfer learning. By sampling from the prior, we can imagine new instances of similar concepts. When learning from a small number of examples, we can reduce the amount of data needed for generalization, which is critical when these examples come from end-users. An inductive bias also can be used to prioritize regions of the search space which are more likely to contain programs fitting the data, and so a strong prior can aid search as well.

We take as our goal to learn an inductive bias, and to do so absent strong supervision– i.e., without supervising on programs. This thesis will focus on generative approaches to learning inductive biases, but the literature also contains discriminative methods, which we will briefly outline.

Generative learning. Generative approaches to learning priors over programs generally calculate:

$$\underset{\text{inductive bias}}{\arg\max}\ \mathrm{P}\left[\text{inductive bias}\right] \prod_{n=1}^{N} \sum_{\text{program}_n} \mathrm{P}\left[\text{data}_n|\text{program}_n\right] \mathrm{P}\left[\text{program}_n|\text{inductive bias}\right] \tag{2.4}$$

This objective is, in general, doubly intractable: it involves summing over the infinite space of all programs and also maximizing over a possibly infinite yet combinatorial space of inductive biases.

Discriminative learning. Discriminative approaches to learning priors over programs assume that each observed datum comes with a paired test set. For example, an observed datum could be a set of input/outputs, and its paired test set could be held out inputs and their desired outputs. Discriminative methods aim to find an inductive bias which minimizes a loss function which depends both on the test set and on the program(s) one predicts when

using that inductive bias:

$$\underset{\text{inductive bias}}{\arg\min} \sum_{n=1}^{N} \text{Loss}\left(\text{test}_n,\ \text{P}\left[\cdot|\text{data}_n, \text{inductive bias}_n\right]\right) \tag{2.5}$$

where $\text{P}\left[\cdot|\text{data}_n, \text{inductive bias}_n\right]$ is the posterior one obtains when training on data_n when using the inductive bias. For example, in [36, 121], the authors consider learning an inductive bias over functions mapping strings to strings. Writing $(\text{program} \vdash \text{data})$ to mean that program is consistent with the input/outputs in data, and writing test_n for the test inputs/outputs corresponding to data_n, the work in [36] performs

$$\underset{\text{inductive bias}}{\arg\max} \sum_{n=1}^{N} \underbrace{\sum_{\text{program}} \text{P}\left[\text{program}|\text{data}_n, \text{inductive bias}\right] \mathbb{1}\left[\text{program} \vdash \text{test}_n\right]}_{=-\text{Loss}(\text{test}_n,\ \text{P}[\cdot|\text{data}_n, \text{inductive bias}])}$$

where

$$\text{P}\left[\text{program}|\text{data}, \text{inductive bias}\right] = \frac{\text{P}\left[\text{data}|\text{program}\right] \text{P}\left[\text{program}|\text{inductive bias}\right]}{\sum_{\text{program}'} \text{P}\left[\text{data}|\text{program}'\right] \text{P}\left[\text{program}'|\text{inductive bias}\right]}$$

$$\text{P}\left[\text{data}|\text{program}\right] = \mathbb{1}\left[\text{program} \vdash \text{data}\right]$$

while the work in [121] employs a similar objective, but with a max-margin loss function rather than a posterior predictive. Discriminative methods are powerful when one has access to held out test instances, and cares solely about generalization. In this sense they are similar in spirit to contemporary meta-learning approaches from the deep learning community [45].

2.2.1 Learning continuous weights for parametric program priors

The most basic approach to defining priors over programs is to define a parametric class of probability distributions over program syntax trees. For example, one can define a context

free grammar (CFG) generating the space of valid expressions, and equip each production rule in that grammar with a probability, hence defining a probabilistic context free grammar (PCFG) over valid syntax trees. More generically, one can write down an arbitrary feature extractor over syntax trees, and then construct e.g. a log linear model [36, 38]. No matter what the details, the key feature of these approaches is that they boil down the entire prior to a fixed-dimensional vector. In the absence of program-level supervision, one can estimate this vector by performing a Expectation-Maximization like procedure (EM: [31]) upon Equation 2.4, updating the fixed-dimensional vector during the "Maximize" step and assigning a soft weighting over possible programs during the "Estimate" step.

The advantage of this basic approach is that it allows injecting of strong prior knowledge through the structure of the context free grammar. But it has the disadvantage that, if this grammar is not well suited to the synthesis tasks at hand, there may be no setting of the parameter vector which assigns high probability to programs consistent with the data. In essence this trades lower variance for higher bias.

2.2.2 Learning the symbolic structure of a language bias

Rather than simply learning probabilities attached to a fixed language, as done in Section 2.2.1, could we learn something like a DSL? In this thesis we focus on a restricted form of DSL induction: library learning, or inducing inventories of reusable functions, as a way of learning a language bias. Formally, library learning corresponds to inducing a set of functions, written $\mathcal{D}$, such that programs written using these library functions are likely to solve the program induction problems:

$$\arg\max_{\mathcal{D}} \mathrm{P}\left[\mathcal{D}\right] \prod_{n=1}^{N} \sum_{\text{program}_n} \mathrm{P}\left[\text{data}_n|\text{program}_n\right] \mathrm{P}\left[\text{program}_n|\mathcal{D}\right]$$

This objective is, in general, doubly intractable: it involves summing over the infinite space of all programs and also maximizing over a possibly infinite combinatorial space of possible libraries. While efficiently performing this learning operation is a major theme of Chapters 4 and 5, for now we will summarize existing work in this area.

Recent AI research has developed frameworks for learning libraries by inferring reusable pieces of code [37, 30, 81, 83]. These systems typically work through either memoization (caching reused subtrees, i.e. [30, 83], or reused command sequences, as an [78]), or antiunification (caching reused tree-templates that unify with program syntax trees, i.e. [37, 64, 59]). 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 [94]; within genetic programming, 'Automatically Defined Functions' (ADF: [104]) play a similar role. Both ADF and predicate invention can be seen as the single-task analogues of the hierarchical Bayesian library-learning techniques discussed above. Within *probabilistic* program synthesis, Bayesian program merging [64] is an approach for synthesizing probabilistic generative models specified as programs. Hwang et al. [64] 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 — a kind of refactoring — as well as search moves specialized to probabilistic programs, such as 'deargumentation' (see [64]). 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 library learning.

2.3 Learning to search for programs

2.3.1 Exploiting inductive biases to search faster

Learning an inductive bias can aid search, as introduced in the Exploration-Compression algorithm [30], particularly for simple enumerative search strategies. In particular, if our search algorithm just enumerates programs in decreasing order of probability under the prior, then the search time is connected to the prior probability of the target program. If a MAP program for a particular induction task is program*, then

$$\begin{aligned}(\text{search time}) &\propto (\#\text{ programs enumerated}) \\ &\leq |\left\{\text{program} \ : \ \mathrm{P}\left[\text{program}|\text{inductive bias}\right] \geq \mathrm{P}\left[\text{program}^*|\text{inductive bias}\right]\right\}| \\ &\leq \frac{1}{\mathrm{P}\left[\text{program}^*|\text{inductive bias}\right]} \end{aligned} \tag{2.6}$$

and, taking logs on each side, we get that the logarithm of the total search time is upper bounded by the description length of the target program ($-\log \mathrm{P}\left[\text{program}^*|\text{inductive bias}\right]$), up to an additive constant which is independent of the program. Thus by minimizing the total description lengths of a corpus of programs, we also minimize an upper bound on the geometric mean of the search times. To the extent that minimizing description length of training programs generalizes to minimizing description length of test programs, we can expect that this strategy will lead to faster search times on held out problems.

Nevertheless, this approach is "blind" in a certain sense: the way in which it searches for a program is independent of the observed data corresponding to that program. Next, we will consider approaches which use the problem-specific observed data to inform the search process.

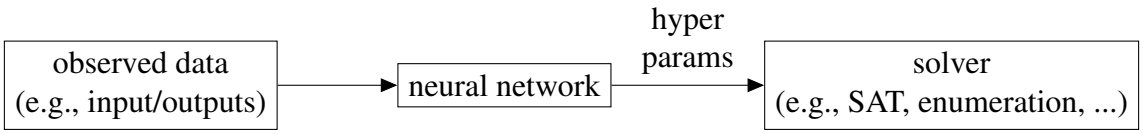


Figure 2-1: Learning-aided solving. A learned model, such as a neural network, is executed on the observed data, and predicts a setting of search hyper parameters which are passed to a downstream solver.

2.3.2 Exploiting per-problem observations to search faster

Can we learn to exploit patterns in the structure of the observed data to guide the search over the space of programs? For example, if we are inducing a program that edits text from input/outputs and we recognize that the outputs are consistently shorter than the inputs, then we could bias the search toward extracting substrings of the input. We will cover several families of approaches with this flavor: approaches which learn to map from observed data to a setting of search hyper parameters for a backend solver (**Learning-aided solving**); neural networks which predict a probability distribution over programs (**Neural recognition models**); and methods for incrementally synthesizing programs that interleave execution with a learned code-proposer (**Execution-guided program induction**).

Learning-aided solving

This family of approaches, best exemplified in the DeepCoder system [10] and illustrated in Figure 2-1, learn a model which maps from observed data (such as input/outputs) to a setting of search hyper parameters for a solver. The key advantage of these approaches is that they can leverage sophisticated solving techniques, such as constraint solvers, and are not bottlenecked by expensive neural network calculations, which only need to be performed once per induction problem.

The DeepCoder system trains a neural network to regress from input/outputs to the

probability of each DSL component being used in the shortest program consistent with those inputs/outputs. This mapping from DSL component to probability can then be used for a number of downstream solvers. In [10] they introduce a solving approach called 'Sort and Add' which sorts the DSL components by probability and iteratively executes an enumerative solver with just the most likely component, then the top two most likely components, then the top three, etc. DeepCoder draws on ideas from an earlier Learning-aided solving approach introduced in [89]. A stochastic-search analog of DeepCoder is introduced in [17], where the probabilities of different DSL components are used to bias proposals within a stochastic search.

Although learning plays a role in approaches such as these, much of the heavy lifting is performed by the backend solver, and the extent to which the learned model can inform the search process is bottlenecked by the choice of solver hyper parameters. Next we consider more generic approaches for guiding search which lean more heavily on learning mechanisms.

Neural recognition models

Neural recognition model approaches train powerful neural networks to directly predict a distribution over program source code. With this distribution in hand, one may then draw samples from this distribution [89], enumerate in decreasing order under this distribution [37], or, if this is intractable, approximate this enumeration via beam search [33]. Certain classes of neural recognition models are special cases of Learning-aided solving: For example, if the distribution is parameterized by weights in a PCFG, then one may use a downstream solver that inputs these weights [89, 37]. On the other hand, if the neural network is more powerful, such as a RNN, then searching using this distribution requires repeatedly querying the neural network to unroll further tokens in the program [33].

In reference to Figure 2-2, these neural recognition models play the role of a **inference strategy**, and, in keeping with the notation of the Helmholtz machine [61], we will notate the distribution predicted by these networks as $Q(\text{program}|\text{data})$. If the objective is just to find the most likely program as quickly as possible, then maximize the following training objective:

$$\log Q\left(\left(\arg\max_{\text{program}} \text{P}\left[\text{program}|\text{data}, \text{inductive bias}\right]\right) \;\middle|\; \text{data}\right) \tag{2.7}$$

as done in e.g. [33]. On the other hand if the goal is to represent uncertainty and ambiguity by modeling the full posterior, then one minimizes the training objective

$$\text{KL}(\text{P}\left[\cdot|\text{data}, \text{inductive bias}\right] \;||\; Q(\cdot|\text{data})) \tag{2.8}$$

Much recent work has explored sophisticated neural network architectures for implementing conditional probabilistic models over source code, using recent ideas such as TranX [147], tree neural networks [5], and sophisticated use of attention mechanisms [33].

Despite this sophistication however, the network is still tasked with generating the code wholesale: it does not get to run the code and compare it with the desired output, as a human might during an interactive coding session. These methods also inherit limitations one sees in language models, where the coherence of the generated text can fall off as the length of the utterance increases. This issue is especially important for programs, which may require global reasoning across the entire body of source code. Indeed, the work in [150] presents empirical findings suggesting that these approaches do not scale to the lengthy programs found in the wild. Next we survey approaches which address some of the shortcomings by augmenting neural methods with online guidance from execution traces.

Execution-guided program induction

Execution-guided program induction approaches are based on the following insight: It is much easier to write a small amount of code, run it to see what it does, and then continue writing more code, than it is to output the entire source code of a complicated program. Concretely, rather than output a (distribution over) an entire program, these approaches learn to predict only enough code to change the execution state of the program so far; this new code is executed, the execution state updated, and the process repeats. This is roughly analogous to writing code inside of an interpreter.

We believe that execution-guided approaches are best framed as a Markov Decision Process (MDP) where the states correspond to intermediate executions of partial programs, the actions correspond to small units of code, and the transition function updates the execution based on the most recently written unit of code. This scheme was independently proposed in [21, 151], who both found that it allows synthesis of much larger programs.

A contribution of this thesis is a reframing of execution-guided synthesis in terms of a MDP, facilitating new improvements that draw on ideas from planning and reinforcement learning. In Section 3.2 we will introduce a new algorithm that trains a policy to write code based on the current execution state, a value function which assesses the long-term prospects of the current execution state, and which wraps both these learned models in an explicit tree search over the space of states in the MDP, roughly analogous to TD-Gammon [133], Expert-Iteration [8], and AlphaGo [119]. In Chapter 4 we will introduce a related algorithm for incrementally synthesizing programs that leverages a SAT solver, but which does not use learning mechanisms.

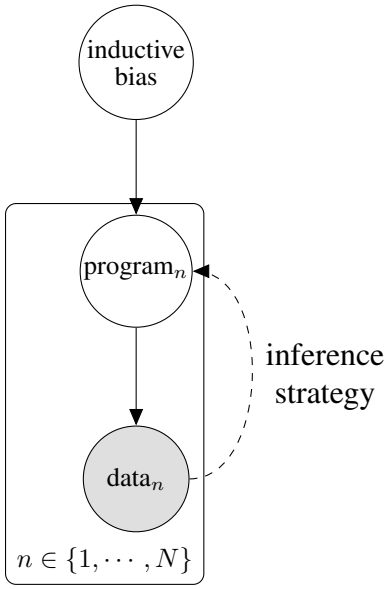


Figure 2-2: A Bayesian approach to program induction. We assume a distantly supervised setting where programs are unobserved. The exact nature of each observation (data) depends on the domain: for instance, for graphics programs the data is an image; for programs that edit text, the data is a set of input strings paired with output strings. The inductive bias captures cross-program regularities. The inference strategy is optional, and corresponds to a learned model which assists in mapping from observed data to unobserved program.

2.4 Learning of neurosymbolic programs

While the learning-aided search methods introduced in this thesis instantiate 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 [7] with symbolic programmatic structure, as recently proposed in the DeepProbLog [84] and HOUDINI [140]. HOUDINI furthermore also reuses these neural modules in a multitask setting (in a manner roughly analogous to dependent learning [83]), 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.

Chapter 3

Programs and Visual Perception

Human vision is rich – we infer shape, objects, parts of objects, and relations between objects – and vision is also abstract: we can perceive the radial symmetry of a spiral staircase, the iterated repetition in the Ising model, see the forest for the trees, and also the recursion within the trees. How could we build an agent with similar visual inference abilities? As a step in this direction, we cast this problem as program learning, and take as our goal to learn high–level graphics programs from simple 2D drawings (Chapter 3.1) and 3D shapes (Chapter 3.2 onward). Chapter 3.1 adopts a pipeline approach, where a neural network first converts an image to a symbolic representation, and then a symbolic solver synthesizes a high-level program from this representation. Chapter 3.2 generalizes ideas in Chapter 3.1 to arrive at a single-stage algorithm which interleaves neural networks with symbolic search to construct code directly from images. This chapter will also illustrate certain principles of program induction, namely learning-to-learn, learning-to-search, and interfacing symbolic abstraction with perceptual data.

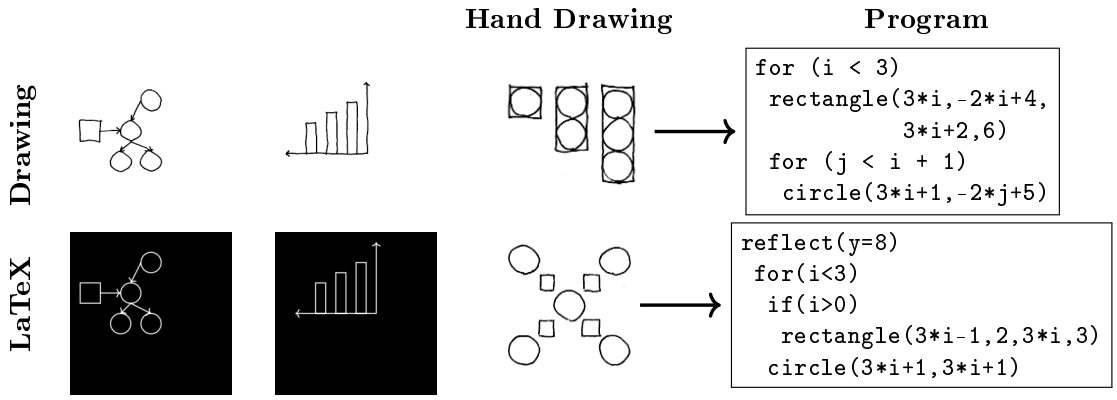


Figure 3-1: **Left**: Model learns to convert hand drawings (top) into LaTeX (rendered below). **Right**: Learns to synthesize high-level graphics program from hand drawing.

3.1 Learning to Infer Graphics Programs from Hand Drawings

The first graphics programs we consider make figures like those found in machine learning papers (Fig. 3-1), and capture high-level features like symmetry, repetition, and reuse of structure.

The key observation behind our work is that going from pixels to programs involves two distinct steps, each requiring different technical approaches. The first step involves inferring what objects make up an image – for diagrams, these are things like as rectangles, lines and arrows. The second step involves identifying the higher-level visual concepts that describe how the objects were drawn. In Fig. 3-1 (right), it means identifying a pattern in how the circles and rectangles are being drawn that is best described with two nested loops, and which can easily be extrapolated to a bigger diagram.

This two-step factoring can be framed as probabilistic inference in a generative model where a latent program is executed to produce a set of drawing commands, which are then rendered to form an image (Fig. 3-2). We refer to this set of drawing commands as a

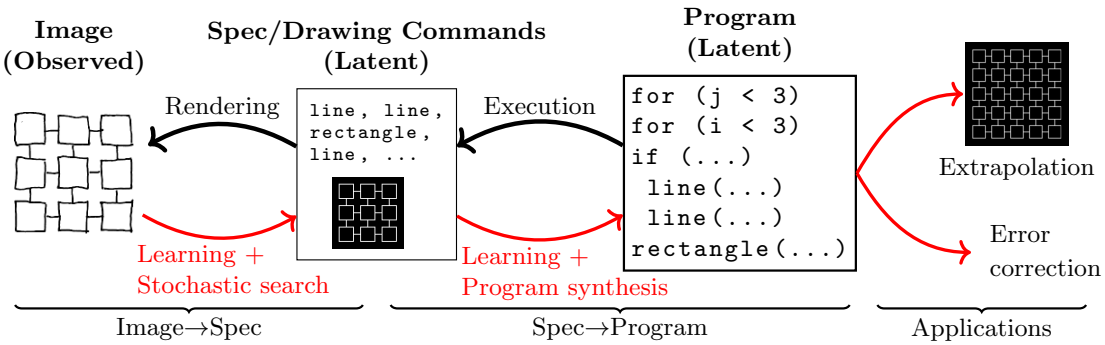


Figure 3-2: Black arrows: Top–down generative model; Program→Spec→Image. Red arrows: Bottom–up inference procedure. **Bold:** Random variables (image/spec/program)

specification (spec) because it specifies what the graphics program drew while lacking the high-level structure determining how the program decided to draw it. We infer a spec from an image using stochastic search (Sequential Monte Carlo) and infer a program from a spec using constraint-based program synthesis [124] – synthesizing structures like symmetries, loops, or conditionals. In practice, both stochastic search and program synthesis are prohibitively slow, and so we learn models that accelerate inference for both programs and specs, in the spirit of "amortized inference" [99], training a neural network to amortize the cost of inferring specs from images and using a variant of Bias–Optimal Search [116] to amortize the cost of synthesizing programs from specs.

The new contributions of this work are (1) a working model that can infer high-level symbolic programs from perceptual input, and (2) a technique for using learning to amortize the cost of program synthesis, described in Section 3.1.2.

3.1.1 Neural architecture for inferring specs

We developed a deep network architecture for efficiently inferring a spec, S, from a hand-drawn image, I. Our model combines ideas from Neurally-Guided Procedural Models [110]

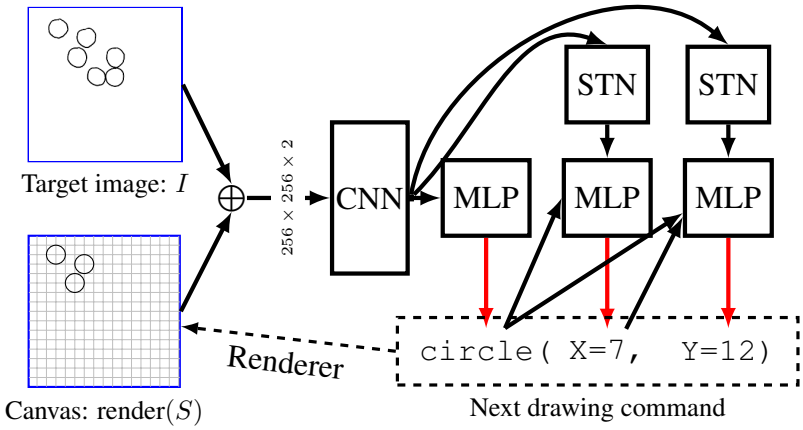


Figure 3-3: Neural architecture for inferring specs from images. Blue: network inputs. Black: network operations. Red: draws from a multinomial. `Typewriter font`: network outputs. Renders on a 16×16 grid, shown in gray. STN: differentiable attention mechanism [65]. Appendix B.1.1 gives further details on the neural network architecture and training.

and Attend-Infer-Repeat [42], but we wish to emphasize that one could use many different approaches from the computer vision toolkit to parse an image in to primitive drawing commands (in our terminology, a "spec") [146]. Our network constructs the spec one drawing command at a time, conditioned on what it has drawn so far (Fig. 3-3). We first pass a 256×256 target image and a rendering of the drawing commands so far (encoded as a two-channel image) to a convolutional network. Given the features extracted by the convnet, a multilayer perceptron then predicts a distribution over the next drawing command to execute (see Tbl. 3.1). We also use a differentiable attention mechanism (Spatial Transformer Networks: [65]) to let the model attend to different regions of the image while predicting drawing commands. We currently constrain coordinates to lie on a discrete 16×16 grid, but the grid could be made arbitrarily fine.

We trained our network by sampling specs S and target images I for randomly gener-

Table 3.1: Primitive drawing commands currently supported by our model.

`circle`(x, y)	Circle at (x, y)
`rectangle`(x_1, y_1, x_2, y_2)	Rectangle with corners at (x_1, y_1) & (x_2, y_2)
`line`$(x_1, y_1, x_2, y_2,$ arrow $\in \{0, 1\}$, dashed $\in \{0, 1\})$	Line from (x_1, y_1) to (x_2, y_2), optionally with an arrow and/or dashed
`STOP`	Finishes spec inference

ated scenes[1] and maximizing $\mathrm{P}_\theta[S|I]$, the likelihood of S given I, with respect to model parameters θ, by gradient ascent. We trained on 10^5 scenes, which takes a day on an Nvidia TitanX GPU. Training does not require backpropagation across the sequence of drawing commands: drawing to the canvas 'blocks' the gradients, effectively offloading memory to an external visual store. This approach is like an autoregressive version of Attend-Infer-Repeat [42], but critically, without a learned recurrent memory state, e.g. an RNN. We can do without an RNN because we learn from ground truth (image, trace) pairs. This allows us to handle scenes with many objects without worrying about the conditioning of gradients as they propagate over long sequences, and makes training more straightforward: it is just maximum likelihood estimation of the model parameters.

Our network can "derender" random synthetic images by doing a beam search to recover specs maximizing $\mathrm{P}_\theta[S|I]$. But, if the network predicts an incorrect drawing command, it has no way of recovering from that error. For added robustness we treat the network outputs as proposals for a Sequential Monte Carlo (SMC) sampling scheme [35]. Our SMC sampler draws samples from the distribution $\propto L(I|\mathrm{render}(S))\mathrm{P}_\theta[S|I]$, where $L(\cdot|\cdot)$ uses the pixel-wise distance between two images as a proxy for a likelihood. Here, the network is learning a proposal distribution to amortize the cost of inverting a generative model (the renderer) [99].

[1]Because rendering ignores ordering we put the drawing commands into a canonical order: top-down left-or-right, first circles, then rectangles, then lines.

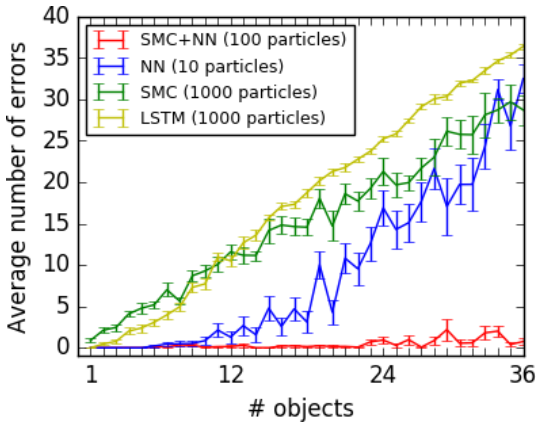
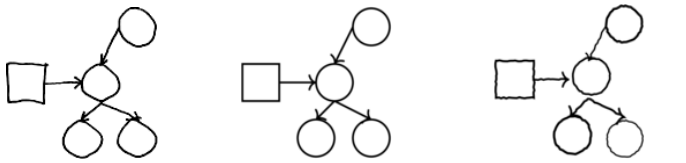


Figure 3-4: Parsing LaTeX output after training on diagrams with ≤ 12 objects. Out-of-sample generalization: Model generalizes to scenes with many more objects ($\approx$ at ceiling when tested on twice as many objects as were in the training data). Neither SMC nor the neural network are sufficient on their own. # particles varies by model: we compare the models *with equal runtime* (≈ 1 sec/object). Average number of errors is (# incorrect drawing commands predicted by model)+(# correct commands that were not predicted by model).

Experiment: Figure 3-4. To evaluate which components of the model are necessary to parse complicated scenes, we compared the neural network with SMC against the neural network by itself (i.e., w/ beam search) or SMC by itself. Only the combination of the two passes a critical test of generalization: when trained on images with ≤ 12 objects, it successfully parses scenes with many more objects than the training data. We compare with a baseline that produces the spec in one shot by using the CNN to extract features of the input which are passed to an LSTM which finally predicts the spec token-by-token (LSTM in Fig. 3-4). This architecture is used in several successful neural models of image captioning (e.g., [143]), but, for this domain, cannot parse cluttered scenes with many objects.



hand drawing | rendering of (a)'s inferred spec | noisy rendering of (b)

Figure 3-5: Noisy renderings produced in LaTeX Ti*k*Z w/ `pencildraw` package. We additionally perturb object positions, sizes, and intensities by small random amounts.

Generalizing to real hand drawings

We trained the model to generalize to hand drawings by introducing noise into the renderings of the training target images, where the noise process mimics the kinds of variations found in hand drawings (Figure 3-5). While our neurally-guided SMC procedure used pixel-wise distance as a surrogate for a likelihood function ($L(\cdot|\cdot)$ in Sec. 3.1.1), pixel-wise distance fares poorly on hand drawings, which never exactly match the model's renders. So, for hand drawings, we learn a surrogate likelihood function, $L_{\text{learned}}(\cdot|\cdot)$. The density $L_{\text{learned}}(\cdot|\cdot)$ is predicted by a convolutional network that we train to predict the distance between two specs conditioned upon their renderings. We train $L_{\text{learned}}(\cdot|\cdot)$ to approximate the symmetric difference, which is the number of drawing commands by which two specs differ:

$$-\log L_{\text{learned}}(\text{render}(S_1)|\text{render}(S_2)) \approx |S_1 - S_2| + |S_2 - S_1| \tag{3.1}$$

We implement L_{learned} by training it to predict two scalars: $|S_1 - S_2|$ and $|S_2 - S_1|$. These predictions are made using linear regression from the image features followed by a ReLU nonlinearity; this nonlinearity makes sense because the predictions can never be negative but could be arbitrarily large positive numbers. We train this network by sampling random synthetic scenes for S_1, and then perturbing them to produce S_2, by randomly adding and removing objects. We minimize the squared loss between the network's prediction and the

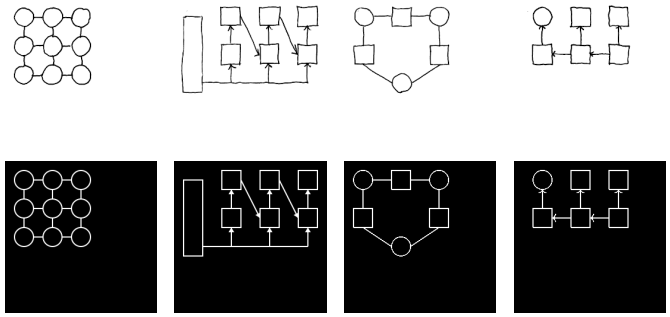


Figure 3-6: Left to right: Ising model, recurrent network architecture, figure from a deep learning textbook [53], graphical model

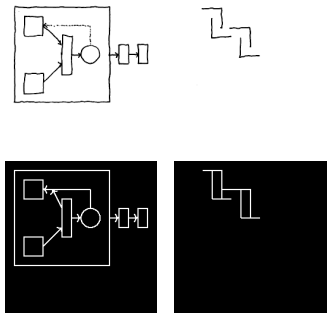


Figure 3-7: Near misses. Rightmost: illusory contours (note: no SMC in rightmost)

ground truth symmetric differences. S_1 is rendered in the "simulated hand drawing" style.

Experiment: Figures 3-6–3-8. We evaluated, but did not train, our system on 100 real hand-drawn figures; see Fig. 3-6–3-7. These were drawn carefully but not perfectly with the aid of graph paper. For each drawing we annotated a ground truth spec and had the neurally guided SMC sampler produce 10^3 samples. For 63% of the drawings, the Top-1 most likely sample exactly matches the ground truth; with more samples, the model finds specs that are closer to the ground truth annotation (Fig. 3-8). We will show that the program synthesizer corrects some of these small errors (Sec. 3.1.3).

3.1.2 Synthesizing graphics programs from specs

Although the spec describes the contents of a scene, it does not encode higher-level features of an image such as repeated motifs or symmetries, which are more naturally captured by a graphics program. We seek to synthesize graphics programs from their specs.

We constrain the space of programs by writing down a context free grammar over programs – what in the programing languages community is called a Domain Specific

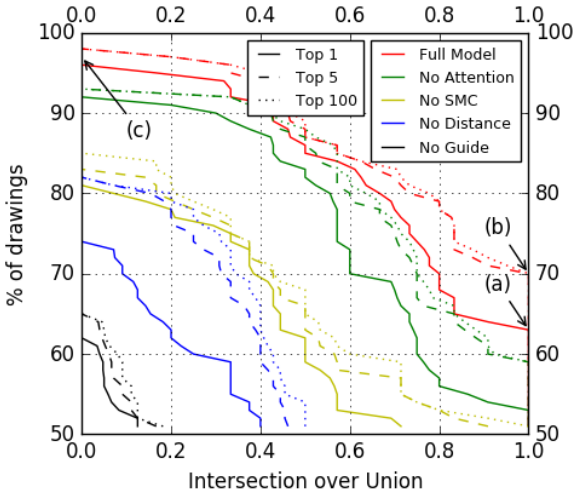


Figure 3-8: How close are the model's outputs to the ground truth on hand drawings, as we consider larger sets of samples (1, 5, 100)? Distance to ground truth measured by the intersection over union (IoU) of predicted spec vs. ground truth spec: IoU of sets (specs) A and B is $|A \cap B|/|A \cup B|$. (a) for 63% of drawings the model's top prediction is exactly correct; (b) for 70% of drawings the ground truth is in the top 5 model predictions; (c) for 4% of drawings all of the model outputs have no overlap with the ground truth. Red: the full model. Other colors: lesioned versions of our model.

Language (DSL) [105]. Our DSL (Tbl. 3.2) encodes prior knowledge of what graphics programs tend to look like.

Table 3.2: Grammar over graphics programs. We allow loops (`for`) with conditionals (`if`), vertical/horizontal reflections (`reflect`), variables (Var) and affine transformations ($\mathbb{Z}\times$Var+$\mathbb{Z}$).

Program→	Statement; $\cdots$; Statement
Statement→	`circle`(Expression,Expression)
Statement→	`rectangle`(Expression,Expression,Expression,Expression)
Statement→	`line`(Expression,Expression,Expression,Expression,Boolean,Boolean)
Statement→	`for`$(0 \leq$ Var $<$ Expression$)$ `{ if` $($Var $> 0)$ `{` Program `}; ` Program `}`
Statement→	`reflect`(Axis) `{` Program `}`
Expression→	$\mathbb{Z}\times$Var+$\mathbb{Z}$
Axis→	`X` $= \mathbb{Z}$ | `Y` $= \mathbb{Z}$
$\mathbb{Z}\rightarrow$	an integer

Given the DSL and a spec S, we want a program that both satisfies S and, at the same time, is the "best" explanation of S. For example, we might prefer more general programs or, in the spirit of Occam's razor, prefer shorter programs. We wrap these intuitions up into a cost function over programs, and seek the minimum cost program consistent with S:

$$\text{program}(S) = \underset{p \in \text{DSL}}{\arg\max}\, \mathbb{1}\left[p \text{ consistent w/ } S\right] \exp\left(-\text{cost}(p)\right) \tag{3.2}$$

We define the cost of a program as follows: Programs incur a cost of 1 for each 'Statement' (Tbl. 3.2). They incur a cost of $\frac{1}{3}$ for each unique coefficient they use in a linear transformation beyond the first coefficient. This encourages reuse of coefficients, which leads to code that has translational symmetry; rather than provide a translational symmetry operator as we did with reflection, we modify what is effectively a prior over the space of program so that it tends to produce programs that have this symmetry. Programs also incur a cost of 1 for having loops of constant length 2; otherwise there is often no pressure from the cost function to explain a repetition of length 2 as being a reflection rather a loop.

Returning to the generative model in Fig. 3-2, this setup is the same as saying that the prior probability of a program p is $\propto \exp\left(-\text{cost}(p)\right)$ and the likelihood of a spec S given a program p is $\mathbb{1}[p \text{ consistent w/ } S]$.

The constrained optimization problem in Eq. 3.2 is intractable in general, but there exist efficient-in-practice tools for finding exact solutions to such program synthesis problems. We use the state-of-the-art Sketch tool [124]. Sketch takes as input a space of programs, along with a specification of the program's behavior and optionally a cost function. It translates the synthesis problem into a constraint satisfaction problem and then uses a SAT solver to find a minimum-cost program satisfying the specification. Sketch requires a *finite program space*, which here means that the depth of the program syntax tree is bounded (we set the bound to 3), but has the guarantee that it always eventually finds a globally

optimal solution. In exchange for this optimality guarantee it comes with no guarantees on runtime. For our domain synthesis times vary from minutes to hours, with 27% of the drawings timing out the synthesizer after 1 hour. Tbl. 3.3 shows programs recovered by our system. A main impediment to our use of these general techniques is the prohibitively high cost of searching for programs. We next describe how to learn to synthesize programs much faster (Sec. 3.1.2), timing out on 2% of the drawings and solving 58% of problems within a minute.

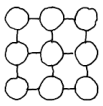
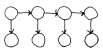
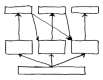
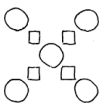
Learning a search policy for synthesizing programs

We want to leverage powerful, domain-general techniques from the program synthesis community, but make them much faster by learning a domain-specific **search policy**. A search policy poses search problems like those in Eq. 3.2, but also offers additional constraints on the structure of the program (Tbl. 3.4). For example, a policy might decide to first try searching over small programs before searching over large programs, or decide to prioritize searching over programs that have loops.

A search policy $\pi_\theta(\sigma|S)$ takes as input a spec S and predicts a distribution over synthesis problems, each of which is written σ and corresponds to a set of possible programs to search over (so $\sigma \subseteq$ DSL). Good policies will prefer tractable program spaces, so that the search procedure will terminate early, but should also prefer program spaces likely to contain programs that concisely explain the data. These two desiderata are in tension: tractable synthesis problems involve searching over smaller spaces, but smaller spaces are less likely to contain good programs. Our goal now is to find the parameters of the policy, written θ, that best navigate this trade-off.

Given a search policy, what is the best way of using it to quickly find minimum cost programs? We use a bias-optimal search algorithm (c.f. Schmidhuber 2004 [116]):

Table 3.3: Drawings (left), their specs (middle left), and programs synthesized from those specs (middle right). Compared to the specs the programs are more compressive (right: programs have fewer lines than specs) and automatically group together related drawing commands. Note the nested loops and conditionals in the Ising model, combination of symmetry and iteration in the bottom figure, affine transformations in the top figure, and the complicated program in the second figure to bottom.

Drawing	**Spec**	**Program**	**Compression factor**
	`Line(2,15, 4,15)` `Line(4,9, 4,13)` `Line(3,11, 3,14)` `Line(2,13, 2,15)` `Line(3,14, 6,14)` `Line(4,13, 8,13)`	`for(i<3)` `  line(i,-1*i+6,` `  2*i+2,-1*i+6)` `  line(i,-2*i+4,i,-1*i+6)`	$\frac{6}{3} = 2$x
	`Circle(5,8)` `Circle(2,8)` `Circle(8,11)` `Line(2,9, 2,10)` `Circle(8,8)` `Line(3,8, 4,8)` `Line(3,11, 4,11)` *... etc. ...; 21 lines*	`for(i<3)` `  for(j<3)` `  if(j>0)` `  line(-3*j+8,-3*i+7,` `  -3*j+9,-3*i+7)` `  line(-3*i+7,-3*j+8,` `  -3*i+7,-3*j+9)` `  circle(-3*j+7,-3*i+7)`	$\frac{21}{6} = 3.5$x
	`Rectangle(1,10,3,11)` `Rectangle(1,12,3,13)` `Rectangle(4,8,6,9)` `Rectangle(4,10,6,11)` *... etc. ...; 16 lines*	`for(i<4)` `  for(j<4)` `  rectangle(-3*i+9,-2*j+6,` `  -3*i+11,-2*j+7)`	$\frac{16}{3} = 5.3$x
	`Line(11,14,13,14,arrow)` `Circle(10,10)` `Line(10,13,10,11,arrow)` `Circle(6,10)` *... etc. ...; 15 lines*	`for(i<4)` `  line(-4*i+13,4,-4*i+13,2,arrow)` `  for(j<3)` `  if(j>0)` `  circle(-4*i+13,4*j+-3)` `  line(-4*j+10,5,-4*j+12,5,` `  arrow)`	$\frac{15}{6} = 2.5$x
	`Line(3,10,3,14,arrow)` `Rectangle(11,8,15,10)` `Rectangle(11,14,15,15)` `Line(13,10,13,14,arrow)` *... etc. ...; 16 lines*	`for(i<3)` `  line(7,1,5*i+2,3,arrow)` `  for(j<i+1)` `  if(j>0)` `  line(5*j-1,9,5*i,5,arrow)` `  line(5*j+2,5,5*j+2,9,arrow)` `  rectangle(5*i,3,5*i+4,5)` `  rectangle(5*i,9,5*i+4,10)` `rectangle(2,0,12,1)`	$\frac{16}{9} = 1.8$x
	`Circle(2,8)` `Rectangle(6,9, 7,10)` `Circle(8,8)` `Rectangle(6,12, 7,13)` `Rectangle(3,9, 4,10)` *... etc. ...; 9 lines*	`reflect(y=8)` `  for(i<3)` `  if(i>0)` `  rectangle(3*i-1,2,3*i,3)` `  circle(3*i+1,3*i+1)`	$\frac{9}{5} = 1.8$x

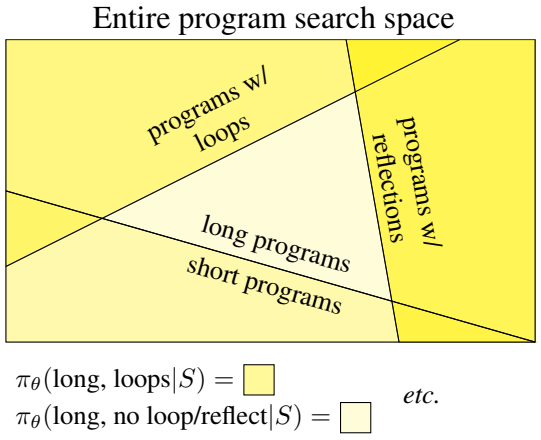


Figure 3-9: The bias-optimal search algorithm divides the entire (intractable) program search space in to (tractable) program subspaces (written σ), each of which contains a restricted set of programs. For example, one subspace might be short programs which don't loop. The policy π predicts a distribution over program subspaces. The weight that π assigns to a subspace is indicated by its yellow shading in the figure to the left, and is conditioned on the spec S.

Definition: Bias-optimality. A search algorithm is *n-bias optimal* with respect to a distribution $\mathrm{P}_{\text{bias}}[\cdot]$ if it is guaranteed to find a solution in σ after searching for at least time $n \times \frac{t(\sigma)}{\mathrm{P}_{\text{bias}}[\sigma]}$, where $t(\sigma)$ is the time it takes to verify that σ contains a solution to the search problem.

Bias-optimal search over program spaces is known as **Levin Search** [79]; an example of a 1-bias optimal search algorithm is an ideal time-sharing system that allocates $\mathrm{P}_{\text{bias}}[\sigma]$ of its time to trying σ. We construct a 1-bias optimal search algorithm by identifying $\mathrm{P}_{\text{bias}}[\sigma] = \pi_\theta(\sigma|S)$ and $t(\sigma) = t(\sigma|S)$, where $t(\sigma|S)$ is how long the synthesizer takes to search σ for a program for S. Intuitively, this means that the search algorithm explores the entire program space, but spends most of its time in the regions of the space that the policy judges to be most promising. Concretely, this means that we run many different program searches *in parallel* (i.e., run in parallel different instances of the synthesizer, one for each σ), but to allocate compute time to a σ in proportion to $\pi_\theta(\sigma|S)$ (Figure 3-9).

Now in theory any $\pi_\theta(\cdot|\cdot)$ is a bias-optimal searcher. But the actual runtime of the algorithm depends strongly upon the bias $\mathrm{P}_{\text{bias}}[\cdot]$. Our new approach is to learn $\mathrm{P}_{\text{bias}}[\cdot]$ by picking the policy minimizing the expected bias-optimal time to solve a training corpus, $\mathcal{C}$,

of graphics program synthesis problems:

$$\textsc{Loss}(\theta;\mathcal{C}) = \mathrm{E}_{S\sim\mathcal{C}}\left[\min_{\sigma\in\textsc{Best}(S)} \frac{t(\sigma|S)}{\pi_\theta(\sigma|S)}\right] + \lambda\|\theta\|_2^2 \tag{3.3}$$

where $\sigma \in \textsc{Best}(S)$ if a minimum cost program for S is in σ.

To generate a training corpus for learning a policy, we synthesized minimum cost programs for each drawing and for each σ, then minimized Eq. 3.3 using gradient descent while annealing a softened minimum to the hard minimization Eq. 3.3. This softened loss is:

$$\textsc{Loss}_\beta(\theta;\mathcal{C}) = \mathrm{E}_{S\sim\mathcal{C}}\left[\textsc{SoftMinimum}_\beta\left\{\frac{t(\sigma|S)}{\pi_\theta(\sigma|S)} : \sigma \in \textsc{Best}(S)\right\}\right] + \lambda\|\theta\|_2^2 \tag{3.4}$$

$$\text{where } \textsc{SoftMinimum}_\beta(x_1, x_2, x_3, \cdots) = \sum_n x_n \frac{e^{-\beta x_n}}{\sum_{n'} e^{-\beta x_{n'}}} \tag{3.5}$$

Notice that $\textsc{SoftMinimum}_{\beta=\infty}(\cdot)$ is just $\min(\cdot)$ and so $\textsc{Loss}_{\beta=\infty} = \textsc{Loss}$. We set the regularization coefficient $\lambda = 0.1$ and minimize equation 3.4 using Adam for 2000 steps, linearly increasing β from 1 to 2. Because we want to learn a policy from only 100 drawings, we parameterize π with a low-capacity bilinear model with only 96 real-valued parameters:

$$\pi_\theta(\sigma|S) \propto \exp\left(\phi_{\text{params}}(\sigma)^\top \theta \phi_{\text{spec}}(S)\right) \tag{3.6}$$

where:

$$\phi_{\text{params}}(\sigma) = [\mathbb{1}[\sigma \text{ can loop}]; \mathbb{1}[\sigma \text{ can reflect}]; \mathbb{1}[\sigma \text{ is incremental}];$$
$$\mathbb{1}[\sigma \text{ has depth bound } 1]; \mathbb{1}[\sigma \text{ has depth bound } 2]; \mathbb{1}[\sigma \text{ has depth bound } 3];]$$
$$\phi_{\text{spec}}(S) = [\#\text{ circles in } S; \#\text{ rectangles in } S; \#\text{ lines in } S; 1]$$

where 'incremental' means: Rather than give the sketch program synthesizer the entire spec all at once, we instead give it subsets of the spec (subsets of the objects in the image) and ask it to synthesize a program for each subset. We then concatenate the resulting programs from each subset to get a program that explains the entire image. This is not guaranteed to be faster, nor is it guaranteed to find a minimum cost program. Thus we allow the search policy to decide what fraction of our search time should be allocated to this incremental approach to program synthesis. Concretely, we partitioned a spec into its constituent lines, circles, and rectangles.

Experiment: Table 3.5; Figure 3-10. We compare synthesis times for our learned search policy with 4 alternatives: *Sketch*, which poses the entire problem wholesale to the Sketch program synthesizer; *DC*, a DeepCoder–style model that learns to predict which program components (loops, reflections) are likely to be useful and uses the 'Sort and Add' test-time policy given in [10] (see Appendix B.1.3); *End–to-End*, which trains a recurrent neural network to regress directly from images to programs (Appendix B.1.2); and an *Oracle*, a policy which always picks the quickest to search σ also containing a minimum cost program. Our approach improves upon Sketch by itself, and comes close to the Oracle's performance. One could never construct this Oracle, because the agent does not know ahead of time which σ's contain minimum cost programs nor does it know how long each σ will take to search. With this learned policy in hand we can synthesize 58% of programs within a minute.

Table 3.4: Parameterization of different ways of posing the program synthesis problem. The policy learns to choose parameters likely to quickly yield a minimal cost program.

Parameter	Description	Range
Loops?	Is the program allowed to loop?	$\{\text{True}, \text{False}\}$
Reflects?	Is the program allowed to have reflections?	$\{\text{True}, \text{False}\}$
Incremental?	Solve the problem piece-by-piece or all at once?	$\{\text{True}, \text{False}\}$
Maximum depth	Bound on the depth of the program syntax tree	$\{1, 2, 3\}$

3.1.3 Applications of graphics program synthesis

Correcting errors made by the neural network

The program synthesizer corrects errors made by the neural network by favoring specs which lead to more concise or general programs. For example, figures with perfectly aligned objects are preferable, and precise alignment lends itself to short programs. Similarly, figures often have repeated parts, which the program synthesizer might be able to model as a loop or reflectional symmetry. So, in considering several candidate specs proposed by the neural network, we might prefer specs whose best programs have desirable features such being short or having iterated structures.

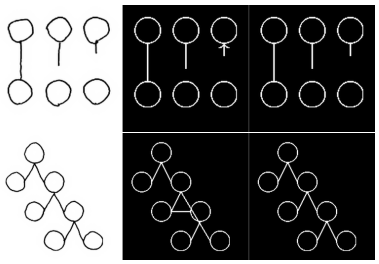


Figure 3-11: Left: hand drawings. Center: interpretations favored by the deep network. Right: interpretations favored after learning a prior over programs. The prior favors simpler programs, thus (top) continuing the pattern of not having an arrow is preferred, or (bottom) continuing the "binary search tree" is preferred.

Concretely, we run the program synthesizer on the Top-k most likely specs output by the neurally guided sampler. Then, the system reranks the Top-k by the prior probability of their programs. The prior probability of a program is learned by optimizing the parameters of the prior so as to maximize the likelihood of the ground truth specs. Mathematically, for an image I, the neurally guided sampling scheme of Section 3.1.1 samples a set of candidate specs, written $\mathcal{F}(I)$. Instead of predicting the most likely spec in $\mathcal{F}(I)$ according

Model	Median search time	Timeouts (1 hr)
Sketch	274 sec	27%
DC	187 sec	2%
End–to–End	63 sec	94%
Oracle	6 sec	2%
Ours	28 sec	2%

Table 3.5: Time to synthesize a minimum cost program. Sketch: out-of-the-box performance of Sketch [124]. DC: Deep–Coder style baseline that predicts program components, trained like [10]. End–to–End: neural net trained to regress directly from images to programs, which fails to find valid programs 94% of the time. Oracle: upper bounds the performance of any bias–optimal search policy. Ours: evaluated w/ 20-fold cross validation.

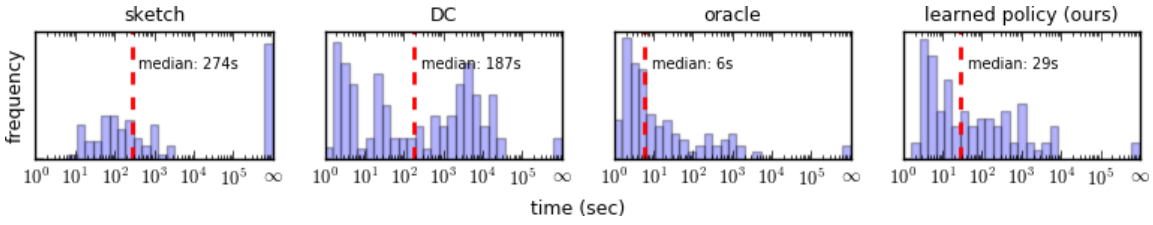


Figure 3-10: Time to synthesize a minimum cost program (compare w/ Table 3.5). End–to–End: not shown because it times out on 96% of drawings, and has its median time (63s) calculated only on non-timeouts, wheras the other comparisons include timeouts in their median calculation. ∞ = timeout. Red dashed line is median time.

to the neural network, we can take into account the programs that best explain the specs. Writing $\hat{S}(I)$ for the spec the model predicts for image I,

$$\hat{S}(I) = \arg\max_{S \in \mathcal{F}(I)} L_{\text{learned}}(I|\text{render}(S)) \times \text{P}[S|I] \times \text{P}_\beta[\text{program}(S)] \tag{3.7}$$

where $\text{P}_\beta[\cdot]$ is a prior probability distribution over programs parameterized by β. This is equivalent to doing MAP inference in a generative model where the program is first drawn from $\text{P}_\beta[\cdot]$, then the program is executed deterministically, and then we observe a noisy version of the program's output, where $L_{\text{learned}}(I|\text{render}(\cdot)) \times \text{P}[\cdot|I]$ is our observation model, and $\text{P}[\cdot|I]$ is the density predicted by the parsing network in Section 3.1.1. Given

a corpus of graphics program synthesis problems with annotated ground truth specs (i.e. (I, S) pairs), we find a maximum likelihood estimate of β:

$$\beta^* = \arg\max_{\beta} \mathrm{E}\left[\log \frac{\mathrm{P}_\beta[\text{program}(S)] \times L_{\text{learned}}(I|\text{render}(S)) \times \mathrm{P}_\theta[S|I]}{\sum_{S' \in \mathcal{F}(I)} \mathrm{P}_\beta[\text{program}(S')] \times L_{\text{learned}}(I|\text{render}(S')) \times \mathrm{P}_\theta[S'|I]}\right] \tag{3.8}$$

where the expectation is taken both over the model predictions and the (I, S) pairs in the training corpus. We define $\mathrm{P}_\beta[\cdot]$ to be a log linear distribution $\propto \exp(\beta \cdot \phi(\text{program}))$, where $\phi(\cdot)$ is a feature extractor for programs. We extract a few basic features of a program, such as its size and how many loops it has, and use these features to help predict whether a spec is the correct explanation for an image.

But, this procedure can only correct errors when a correct spec is in the Top-k. Our sampler could only do better on 7/100 drawings by looking at the Top-100 samples (see Fig. 3-8), precluding a statistically significant analysis of how much learning a prior over programs could help correct errors. But, learning this prior does sometimes help correct mistakes made by the neural network; see Fig. 3-11 for a representative example of the kinds of corrections that it makes.

Extrapolating figures

Having access to the source code of a graphics program facilitates coherent, high-level image editing. For example, we could change all of the circles to squares or make all of the lines be dashed, or we can (automatically) extrapolate figures by increasing the number of times that loops are executed. Extrapolating repetitive visuals patterns comes naturally to humans, and is a practical application: imagine hand drawing a repetitive graphical model structure and having our system automatically induce and extend the pattern. Fig. 3-12 shows extrapolations produced by our system.

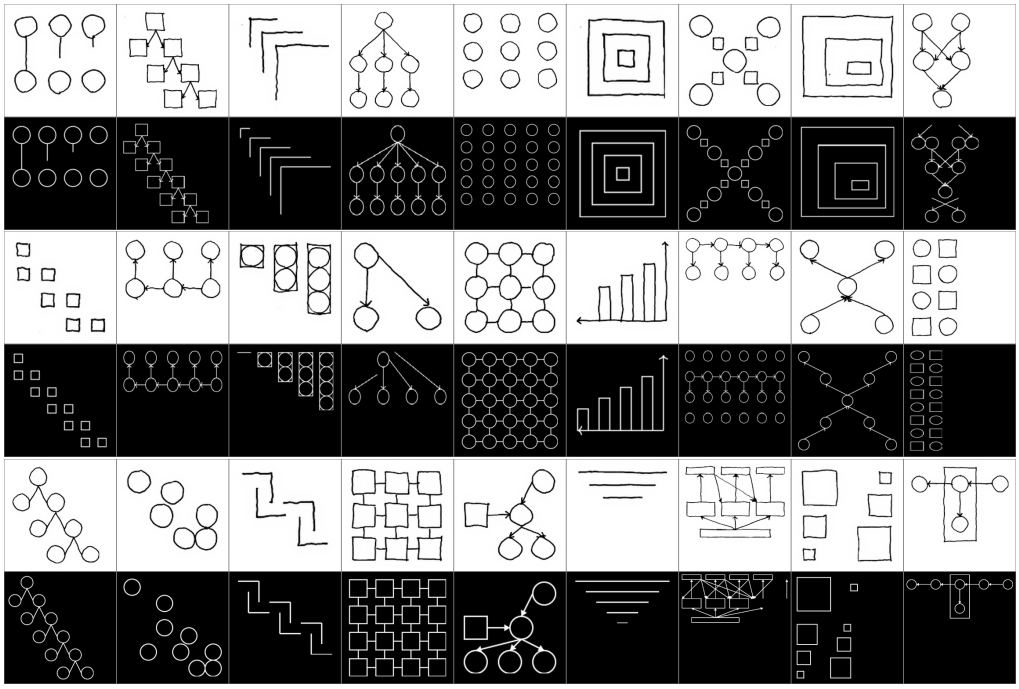


Figure 3-12: Top, white: drawings. Bottom, black: extrapolations automatically produced by our system.

Measuring similarity between drawings

Modeling drawings using programs opens up new ways to measure similarity between them. For example, we might say that two drawings are similar if they both contain loops of length 4, or if they share a reflectional symmetry, or if they are both organized according to a grid-like structure.

We measure the similarity between two drawings by extracting features of the lowest-cost programs that describe them. Our features are counts of the number of times that different components in the DSL were used (Tbl. 3.2). We then find drawings which are either close together or far apart in program feature space. One could use many alternative similarity metrics between drawings which would capture pixel-level similarities while

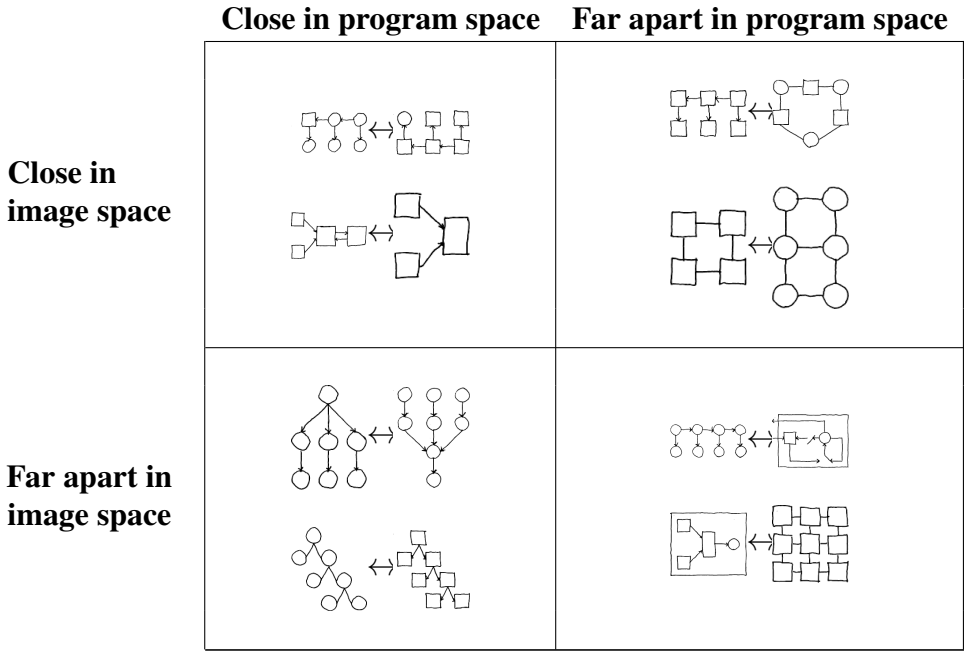


Figure 3-13: Pairs of images either close together or far apart in different features spaces. The symbol $\leftrightarrow$ points to the compared images. Features of the program capture abstract notions like symmetry and repetition. Distance metric over images is $L_{\text{learned}}(\cdot|\cdot)$ (see Section 3.1.1). The off-diagonal entries highlight the difference between these metrics: similarity of programs captures high-level features like repetition and symmetry, whereas similarity of images correspondends to similar drawing commands being in similar places.

missing high-level geometric similarities. We used our learned distance metric between specs, $L_{\text{learned}}(\cdot|\cdot)$, to find drawings that are either close together or far apart according to the learned distance metric over images. Fig. 3-13 illustrates the kinds of drawings that these different metrics put closely together.

3.2 Learning to Search for Programs via a REPL

Following our treatment of 2D graphics routines, we now explore programs that generate 3D objects. Certain key ideas from the 2D setting are generalized and reused: in particular,

we will introduce a general framework for training neural networks to search for programs, inspired by the image→spec pipeline of Section 3.1.

The core insight driving this next section is the following: almost no programmer can write out a large body of code and have it just work on the first try. Writing a large body of code is a process of *trial and error* that alternates between trying out a piece of code, executing it to see if you're still on the right track, and trying out something different if the current execution looks buggy. Crucial to this human work-flow is the ability to *execute* the partially-written code, and the ability to *assess* the resulting execution to see if one should continue with the current approach. Thus, if we wish to build machines that automatically write large, complex programs, designing systems with the ability to effectively transition between states of writing, executing, and assessing the code may prove crucial.

In this work, we present a model that integrates components which write, execute, and assess code to perform a stochastic search over the *semantic* space of possible programs. We do this by equipping our model with one of the oldest and most basic tools available to a programmer: an interpreter, or read-eval-print-loop (REPL), which immediately executes partially written programs, exposing their semantics. The REPL is analogous to the renderer in Section 3.1.1, and addresses a fundamental challenge of program synthesis: tiny changes in syntax can lead to huge changes in semantics. The mapping between syntax and semantics is a difficult relation for a neural network to learn, but comes for free given a REPL. By conditioning the search solely on the execution states rather than the program syntax, the search is performed entirely in the *semantic* space. By allowing the search to branch when it is uncertain, discard branches when they appear less promising, and attempt more promising alternative candidates, the search process emulates the natural iterative coding process used by a human programmer.

In the spirit of systems such as AlphaGo [119], we train a pair of models – a *policy* that proposes new pieces of code to write, and a *value function* that evaluates the long-term

prospects of the code written so far, and deploy both at test time in a symbolic tree search. The policy is analogous to the code-emitting CNN of Figure 3-3, and the value function is analogous to the learned loss of Section 3.1.1. As before, we combine the policy, value, and REPL with a Sequential Monte Carlo (SMC) search strategy at inference time. We sample next actions using our learned policy, execute the partial programs with the REPL, and re-weight the candidates by the value of the resulting partial program state. This algorithm allows us to naturally incorporate writing, executing, and assessing partial programs into our search strategy, while managing a large space of alternative program candidates. In contrast to the previous 2D system, we employ relatively simple symbolic methods while leaning more heavily on learned search heuristics.

Integrating learning and search to tackle the problem of program synthesis is an old idea experiencing a recent resurgence [47, 116, 10, 96, 33, 66, 135, 106]. This work builds on recent ideas termed 'execution-guided neural program synthesis,' independently proposed by [21] and [151], where a neural network writes a program conditioned on intermediate execution states. We extend these ideas along two dimensions. First, we cast these different kinds of execution guidance in terms of interaction with a REPL, and use reinforcement learning techniques to train an agent to both interact with a REPL, *and* to assess when it is on the right track. Prior execution-guided neural synthesizers do not learn to *assess* the execution state, which is a prerequisite for sophisticated search algorithms, like those we explore in this work. Second, we investigate several ways of interleaving the policy and value networks during search, finding that an SMC sampler provides an effective foundation for an agent that writes, executes and assesses its code.

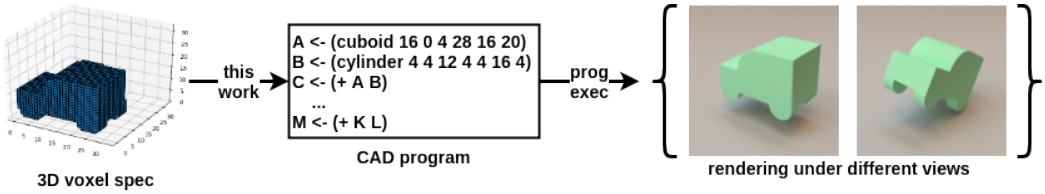


Figure 3-14: Example of program synthesized by our system, showing a graphics program inferred from voxel specification.

3.2.1 An Illustrative Example

To make our framework concrete, consider the following program synthesis task of synthesizing a constructive solid geometry (CSG) representation of a simple 2D scene (see Figure 3-15). CSG is a shape-modeling language that allows the user to create complex renders by combining simple primitive shapes via boolean operators. The CSG program in our example consists of two boolean combinations: union $+$ and subtraction $-$ and two primitives: circles $C^r_{x,y}$ and rectangles $R^{w,h,\theta}_{x,y}$, specified by position x, y, radius r, width and height w, h, and rotation θ. This CSG language can be formalized as the context-free grammar below:

`P ->` $P + P$ `|` $P - P$ `|` $C^r_{x,y}$ `|` $R^{w,h,\theta}_{x,y}$

The synthesis task is to find a CSG program that renders to $spec$. Our policy constructs this program one piece at a time, conditioned on the set of expressions currently in scope. Starting with an empty *set* of programs in scope, $pp = \{\}$, the policy proposes an action a that extends it. This proposal process is iterated to incrementally extend pp to contain longer and more complex programs. In this CSG example, the action a is either adding a primitive shape, such as a circle $C^3_{2,8}$, or applying a boolean combinator, such as $p_1 - p_2$, where the action also specifies its two arguments p_1 and p_2.

To help the policy make good proposals, we augment it with a REPL, which takes a set of programs pp in scope and executes each of them. In our CSG example, the REPL

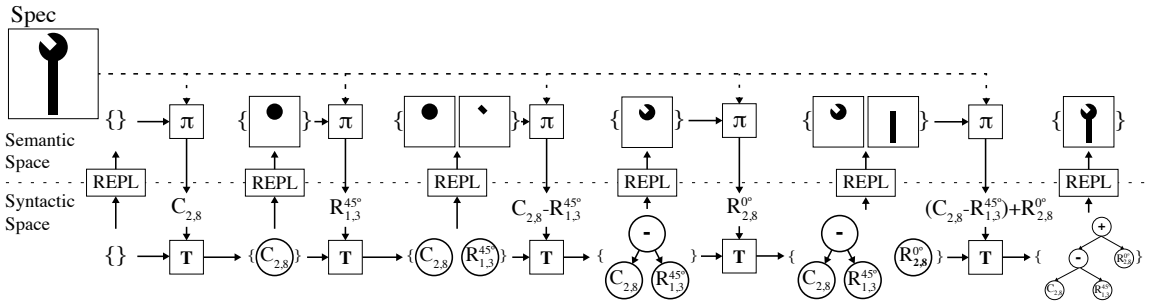


Figure 3-15: A particular trajectory of the policy building a 2D wrench. At each step, the REPL renders the set of partial programs pp into the semantic (image) space. These images are fed into the policy π which proposes how to extend the program via an action a, which is incorporated into pp via the transition T.

renders the set of programs pp to a set of images. The policy then takes in the REPL state (a set of images), along with the specification $spec$ to predict the next action a. This way, the input to the policy lies entirely in the semantic space, akin to how one would use a REPL to iteratively construct a working code snippet. Figure 3-15 demonstrates a potential roll-out through a CSG problem using only the policy.

However, code is brittle, and if the policy predicts an incorrect action, the entire program synthesis fails. To combat this brittleness, we use Sequential Monte Carlo (SMC) to search over the space of candidate programs. Crucial to our SMC algorithm is a learned value function v which, given a REPL state, assesses the likelihood of success on this particular search branch. By employing v, the search can be judicious about which search branch to prioritize in exploring and withdraw from branches deemed unpromising. Figure 3-16 demonstrates a fraction of the search space leading up to the successful program and how the value function v helps to prune out unpromising search candidates.

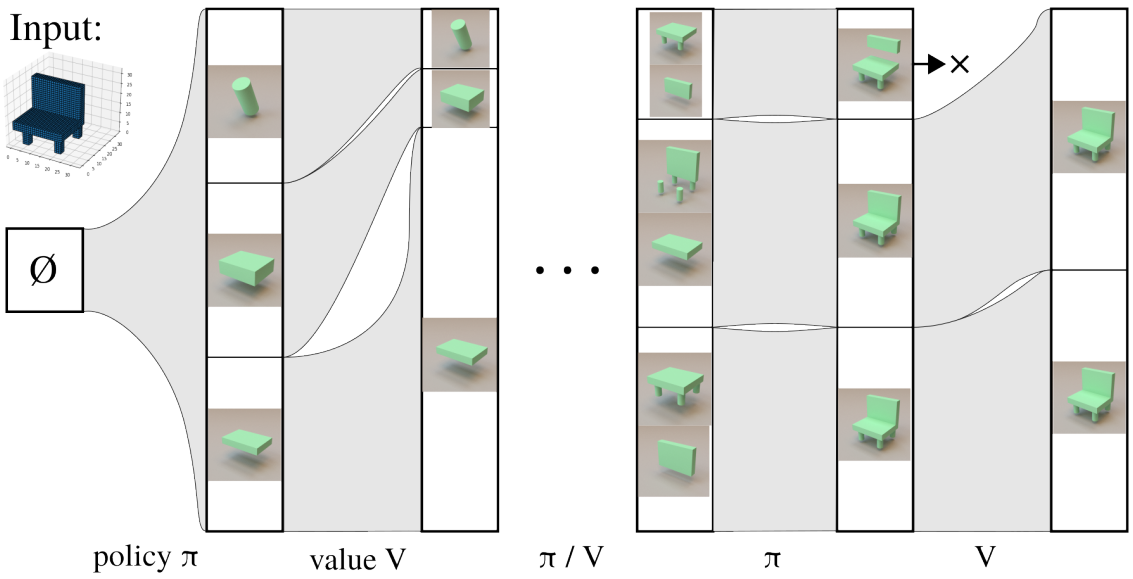


Figure 3-16: SMC sampler maintains a population of particles (i.e. programs), which it evolves forward in time by (1) sampling from policy π to get a new generation of particles, then (2) reweighting by the value function v and resampling to prune unpromising candidates and up-weight promising ones.

3.2.2 Our Approach

Our program synthesis approach integrates components that write, execute, and assess code to perform a stochastic search over the semantic space of programs. Crucial to this approach are three main components: First, the definition of the search space; Second, the training of the code-writing policy and the code-assessing value function; Third, the Sequential Monte Carlo algorithm that leverages the policy and the value to conduct the search by maintaining a population of candidate search branches.

The Semantic Search Space of Programs

The space of possible programs is typically defined by a context free grammar (CFG), which specifies the set of syntactically valid programs. However, when one is writing the code, the programs are often constructed in a piece-wise fashion. Thus, it is natural to express the search space of programs as a markov decision process (MDP) over the set of partially constructed programs.

State The state is a tuple $s = (pp, spec)$ where pp is a *set* of partially-constructed program trees (intuitively, 'variables in scope'), and $spec$ is the goal specification. Thus, our MDP is goal conditioned. The start state is $(\{\}, spec)$.

Action The action a is a production rule from the CFG (a line of code typed into the REPL).

Transitions The transition, T, takes the set of partial programs pp and applies the action a to either:

1. instantiate a new sub-tree if a is a terminal production: $T(pp, a) = pp \cup \{a\}$
2. combine multiple sub-trees if a is a non-terminal: $T(pp, a) = (pp \cup \{a(t_1 \dots t_k)\}) - \{t_1 \dots t_k\}$

Note that in the case of a non-terminal, the children $t_1 \dots t_k$ are removed, or 'garbage-

collected' [151].

Reward The reward is 1 if there is a program $p \in pp$ that satisfies the spec, and 0 otherwise.

Note that the state of our MDP is defined jointly in the syntactic space, pp, and the semantic space, $spec$. To bridge this gap, we use a REPL, which evaluates the set of partial programs pp into a semantic or "executed" representation. Let pp be a set of n programs, $pp = \{p_1 \dots p_n\}$ and let p denote the execution of a program p, then we can write the REPL state as $pp = \{p_1 \dots p_n\}$.

Training the Code-Writing Policy π and the Code-Assessing Value v

Given the pair of evaluated program states and spec (pp, $spec$), the policy π outputs a distribution over actions, written $\pi(a \mid pp, spec)$, and the value function v predicts the expected reward starting from state (pp, $spec$). In our MDP, expected total reward conveniently coincides with the probability of a rollout satisfying $spec$ starting with the partial program pp:

$$\begin{aligned} v\left(pp, spec\right) &= \mathrm{E}_{\pi}\left[R \mid pp, spec\right] = \mathrm{E}_{\pi}\left[spec \text{ is satisfied} \mid pp, spec\right] \\ &= \mathrm{P}\left[\text{rollout w/ } \pi \text{ satisfies } spec \mid pp, spec\right] \end{aligned} \tag{3.9}$$

Thus the value function simply performs binary classification, predicting the probability that a state will lead to a successful program.

Pretraining π. Because we assume the existence of a CFG and a REPL, we can generate an infinite stream of training data by sampling random programs from the CFG, executing them to obtain a spec, and then recovering the ground-truth action sequence. Specifically, we draw samples from a distribution over synthetic training data, $\mathcal{C}$, consisting of triples of the spec, the sequence of actions, and the set of partially constructed programs

at each step: $(spec, \{a_t\}_{t \leq T}, \{pp_t\}_{t \leq T}) \sim \mathcal{C}$. During pretraining, we maximize the log likelihood of these action sequences under the policy:

$$\mathcal{L}^{\text{pretrain}}(\pi) = \mathrm{E}_{\mathcal{C}} \left[\sum_{t \leq T} \log \pi(a_t | pp_t, spec) \right] \tag{3.10}$$

Training π and v. We fine-tune the policy and train the value function by sampling the policy's roll-outs against $spec \sim \mathcal{C}$ in the style of REINFORCE. Specifically, given $spec \sim \mathcal{C}$, the policy's rollout consists of a sequence of actions $\{a_t\}_{t \leq T}$, a sequence of partial programs $\{pp_t\}_{t \leq T}$, and a reward $R = \mathbb{1}\left[pp_T \text{ satisfies spec}\right]$. Given the specific rollout, we train v and π to maximize:

$$\begin{aligned} \mathcal{L}^{\text{RL}}(v, \pi) = \; & R \sum_{t \leq T} \log v(pp_t, spec) + (1 - R) \sum_{t \leq T} \log(1 - v(pp_t, spec)) \\ & + R \sum_{t \leq T} \log \pi(a_t | pp_t, spec) \end{aligned} \tag{3.11}$$

An SMC Inference Algorithm That Interleaves Writing, Executing, and Assessing Code

At test time we interleave *code writing*, i.e. drawing actions from the policy, and *code assessing*, i.e. querying the value function (and thus also interleave execution, which always occurs before running these networks). Sequential Monte Carlo methods [35], of which particle filters are the most famous example, are a flexible framework for designing samplers that infer a sequence of T latent variables $\{x_t\}_{t \leq T}$ conditioned on a paired sequence of observed variables $\{y_t\}_{t \leq T}$. Following Section 3.1 we construct an SMC sampler for our setting by identifying the policy rollout over time as the sequence of latent variables (i.e. $x_t = pp_t$), and identify the spec as the observed variable at every time step (i.e. $y_t = spec$),

which are connected to the policy and value networks by defining

$$\mathrm{P}(x_{t+1}|x_t) = \sum_{\substack{a \\ T(x_t,a)=x_{t+1}}} \pi(a|x_t) \qquad\qquad \mathrm{P}(y_t|x_t) \propto v(x_t, y_t) \tag{3.12}$$

and, like a particle filter, we approximately sample from $\mathrm{P}(\{pp_t\}_{t\leq T}|spec)$ by maintaining a population of K particles – each particle a state in the MDP – and evolve this population of particles forward in time by sampling from π, importance reweighting by v, and then resampling. Unlike MCMC, Sequential Monte Carlo methods do not suffer from 'burn-in' problems, but may need a large population of particles. We repeatedly run our SMC sampler, doubling the particle count each time, until a timeout is reached.

SMC techniques are not the only reasonable approach: one could perform a beam search, seeking to maximize $\log \pi(\{a_t\}_{t<T}|spec) + \log v(pp_T, spec)$; or, A* search by interpreting $-\log \pi(\{a_t\}_{t<T}|spec)$ as cost-so-far and $-\log v(pp_T, spec)$ as heuristic cost-to-go; or, as popularized by AlphaGo, one could use MCTS jointly with the learned value and policy networks. SMC confers two main benefits: (1) it is a stochastic search procedure, immediately yielding a simple any-time algorithm where we repeatedly run the sampler and keep the best program found so far; and (2) the sampling/resampling steps are easily batched on a GPU, giving high throughput unattainable with serial algorithms like A*.

3.2.3 Experiments

We study a spectrum of models and test-time inference strategies with the primary goal of answering three questions: Is a value function useful to a REPL-guided program synthesizer; how, in practice, should the value function be used at test time; and how, in practice, does our full model compare to prior approaches to execution-guided synthesis. We answer these questions by measuring, for each model and inference strategy, how efficiently it can search

the space of programs, i.e. the best program found as a function of time spent searching.

We test the value function's importance by comparing SMC against policy rollouts[2], and also by comparing a beam search using π & v against a beam decoding under just π; orthogonally, to test how the value function should be best used at test time, we contrast SMC (w/ value) against beam search (w/ value) and also A* (w/ value as the heuristic function). Finally we contrast our best approach—SMC—with recent execution-guided approaches [21, 151], which both perform a beam-search decoding under a learned policy. As a sanity check, we additionally train a 'no REPL' baseline, which decodes an entire program in one shot using only the spec and intermediate program syntax. This tests whether the REPL helps or hinders. These 'no REPL' architecture roughly mirrors the prior work CSGNet [118].

Modern mechanical parts are created using Computer Aided Design (CAD), a family of programmatic shape-modeling techniques. Here we consider two varieties of *inverse* CAD: inferring programs generating 3D shapes, and programs generating 2D graphics, which can also be seen as a kind of high-level physical object understanding in terms of parts and relations between parts. We use CSG as our CAD modeling language, where the specification $spec$ is an image, i.e. a pixel/voxel array for 2D/3D, and the the goal is to write a program that renders to the target image. These programs build shapes by algebraically combining primitive drawing commands via addition and subtraction, including circles and rotated rectangles (for 2D, w/ coordinates quantized to 16×16) as well as spheres, cubes, and cylinders (for 3D, w/ coordinates quantized to $8 \times 8 \times 8$), although in both cases the quantization could in principle be made arbitrarily fine. Our REPL renders each partial program $p \in pp$ to a distinct canvas, which the policy and value networks take as input.

The space of allowed programs are generated by the CFG below:

[2]SMC w/o a value function would draw samples from π, thus the relevant comparison for measuring the value of the value function is SMC w/ value against policy rollouts.

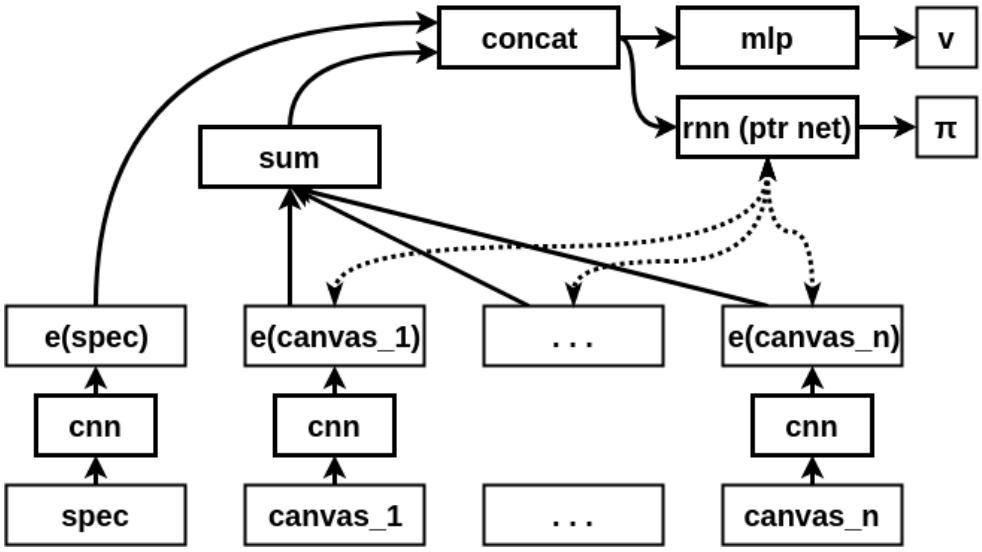


Figure 3-17: Inverse CAD neural network architecture. The policy is a collection of CNNs w/ shared weights (notated $e(\cdot)$), pooled together and then followed by a pointer network [141] (attending over the set of partial programs pp) which decodes into the next line of code. The value function is an additional 'head' to the pooled CNN activations. Appendix B.2.1 provides further details of the network architecture.

```
P -> P + P | P - P | S

# For 2D graphics
S -> circle(radius=N, x=N, y=N)
   | quadrilateral(x0=N, y0=N,
                   x1=N, y1=N,
                   x2=N, y2=N,
                   x3=N, y3=N)

N -> [0 : 31 : 2]

# For 3D graphics
S -> sphere(radius=N, x=N, y=N, z=N)
   | cube(x0=N, y0=N, z0=N,
          x1=N, y1=N, z1=N)
   | cylinder(x0=N, y0=N, z0=N,
              x1=N, y1=N, z1=N, radius=N)
N -> [0 : 31 : 4]
```

In principle the 2D language admits arbitrary quadrilaterals. When generating synthetic training data we constrain the quadrilaterals to be take the form of rectangles rotated by 45 increments, although in principle one could permit arbitrary rotations by simply training a higher capacity network on more examples.

Experimental evaluation We train our models on randomly generated scenes with up to 13 objects. As a hard test of *out-of-sample generalization* we test on randomly generated scenes with up to 30 or 20 objects for 2D and 3D, respectively. Appendix B.2.2 further details neural network training. Figure 3-18 measures the quality of the best program found so far as a function of time, where we measure the quality of a program by the intersection-over-union (IoU) with the spec. Incorporating the value function proves important for both beam search and sampling methods such as SMC. Given a large enough time budget the 'no REPL' baseline is competitive with our ablated alternatives: inference time is dominated by CNN evaluations, which occur at every step with a REPL, but only once without

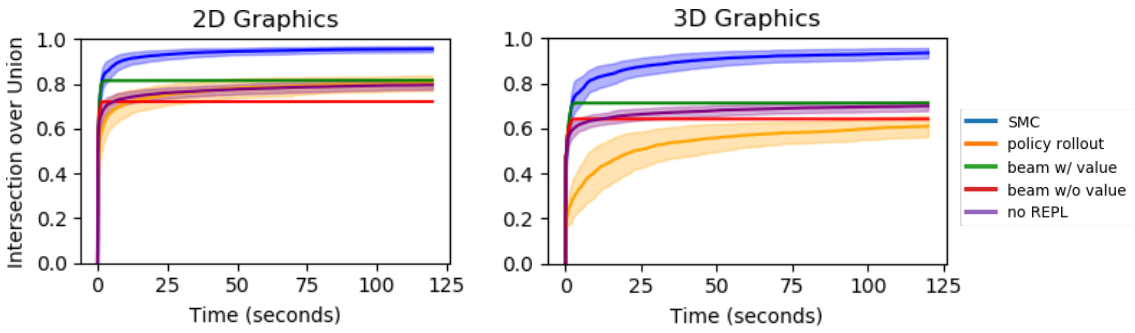


Figure 3-18: Quantitative results for CAD on out-of-sample testing problems. Both models trained on scenes with up to 13 objects. Left: 2D models tested on scenes with up to 30 objects. Right: 3D models tested on scenes with up to 20 objects. A* unapplicable due to extremely large action space (here, branching factor > 1.3 million). Error bars: average stddev for sampling-based inference procedures over 5 random seeds. SMC achieves the highest test accuracy.

it. Qualitatively, an integrated policy, value network, and REPL yield programs closely matching the spec (Figure 3-19). Together these components allow us to infer very long programs, despite a branching factor of ≈1.3 million per line of code: the largest programs we successfully infer[3] go up to 21 lines of code/104 tokens for 3D and 29 lines/158 tokens for 2D, but the best-performing ablations fail to scale beyond 11 lines/59 tokens for 3D and 19 lines/117 tokens for 2D.

3.3 Lessons

We suggest taking the following general lessons from this chapter:

Programs are compatible with perception. It is practical to synthesize programs from perceptual data. Conceptually, the Bayesian framework provides a kind of " mathematical glue" for writing down objective functions that bridge high-dimensional perception and symbolic programs. Neural networks are naturally embedded in this Bayesian framework,

[3]By "successfully infer" we mean IoU≥0.9

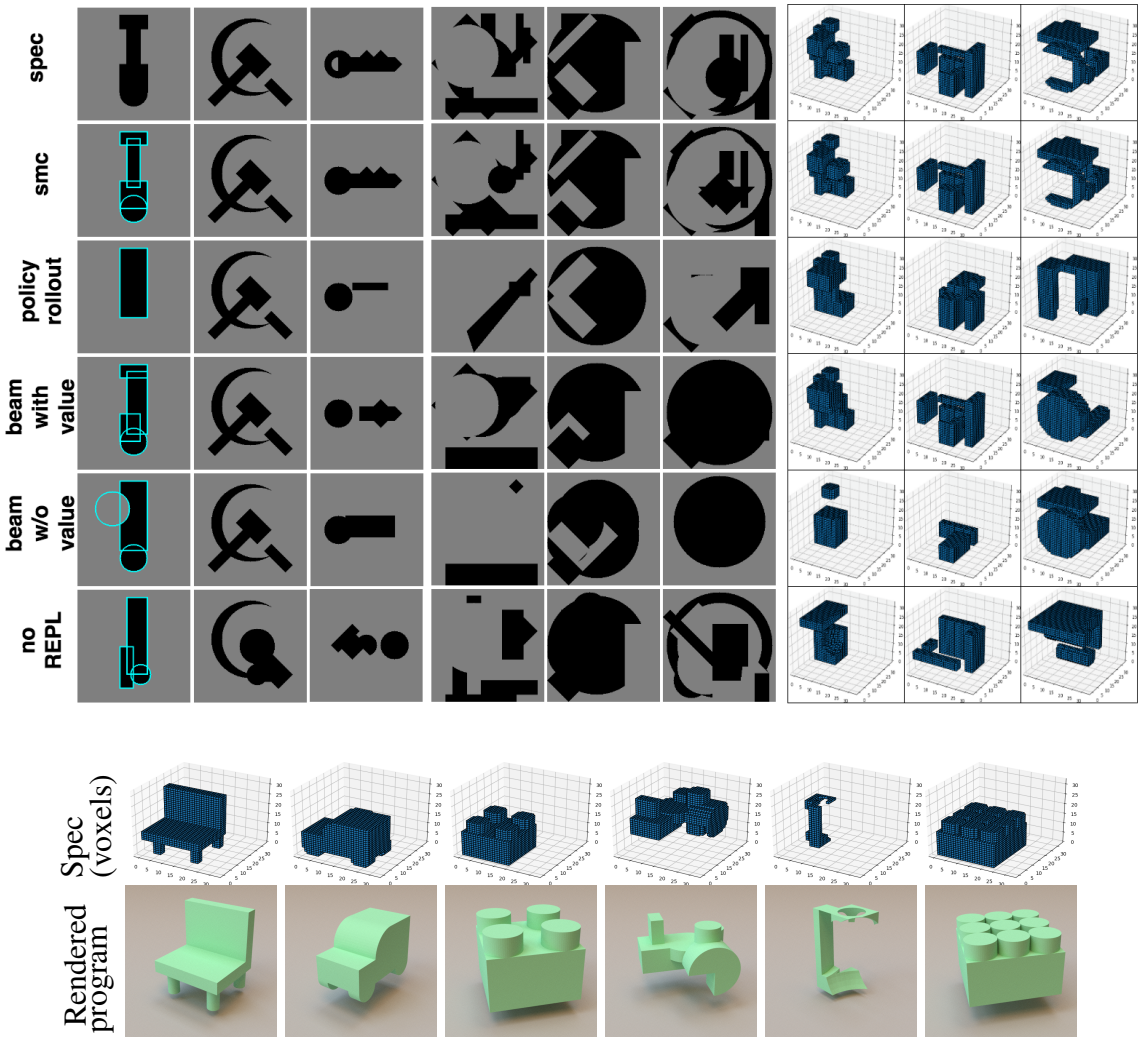


Figure 3-19: Qualitative inverse CAD results. Top: Derendering random scenes vs. ablations and no-REPL baseline. Teal outlines show shape primitives in synthesized 2D programs. Bottom: Rerendering program inferred from voxels from novel viewpoints.

and can act as a perceptual front end for program synthesizers. An outstanding and unresolved question, however, is whether an intermediate symbolic representation–the "trace set"–is needed when synthesizing programs with control flow. Chapter 5 will present early evidence that this intermediate representation is not necessary, and that one may learn neural networks that regress directly from pixel input to high-level programs.

Programs facilitate strong generalization. By having a strong language bias in the form of higher order constructs such as loops, we can get strong kinds of generalization, such as extrapolation. Within perception in particular, we can get an interaction between bottom-up cues and this top-down bias which leads to more accurate reconstructions from images. Chapter 5 will draw out this aspect of programs more strongly when we learn much of this language bias.

Error correction methods combat exponentially decaying probabilities of success. Writing a long piece of code requires making a long sequence of decisions, and if even a single one of those syntactic decisions is incorrect, the program could fail catastrophically. Even if our success rate for single decision is very high (i.e. $(1-\epsilon)$) the probability of having no failures decays exponentially (i.e. $(1-(1-\epsilon)^L)$ for a program of length L). Thus we require some means of error correction, such as the learned distance metric in Section 3.1 or the value function in Section 3.2.

Draw insights from human coders. Human programmers use many sophisticated tools to write software, including interpreters, version control, libraries, debuggers, Stack overflow, etc. The work here has explored what it would mean to equip a program synthesizer with an interpreter, and the work in Chapter 5 will explore what it would mean to equip the program synthesizer with the ability to build a library. Many avenues remain open: for example, we could augment synthesizers with a "debugger" that allowed conditioning on intermediate stack states, variable bindings during loops, etc.

Acknowledgments

This chapter contains material taken from “Learning to infer graphics programs from hand-drawn images” (NeurIPS 2018, by Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, Joshua Tenenbaum) and “Write, Execute, Assess: Program synthesis with a REPL” (NeurIPS 2019, by Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Joshua Tenenbaum, Armando Solar-Lezama; equal contribution by first four authors). The dissertation author was a primary author on these papers.

Chapter 4

Discovering models and theories: A case study in computational linguistics

A key aspect of human intelligence is our ability to build theories about the world. This faculty is most clearly manifested in the historical development of science, but also occurs in miniature in everyday cognition [50], and during childhood development [18]. The similarities between the process of developing scientific theories and the way that children construct an understanding of the world around them has led to the *child as scientist* metaphor in developmental psychology, which views conceptual changes during development as a form of scientific theory discovery [20, 75, 117, 76, 100]. Thus, a key goal for both artificial intelligence and computational cognitive science is to develop methods to understand—and perhaps even automate—the process of theory discovery.

In this chapter, we study the problem of theory discovery in the domain of human language. We primarily focus on the linguist's construction of language-specific theories, and the linguist's synthesis of abstract cross-language meta-theories, but argue that our accounts can shed light on the child's acquisition of language. The cognitive sciences of

language have long drawn an explicit analogy between the working scientist constructing grammars of particular languages and the child learning their languages [27, 25]. Language specific grammars must be formulated within a common theoretical framework, sometimes called *universal grammar*. For the linguist, this is the target of empirical inquiry, for the child, this constitutes the sum total of resources that they bring to the table for language acquisition.

Natural language is an ideal domain to study theory discovery for several reasons. First, on a practical level, decades of work in linguistics, psycholinguistics, and other cognitive sciences of language provide diverse raw material to develop and test models of automated theory discovery. There exist corpora, datasets, and grammars from a large variety of typologically distinct languages, giving a rich and varied testbed for benchmarking theory induction algorithms. Second, children easily acquire language from quantities of data that are modest by the standards of modern artificial intelligence. Similarly, working field linguists also develop grammars based on very small amounts of elicited data. These facts suggest that the child-as-linguist analogy is a productive one and that inducing theories of language is tractable from sparse data with the right inductive biases. Third, theories of language representation and learning are formulated in computational terms, exposing a suite of formalisms ready to be deployed by AI researchers. These three features of human language — the availability of a large number of highly diverse empirical targets, the interfaces with cognitive development, and the computational formalisms within linguistics — conspire to single out language as an especially suitable target for research in automated theory induction.

Ultimately, the goal of the language sciences is to understand the general representations, processes, and mechanisms which allow people to learn and use language, not merely to catalog and describe particular languages. We use program induction to capture this *framework-level* aspect of the problem of theory synthesis. We embed classic linguistic

formalisms within a programming language provided to a program learner, and with this inductive bias, the model can then learn programs capturing a wide diversity of natural language phenomena. By systematically ablating and manipulating the system, and therefore varying the inductive bias, we can systematically study elements of the induction problem that span multiple languages. By doing hierarchical Bayesian inference on the programming language itself, we can also automatically discover some of these universal trends. But program induction comes at a steep computational cost, and so we develop new algorithms which combine techniques from program synthesis with intuitions drawn from how scientists build theories and how children learn languages. We restrict our focus to theories of natural language *morpho-phonology*—the domain of language which concerns the interaction of word formation and sound structure. Acquiring the morpho-phonology of a language involve solving a basic problem confronting both linguists and children: to build theories of the causal relationship between form and meaning given a collections of utterances, together with aspects of their meanings.

We evaluate our algorithm on 70 data sets spanning 58 languages, automatically finding interpretable causal theories that can model a wide swath of a core component of human language. We will then shift our focus from linguists to children, and show that the same approach for natural language can also capture classic findings in the infant artificial grammar learning literature. Finally, by performing hierarchical Bayesian inference across these linguistic data sets, we show that the model can distill out universal cross-language patterns, and express those patterns in a compact, human understandable form. Collectively these findings point the way toward more human-like AI systems for learning theories, and for systems that learn-to-learn those theories more effectively over time by refining their inductive biases.

From an algorithmic perspective, our work here draws on principles introduced earlier in this thesis, while also foreshadowing further techniques. Like in Chapter 3, we will

synthesize programs in a piecemeal fashion rather than tackle the whole problem at once; additionally we will learn an inductive bias over programs (i.e. grammars). However, the nature of this inductive bias will be more sophisticated than the log-linear model of Section 3.1. Here we will induce an inventory of symbolic schemas, which function both as a human-understandable representation of cross linguistic knowledge, as well as a useful inductive bias over the space of grammars. This symbolic encoding of inductive biases will be further elaborated in Chapter 5.

4.1 Discovering Theories by Synthesizing Programs

The core problem of natural language learning is to acquire a grammar which maps back and forth between *form* (perception, articulation, etc.) and *meaning* (concepts, intentions, thoughts, etc.). We realize this by thinking of a grammar as generating a distribution over *expressions*. Each *expression* is a form-meaning tuple, $\langle f, m \rangle$, where each *form* corresponds to sound structure (sequences of phonemes described as phonetic feature vectors), and each *meaning* is a set of atomic meaning features. For example, in English, the word *opened* has the expression $\langle$/opɛnd/, [**root**:OPEN;**tense**:PAST]$\rangle$, which the grammar builds from the expression for *open*, namely $\langle$/opɛn/, [**root**:OPEN]$\rangle$, and the expression for the past tense, namely $\langle$/d/, [**tense**:PAST]$\rangle$.

Grammars can generate infinitely many form-meaning pairs depending on what words are in the language (e.g., see the "Jabberwocky" poem), just as theories in other sciences can generate an infinite set of possible observations — in Newtonian mechanics, the theory prescribes how bodies will interact, but does not prescribe the number of bodies or their masses; analogously, grammars prescribe systems of rules, but allow new words to enter the language provided they interact according to this rule system. Thus, for a theory to explain a particular dataset it must introduce additional latent (unobserved) variables on a

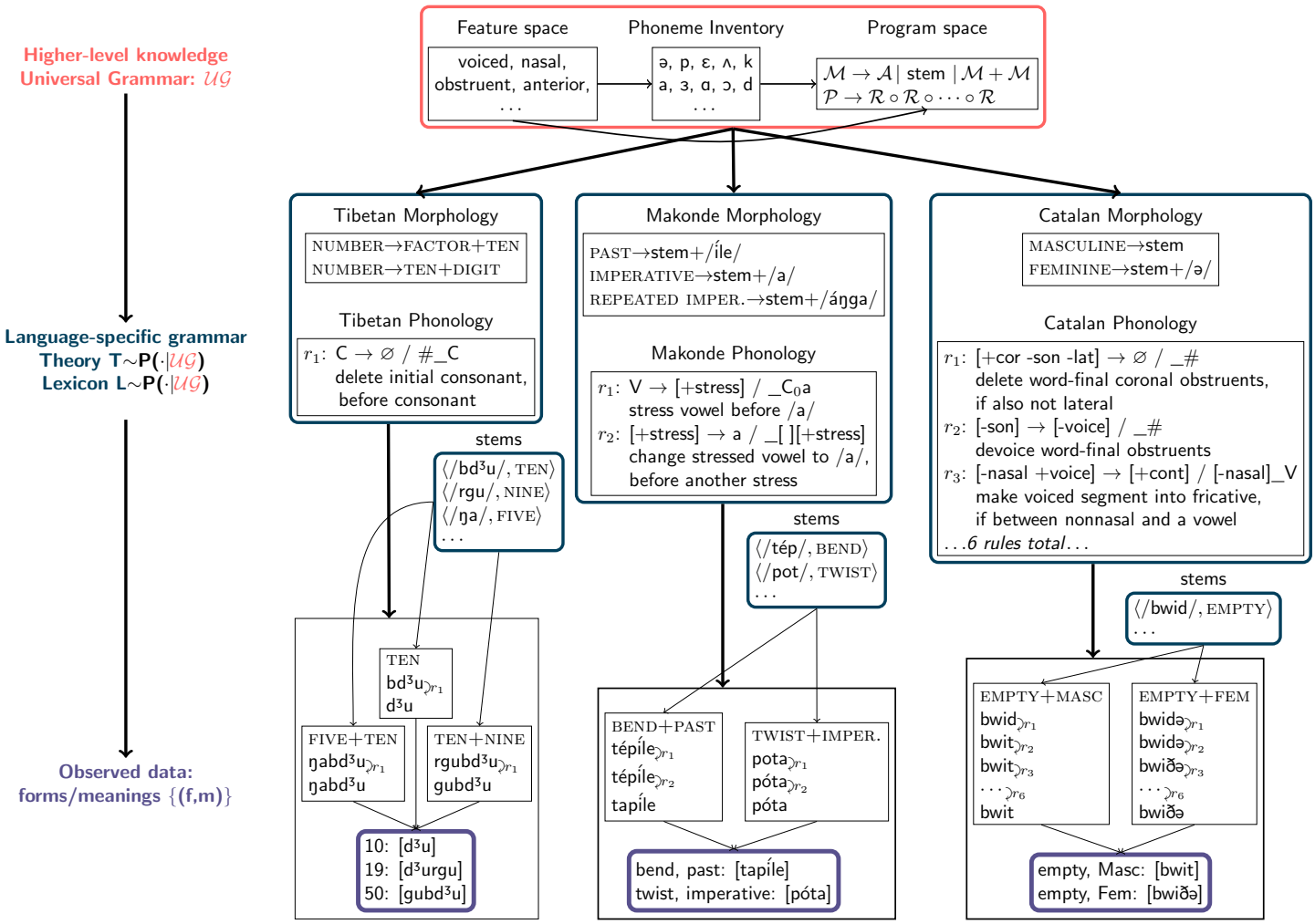


Figure 4-1: The generative model underlying our approach. We infer grammars for a range of languages, given only form/meaning pairs (purple) and a space of programs (pink). Grammars include morphology, which concatenates different suffixes/prefixes for each inflection to the language's stems, and phonology, which transforms the output of morphology into the observed surface forms (written in brackets), using a sequence of ordered rules, labeled r_1, r_2, etc. Each rule is written as a context-sensitive rewrite, and beneath it, an English description. In the lower black boxes we show the inferred derivation of the observed data, i.e. the execution trace of the synthesized program. Grammars are expressed as programs drawn from a universal grammar, or space of allowed programs.

dataset-by-dataset basis. For the theories (grammars) considered here, this latent variable is the *lexicon*, written $\mathbb{L}$, which is a set of *lexical entries*. We assume each lexical entry maps each meaning to a unique sound structure and is annotated with its morphosyntactic category: here, either a stem, prefix (`pfx`), or suffix (`sfx`). For example, in English, $\langle$/opɛn/, `stem`, [**root**:OPEN]$\rangle \in \mathbb{L}$ and $\langle$/d/, `sfx`, [**tense**:PAST]$\rangle \in \mathbb{L}$, but neither the form /opɛnd/ nor its meaning [**root**:OPEN;**tense**:PAST] need be in $\mathbb{L}$: instead, the rules of English interact with the lexicon to derive this form-meaning pair.

The system we have described thus far is known as *morphology*, which builds new lexical entries by combining elements of the lexicon. Morphology explains why the past tense for "open" is /opɛnd/, namely because the lexicon records the stem as /opɛn/, records the past tense affix as /d/, and dictates that the affix should be appended. But morphology alone can not explain the pronunciation of the past tense form "walked," which is [wɔk**t**] rather than [wɔk**d**]. Nor can morphology explain other aspects of word production, such as stress patterns (e.g., "monotone" stresses the second vowel while "monotonic" stresses the penultimate vowel). *Phonology* explains phenomena like these. A language's phonology is a function that transduces the output of morphology into an utterance's actual pronunciation. The reason why "walked" is pronounced [wɔk**t**] while "opened" is [opɛn**d**] is because English phonology converts word-final /d/ into /t/ when preceded by /k/ but not /n/. Phonological output is referred to as an utterance's *surface form*, conventionally written in brackets (e.g., [wɔk**t**]). Intermediate forms, such as those output by morphology, are referred to as *underlying forms*, conventionally written between slashes (e.g., /wɔk**d**/). Composed together, *morpho-phonology* explains why word pronunciation varies systematically across different inflections, and allows the speaker of a language to hear just a single example of a new word and immediately generate and comprehend all its inflected forms.

We take as a goal to inductively synthesize a language's morpho-phonology. Our model

works by explaining a set $\mathbf{X}$ of form-meaning pairs $\langle f, m\rangle$ by inferring a theory (grammar) $\mathbb{T}$, alongside the latent lexicon. Viewed as probabilistic inference, we seek to maximize $P(\mathbb{T}, \mathbb{L}|\mathcal{UG}) \prod_{\langle f,m\rangle \in \mathbf{X}} P(f, m|\mathbb{T}, \mathbb{L})$, where $\mathcal{UG}$ (for universal grammar) encapsulates higher-level abstract knowledge across different languages. We decompose each language-specific theory into separate modules for morphology and for phonology (Fig. 4-1). We restrict ourselves to concatenative morphology, which builds new words by concatenating prefixes and suffixes. We model phonology as K ordered rules, written $\{r_k\}_{k=1}^{K}$, each of which is a function mapping sequences of phonemes to sequences of phonemes. We refine the theory-induction objective into finding the phonological rules and lexicon maximizing

$$P(\mathbb{T}, \mathbb{L}|\mathcal{UG}) \prod_{\langle f,\ [\textbf{root}{:}\rho;\ i]\rangle \in \mathbf{X}} \mathbb{1}\Big[\exists\langle p, \texttt{pfx}, i\rangle \in \mathbb{L},\quad \exists\langle s, \texttt{sfx}, i\rangle \in \mathbb{L},\quad \exists\langle r, \texttt{stem}, [\textbf{root}{:}\rho]\rangle \in \mathbb{L} : f = r_1(r_2(\cdots r_K(p \cdot r \cdot s)\cdots))\Big] \tag{4.1}$$

where $\langle f,\ [\textbf{root}{:}\rho;\ i]\rangle$ is a form-meaning pair with form f, root ρ, and i are the remaining aspects of meaning that exclude the root (e.g., i could be [**tense**:PAST; **gender**:NEUT]).

To represent the space of rules we adopt the classical formulation in terms of context-sensitive rewrites, as in [26].[1] Rules are written (focus) $\rightarrow$ (structural change) / (left trigger)_(right trigger), meaning that the *focus* phoneme(s) are transformed according to the *structural change* whenever the left/right triggering environments occur immediately to the left/right of the focus (see Figure 4-2, and Figure 4-3 for prior over grammars). For example, in English, phonemes which are [-sonorant] (such as /d/) become [-voice] (e.g., /d/ becomes /t/) at the end of a word (written #) whenever the phoneme to the left is also an unvoiced nonsonorant ([-voice -sonorant], such as /k/), written [-sonorant] $\rightarrow$ [-voice] / [-voice -sonorant]_#. This specific rule transforms the past tense *walked* from its

[1] These are sometimes called *SPE-style rules* since they were used to model phonology extensively in the *Sound Pattern of English* [26].

underlying form /wɔkd/ into its surface form [wɔkt]. When such rules are restricted to not be able to cyclically apply to their own output, they have the formal power of finite-state transducers [67]. It has been argued that the space of finite-state transductions has sufficient representational power to cover known empirical phenomenon in morpho-phonology and represents a limit on the descriptive power actually used by phonological theories–including Optimality Theory–even those that are formally more powerful [58].

To learn such grammars we model each $\mathbb{T}$ as a program in a programming language which captures domain specific constraints on the problem space. The architecture of the linguistic system common to all languages is often referred to as *universal grammar*. Our approach can be seen as a modern instantiation of a long-standing approach in linguistics which adopts causal, procedural, human-understandable formalisms to formalize universal grammar [26].

4.1.1 A new program synthesis algorithm for incrementally building generative theories

We have defined the problem a theory inductor needs to solve, but have not given any guidance on how to solve it. In particular, the space of all programs (theories) is infinitely large and lacks the local smoothness exploited by local optimization algorithms like gradient descent or MCMC. We adopt a strategy based on constraint-based program synthesis, where the optimization problem is translated into a combinatorial constraint satisfaction problem and solved using a Satisfability Modulo Theories (SMT) solver [12]. These solvers implement an exhaustive but efficient search and guarantee that, given enough time, an exact and optimal solution will be found. We use the Sketch [125] program synthesizer, which we can treat as an oracle capable of solving the following constrained optimization

	Template for a single rule	
Rule ::=	Focus→Change / Trigger _ Trigger	
	Focus of rule, and what it can change into	
Focus ::=	$\varnothing$	*insertion rule*
Focus ::=	FeatureMatrix	
Change ::=	$\varnothing$	*deletion rule*
Change ::=	$\mathbb{Z}$	*copying rule: an integer*
Change ::=	αplace	*copy place from α, an integer*
Change ::=	FeatureMatrix	
	Every triggering environment with at most two feature matrices	
Trigger::=		*empty conditioning (triggering) environment*
Trigger::=	FeatureMatrix	*a single feature matrix to the side of focus*
Trigger::=	FeatureMatrix_0 FeatureMatrix	*zero or more repeats of a feature matrix*
Trigger::=	FeatureMatrix FeatureMatrix	
Trigger::=	#	*end-of-string*
Trigger::=	FeatureMatrix #	
Trigger::=	FeatureMatrix_0 FeatureMatrix #	
Trigger::=	FeatureMatrix FeatureMatrix #	
Trigger::=	{#,FeatureMatrix}	*end-of-string or a feature matrix*
Trigger::=	FeatureMatrix_0 {#,FeatureMatrix}	
Trigger::=	FeatureMatrix {#,FeatureMatrix}	
	Build feature matrices from constant phoneme or sequence of ±feature	
FeatureMatrix ::=	Phoneme	
FeatureMatrix ::=	[ ]	*empty feature matrix*
FeatureMatrix ::=	[+ Feature FeatureMatrix]	*+feature appended to matrix*
FeatureMatrix ::=	[- Feature FeatureMatrix]	*−feature appended to matrix*
Phoneme ::=	ə | a | g | ⋯	*a constant phoneme*
Feature ::=	voice | nasal | coronal | ⋯	*a phonological feature*
	integer indices are used for copying, see caption	
$\mathbb{Z}$ ::=	$-2 \mid -1 \mid 1 \mid 2$	*copying target, an integer*
α ::=	$-2 \mid -1 \mid 1 \mid 2$	*place copy target, an integer*

Figure 4-2: Context-free grammar generating phonological rules used by our system. Non-terminal symbols begin with a capital letter, as well as $\mathbb{Z}$ and α. For increased tractability, we arbitrarily bound the size of each FeatureMatrix to have had most three features, and as outlined in the above grammar over SPE rules, each trigger may have at most two feature matrices, hence the maximum range of copying targets ($\mathbb{Z}$/α) of ± 2. Copying targets are expressed as integers indexing into the triggering environments. For example the rule V $\rightarrow$ V_i / V_i_CC is expressed using the above grammar as V $\rightarrow$ -1 / V_CC, while the rule V $\rightarrow$ V_i / C_C_0V_i would be expressed as V $\rightarrow$ 2 / C_C_0V.

$$
\begin{aligned}
P(f) &\propto \exp\left(-|f|\right),\ f \text{ a form in the lexicon} \\
P(r) &\propto \exp\left(-\text{cost}(r)\right),\ r \text{ a rule} \\
\text{cost}(\text{focus} \rightarrow \text{change} \ /\ \text{left\_right}) &= \text{cost}(\text{focus}) + \text{cost}(\text{change}) + \text{cost}(\text{left}) + \text{cost}(\text{right}) \\
\text{cost}(\varnothing) &= 2 \\
\text{cost}(k) &= 2,\ k \text{ a constant phoneme} \\
\text{cost}([\pm\text{feature}_1 \ \cdots \pm \text{feature}_n]) &= 1 + n \\
\text{cost}(\#) &= 1 \\
\text{cost}(\{\#, m\}) &= 1 + \text{cost}(\#) + \text{cost}(m) \\
\text{cost}(m_0) &= 1 + \text{cost}(m) \\
\text{cost}(uv) &= \text{cost}(u) + \text{cost}(u), \text{ when } u,v \text{ elements of a trigger}
\end{aligned}
$$

Figure 4-3: Prior over grammars, including both theories and lexica. As a baseline $\mathcal{UG}$, we simply count the number of symbols present in the lexicon and rules; we heuristically penalize insertions, deletions, and constant phonemes by counting them as two symbols.

problem, which is equivalent to our goal of maximizing $P(\mathbf{X}|\mathbb{T},\mathbb{L})P(\mathbb{T},\mathbb{L}|\mathcal{UG})$:

$$\begin{aligned}
&\text{maximize } F(\mathbf{X},\mathbb{T}) = \sum_{k=1}^{K} \log P(r_k|\mathcal{UG}) + \sum_{\langle f,c,m\rangle\in\mathbb{L}} \log P(f|\mathcal{UG}) \\
&\text{subject to } C(\mathbf{X},\mathbb{T}) = \forall\langle f,\ [\textbf{root}{:}\rho;\, i]\rangle \in \mathbf{X}: \\
&\qquad\qquad \exists\langle p,\texttt{pfx},i\rangle\in\mathbb{L},\quad \exists\langle s,\texttt{sfx},i\rangle\in\mathbb{L},\quad \exists\langle r,\texttt{stem},[\textbf{root}:\rho]\rangle\in\mathbb{L}: \\
&\qquad\qquad f = r_1(r_2(\cdots r_K(\mathbb{L}(\langle i,\texttt{pfx}\rangle)\cdot\mathbb{L}(\langle\rho,\texttt{stem}\rangle)\cdot\mathbb{L}(\langle i,\texttt{sfx}\rangle)\cdots))
\end{aligned} \tag{4.2}$$

given observations $\mathbf{X}$ and bound on number of rules K

In practice, the clever exhaustive search techniques employed by SMT solvers fail to scale to the many rules needed to explain large corpora. To scale these solvers to large and complex theories, we take inspiration from a basic feature of how children acquire language and how scientists build theories. Children don't learn language in one fell swoop, instead progressing through intermediate stages of linguistic development, gradually enriching their mastery of both grammar and lexicon. Similarly, a sophisticated scientific theory might start with a simple conceptual kernel, and then gradually grow to encompass more and more phenomena. Motivated by these observations, we engineered a program synthesis algorithm that starts with a small program, and then repeatedly uses an SMT solver to search for small modifications that allow it to explain more and more data. Concretely, we find a counterexample to our current theory, and then use the solver to exhaustively explore the space of all small modifications to the theory which can accommodate this counterexample, combining the idea of counter-example guided synthesis [124] with test-driven synthesis [101]. Figure 4-4 illustrates this incremental, solver-aided synthesis algorithm.

Mathematically, we iteratively construct a sequence of theories $\mathbb{T}_0$, $\mathbb{T}_1$, ... alongside

successively larger data sets $\mathbf{X}_0$, $\mathbf{X}_1$, ... converging to the full data set $\mathbf{X}$, such that the t^{th} theory $\mathbb{T}_t$ explains data set $\mathbf{X}_t$, and successive theories are close to one another as measured by edit distance:

$$\mathbf{X}_{t+1} = \mathbf{X}_t \cup (\text{a set } \mathbf{X}' \subseteq \mathbf{X} \text{ where } C(\mathbb{T}_t, \mathbf{X}') \text{ does not hold}) \tag{4.3}$$

$$D_{t+1} = \min_D D, \text{ such that there exists } \mathbb{T} \text{ where } d(\mathbb{T}, \mathbb{T}_t) \leq D \text{ and } C(\mathbb{T}, \mathbf{X}_{t+1}) \text{ holds} \tag{4.4}$$

$$\mathbb{T}_{t+1} = \arg\max_{\mathbb{T}} F(\mathbf{X}, \mathbb{T}), \text{ such that } d(\mathbb{T}, \mathbb{T}_t) \leq D_t \text{ and } C(\mathbb{T}, \mathbf{X}_{t+1}) \text{ holds} \tag{4.5}$$

where $d(\cdot, \cdot)$ measures edit distance between theories, D_{t+1} is the edit distance between the theory at iteration $t+1$ and t, and we use the $t = 0$ base cases $\mathbf{X}_0 = \varnothing$ and $\mathbb{T}_0$ is an empty theory containing no phonological rules. We define the edit distance, $d(\mathbb{T}_1, \mathbb{T}_2)$, between a pair of theories $\mathbb{T}_1$, $\mathbb{T}_2$ by counting the number of insertions, deletions, substitutions, and swaps that separate the sequences of rules for $\mathbb{T}_1$ and $\mathbb{T}_2$. We 'minibatch' counterexamples to the current theory ($\mathbf{X}'$ in Eq. 4.3) grouped by lexeme, and ordered by their occurrence in the textbook problem (e.g., if the theory fails to explain walk/walks/walked, and this is the next example in the problem, then the surface forms of walk/walks/walked will be added to $\mathbf{X}_{t+1}$). This leverages the pedagogical example ordering selected by the textbook author for each problem. We automatically set the number of lexemes in each batch such that the surface forms in a minibatch will be less than ten (ie, with 3 inflections, each minibatch will comprise 3 lexemes; with 5 inflections, each minibatch will comprise 1 lexeme).

A naive implementation of this approach would encode the edit-distance constraint directly into the Sketch system. A more efficient, parallelizable approach is to enumerate a finite set of *theory templates*, where each template corresponds to a family of edits to the original theory. A *theory template* is a list, where each list element is either a rule or a

rule-valued hole that Sketch will solve for. Given a sequence of rules (the current theory) and a maximum edit distance, we automatically construct a set of theory templates such that any new theory within the maximum edit distance is an instantiation of one of the theory templates, i.e. there exists a substitution of holes to rules such that the new theory is equal to the theory template with the substitution applied. Given the set of theory templates, we can then invoke Sketch in parallel across multiple CPUs to exhaustively check if any template yields an improved theory accommodating the counterexamples. We used 40 CPUs in our experiments.

The reason why these successive optimization problems are more tractable than direct optimization of F subject to C over the entirety of $\mathbf{X}$ is because we are constraining each new theory to be close in theory-space to its preceding theory, leading to polynomially smaller constraint satisfaction problems and therefore exponentially faster search times, because SMT solvers scale, in the worst case, exponentially with problem size.

4.2 Experimental results

We apply our model to 70 problems from linguistics textbooks [97, 111, 57]. Each textbook problem requires synthesizing a causal theory of a number of forms drawn from some natural language. These problem span a wide range of difficulties and cover a diverse set of natural language phenomena. This includes tonal languages, for example, in Kerewe, 'to count' is [kubala], but 'to count it' is [kukíbála], where accents mark high tones; languages with vowel harmony, for example Turkish has [el]/[tʃan] meaning hand/bell, respectively, and [eller]/[tʃanlar] for the plurals hands/bells, respectively; and many other linguistic phenomena such as assimilation and epenthesis (Fig. 4-5, Fig. 4-6,4-7,4-8).

Figure 4-9 shows the results of applying our model to 70 textbook problems. Compared to ground-truth lexica, our model finds grammars correctly explaining the entirety of the

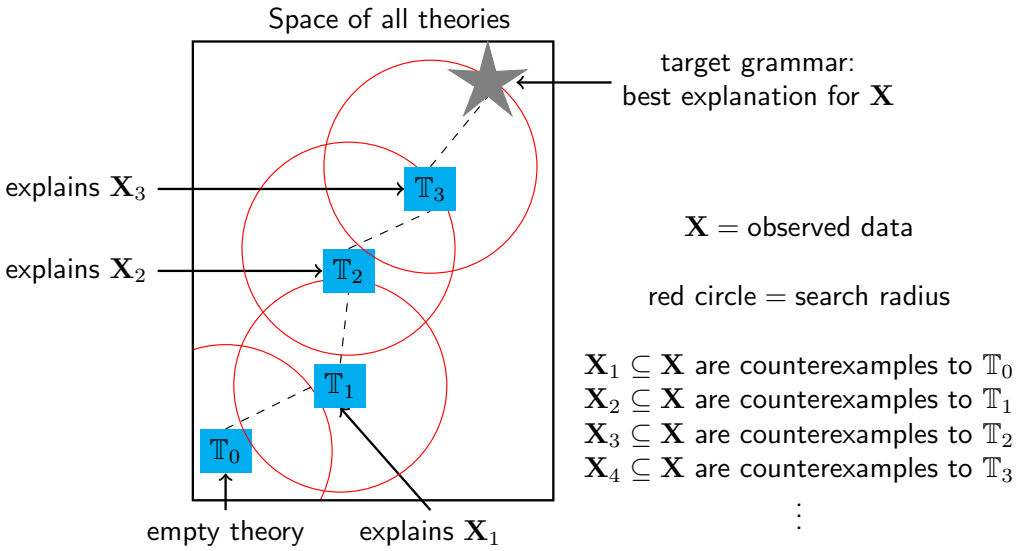


Figure 4-4: In order to scale to large programs explaining large corpora, we repeatedly search for small modifications to our current theory. Each small modification is driven by a counterexample to the current theory. The next counterexamples are taken from the problem in the order presented in the corresponding textbook entry, leveraging the pedagogical ordering given by the text.

problem for 60% of the benchmarks, and correctly explains the majority of the data for 79% of the problems. (We quantify correctness by agreement with ground truth lexica rather than rules because many possible rules may explain the data using the same lexicon.) This is by far the largest variety of morphophonological problems ever correctly solved by a single computational model. Prior approaches either recover interpretable causal models (e.g., [1, 52, 108]) but do not scale to a wide range of challenging and realistic data sets, or abandon theory induction and instead learn opaque probabilistic models [28] that may nonetheless predict the data well but which do not provide explanatory scientific theories.

The model's performance hinges on several factors. A primary key ingredient is a correct set of constraints on the space of hypotheses, i.e. a universal grammar, which we can systematically vary: ablating the model by switching from phonological articulatory features to simpler acoustic features degrades performance ('simple features' in Fig. 4-9); further ablating the essential sources of representational power, phonetic features and Kleene star, renders only the simplest problems solvable ('representation' in Fig. 4-9). Basic algorithmic details also matter: attempting to avoid the joint optimization of the grammar and lexicon by heuristically proposing plausible lexica, as done in concurrent work [11], fails to solve 76% of the problems ('PhonoSynth' in Fig. 4-9); building a large theory at once is intractable for human learners, and also for our model ('CEGIS' in Fig. 4-9). For the first time, these results show that it is possible and practical for a machine system to synthesize many of the formal theories linguists have built to explain structure in natural language.

If our model captures aspects of linguistic analysis from naturalistic data, and assuming linguists and children confront similar problems, then our approach should extend to model at least some aspects of the child's linguistic generalization. Studying children (and adult's) learning of carefully constructed artificial grammars has a long tradition in psycholinguistics and language acquisition [48, 9, 85], because it permits controlled and careful study of the

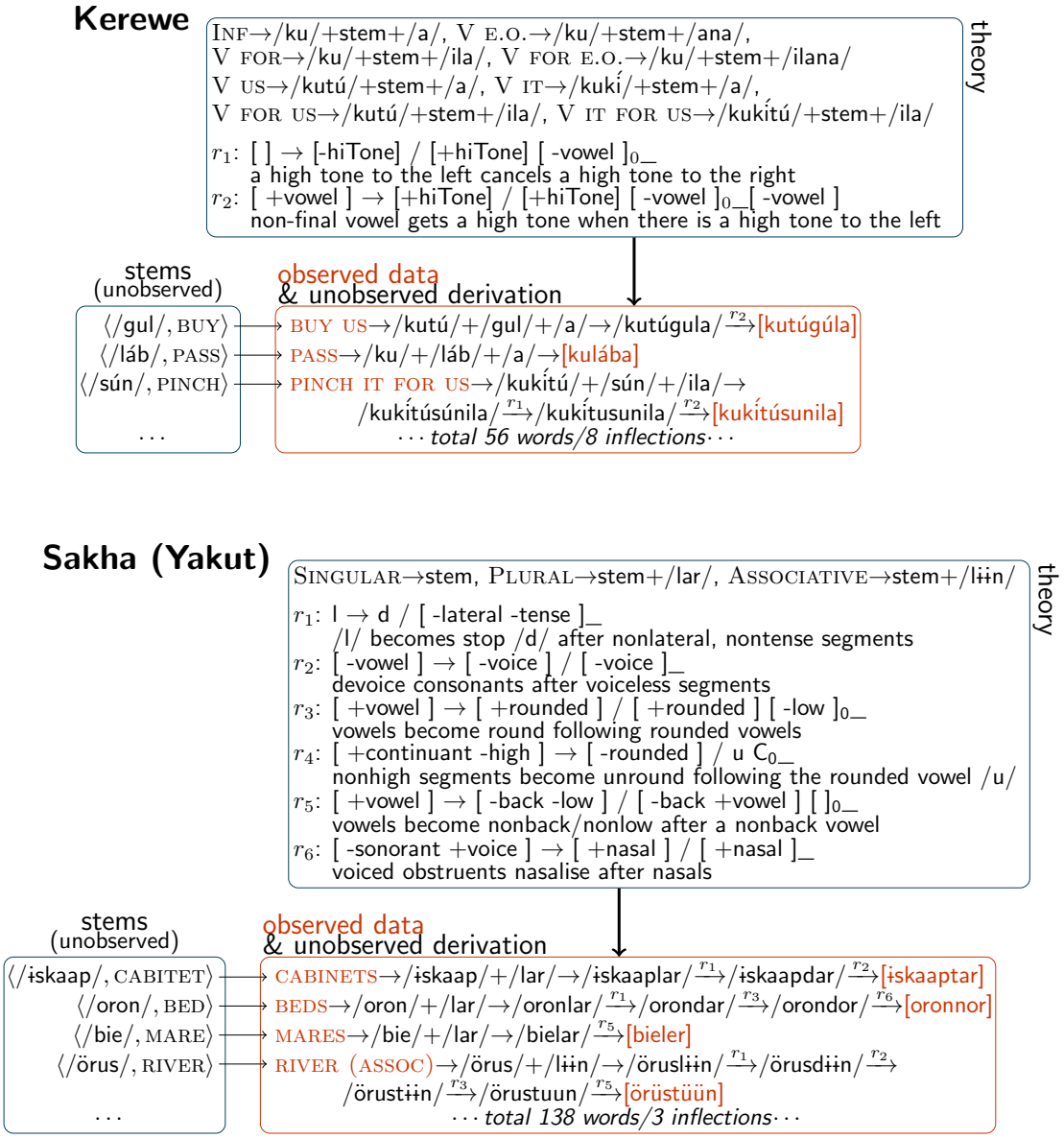


Figure 4-5: Qualitative results on morpho-phonological grammar discovery, illustrated on two phonology textbook problems. The model observes form/meaning pairs (orange) and jointly infers both a language-specific theory (morphology, top of "theory" box, written INFLECTION→ · · · ; phonological rules, bottom of "theory" box, labeled r_1, r_2, etc.) and a dataset-specific stems. Together the theory and stems explain the orange data via a derivation where the morphology output is transformed according to the ordered rules. Notice multiple vowel harmony rules in Sakha. Notice interacting nonlocal rules in Kerewe, a language with tones. Fig. 4-6,4-7,4-8 provides analogous illustrations for epenthesis and stress (Serbo-Croatian), more vowel harmony (Turkish & Hungarian), more complicated assimilation processes (Lumasaaba), and a representative partial failure case on Somali, where it recovers a partly correct rule set that fails to explain 20% of the data, while also illustrating spirantization.

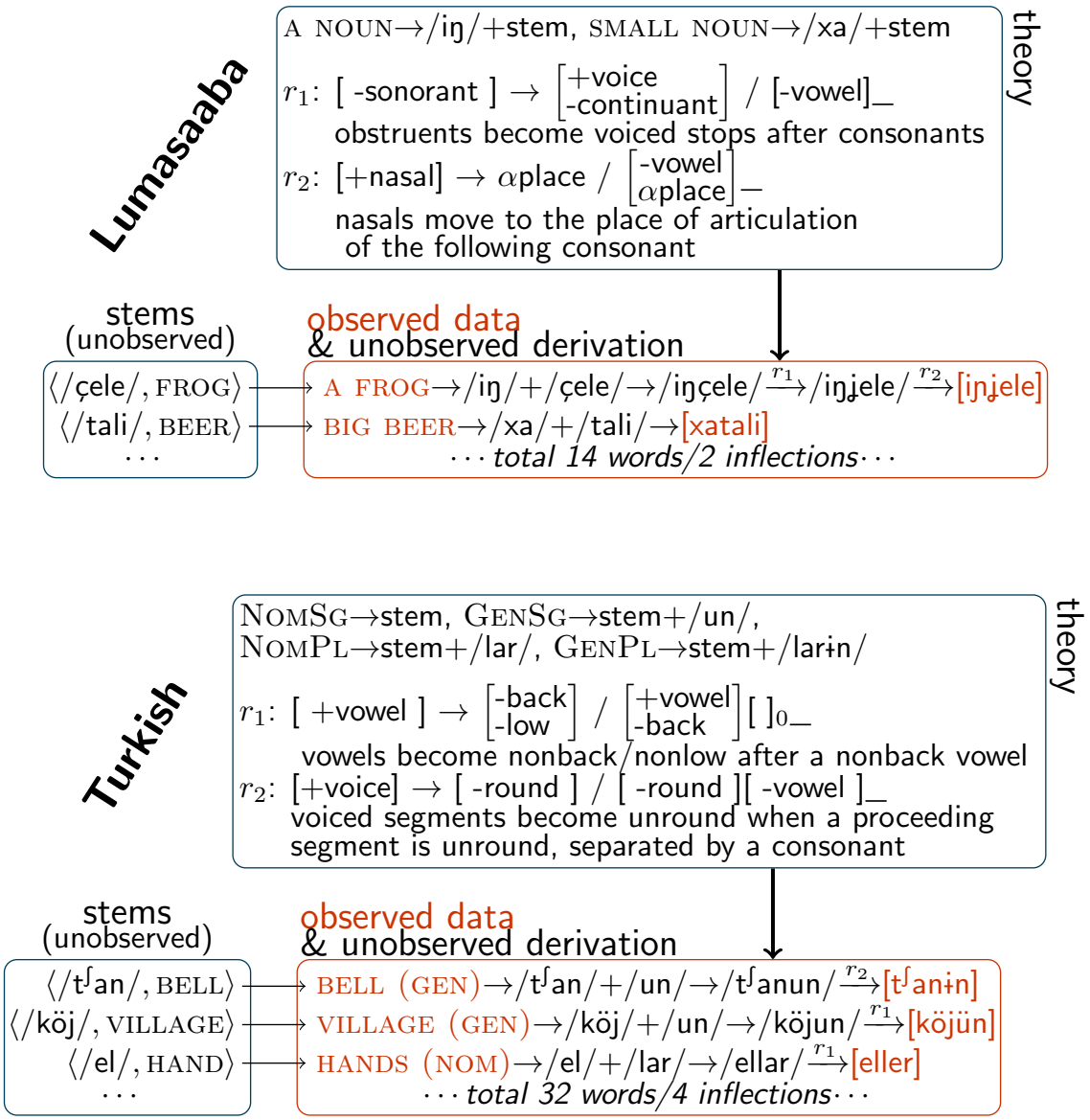


Figure 4-6: Given dataset, highlighted in orange, system jointly infers both language-specific theory (morphology, top of "theory" box, written INFLECTION→ · · · ; phonological rules, bottom of "theory" box, labeled r_1, r_2, etc.) and dataset-specific stems. Together the theory and stems explain the orange data via a derivation where the morphology output is transformed according to the ordered rules.

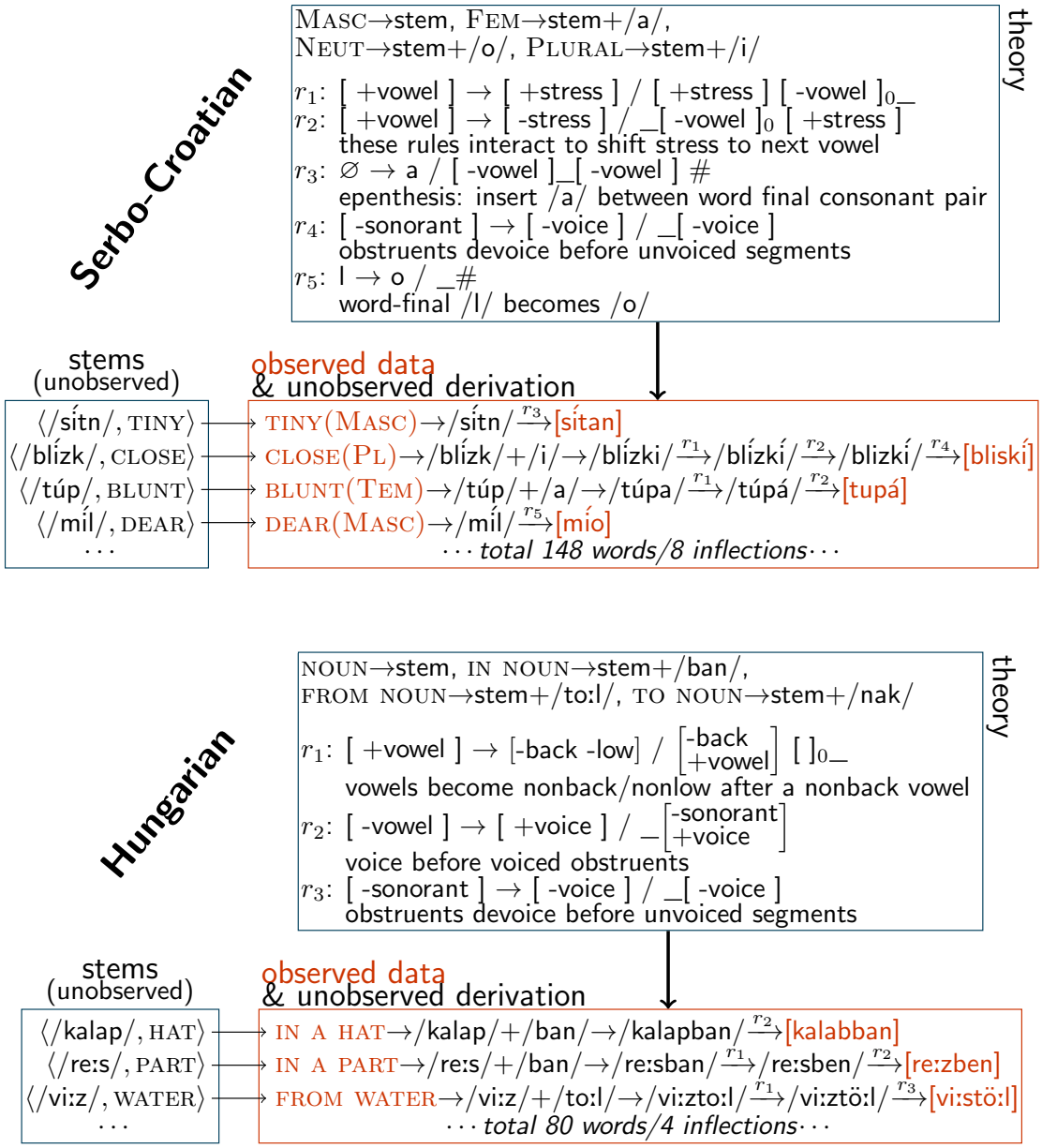


Figure 4-7: Given dataset, highlighted in orange, system jointly infers both language-specific theory (morphology, top of "theory" box, written INFLECTION→ · · · ; phonological rules, bottom of "theory" box, labeled r_1, r_2, etc.) and dataset-specific stems. Together the theory and stems explain the orange data via a derivation where the morphology output is transformed according to the ordered rules.

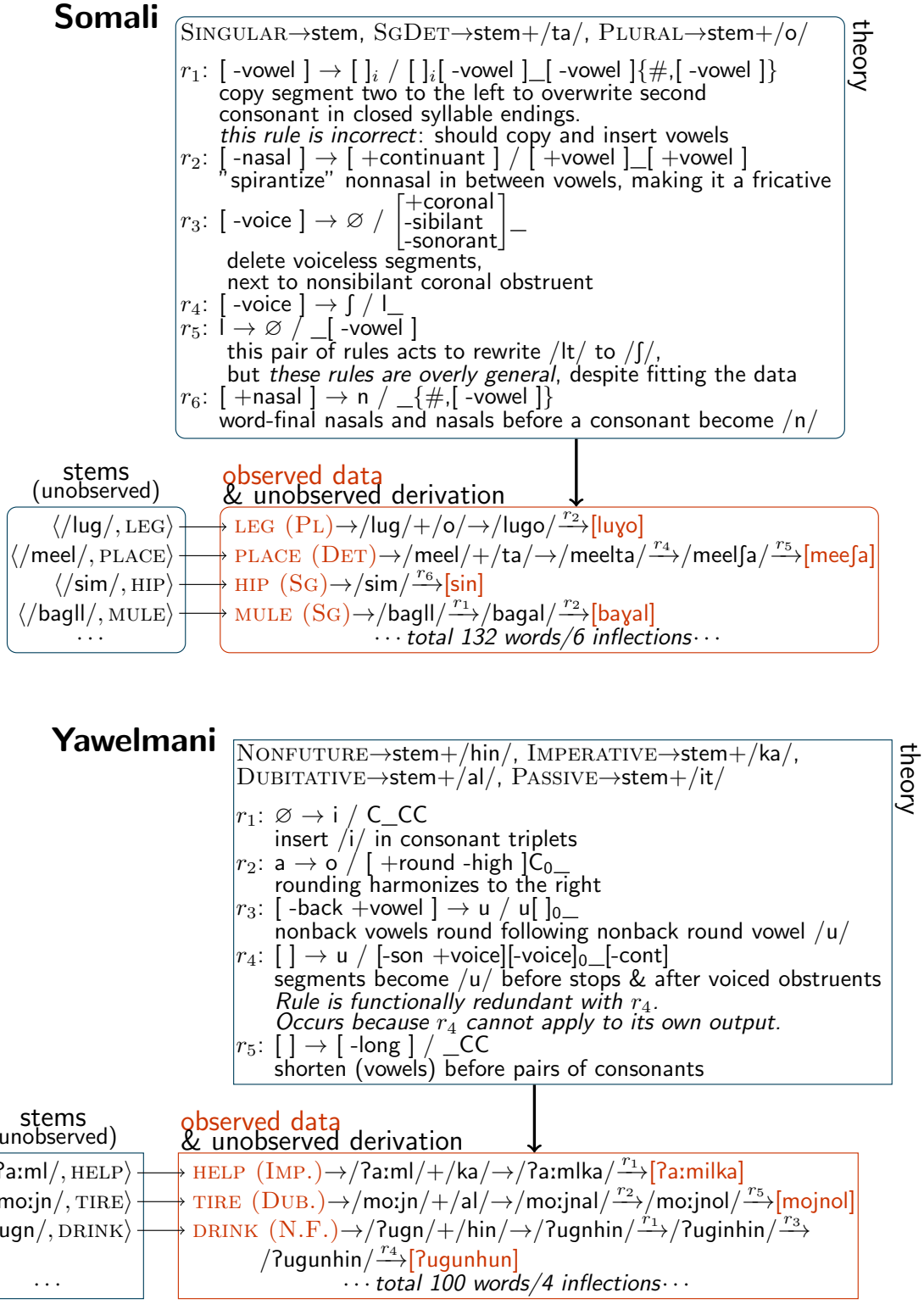


Figure 4-8: A representative failure mode for our system; illustration is analogous to Fig. 4-6,4-7. Rule system fails to explain 20% of the textbook problem, and many of the individual rules are implausible upon inspection, such as the first copying rule r_1 — see the bottom of derivation of the singular form of "mule." Other rules are essentially correct, such as the spirantization process implemented by rule r_2, or the neutralization process in r_6.

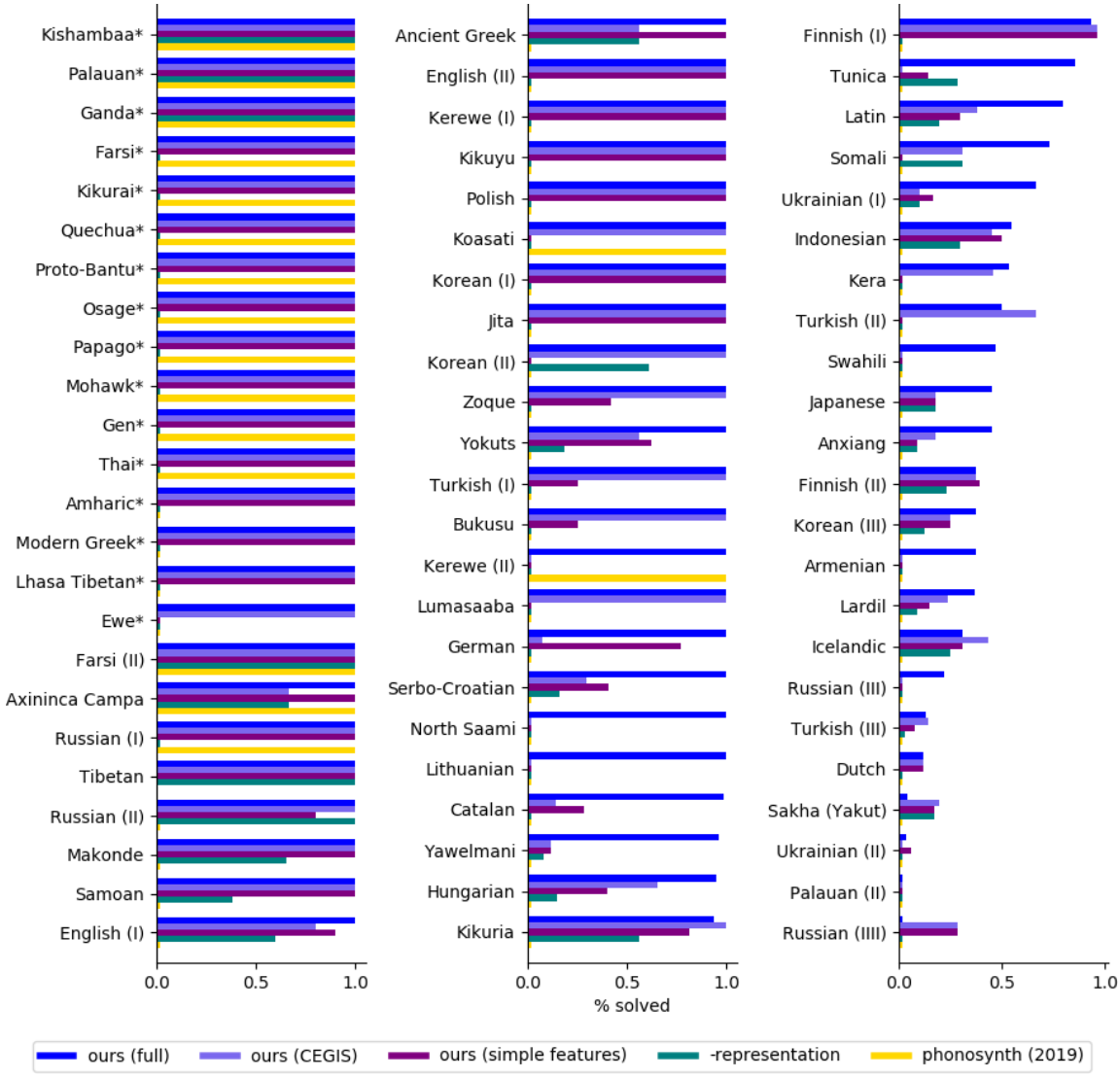


Figure 4-9: Program-synthesis-based phonological rule learner applied to problems from three widely-used phonology textbooks [97, 57, 111]. % solved: fraction of the phonology problem that is explained by the learned program and for which the stem matches gold ground-truth annotations. Problems marked with an asterisk are 'allophony' problems, and are typically easier. For allophony problems we count % solved as 0% when no rule explaining an alternation is found and 100% otherwise. Blue: Full model w/ incremental theory search. Green: Ablated model using the CEGIS program synthesis algorithm. Black: Ablated model w/o features and Kleene star. Yellow: PhonoSynth system developed in Barke et al. 2019 [11]. For allophony problems, full/CEGIS models are equivalent, because we batch the full problem at once

generalization of language-like patterns. We present our model with the artificial stimuli used in a number of AGL experiments [48, 85, 46] (Fig. 4-10A), systematically varying the quantity of data given to the model (Fig. 4-10B). The model demonstrates few-shot inference of the same language patterns probed in classic infant studies of AGL.

A notable feature of these AGL stimuli is the broad range of possible generalizations—the sparsity of the data leaves the inference problem highly underconstrained. Children select from this large space of possible generalizations to select the linguistically plausible ones. Thus, rather than producing a single grammar we use the model to search a massive space of possible grammars and then visualize all those grammars that are Pareto-optimal solutions [87] to the trade-off between *parsimony* (size of grammar) and *fit to data* (size of lexicon). Figure 4-12 visualizes the Pareto fronts for two classic artificial grammars as the number of example words provided to the learner is varied, illustrating both the set of grammars entertained by the learner, and how the learner weighs these grammars against each other. These figures show the *exact* contours of the Pareto frontier, a precision that comes from our use of constraint-based program synthesis techniques. With more examples the shape of the Pareto frontier develops a sharp kink around the correct generalization; with fewer examples the frontier is smoother and more diffuse. By explaining both natural language data and AGL studies, we see our model as delivering on a basic hypothesis underpinning AGL research: that artificial grammar learning must engage some cognitive resource shared with first language acquisition. To the extent that this hypothesis holds, we should expect an overlap between models capable of learning real linguistic phenomena, like ours, and models of AGL phenomena.

A

grammar	example input to learner	inferred grammar	natural language analogues
ABB Marcus et al. 1999.	wofefe lovivi fimumu	$\varnothing \rightarrow \sigma_i$ / σ_i_#	reduplication, e.g., Tagalog
ABA Marcus et al. 1999.	wofewo lovilo fimufi	$\varnothing \rightarrow \sigma_i$ / $\sigma_i\sigma$_#	reduplication
AAx Gerken 2006.	wowoka loloka fifika	stem+ka $\varnothing \rightarrow \sigma_i$ / #_σ_i	reduplication concatenative morphology Mandarin (Fig. 6, lower right)
AxA Gerken 2006.	wokawo lokalo fikafi	ka+stem $\varnothing \rightarrow \sigma_i$ / #_$\sigma\sigma_i$	infixing, e.g. Arabic reduplication
Pig Latin	pɪg→ɪgpe latɪn→atɪle æsk→æske	$\varnothing \rightarrow C_i$ / #C_i[ ]$_0$_# $\varnothing \rightarrow$ e / _# C $\rightarrow \varnothing$ / #_	child language games

B

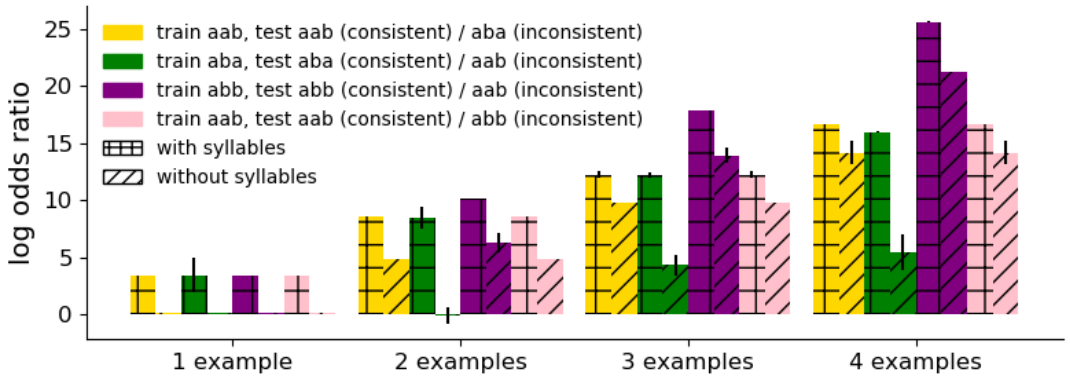


Figure 4-10: **A**: Children can learn (from few examples) many qualitatively different grammars, as studied in controlled conditions in AGL experiments. Our model learns these as well. NB: Actual reduplication is subtler than syllable-copying [107]. **B**: Model learns to discriminate between different artificial grammars by training on examples of a grammar (e.g., AAB) and then testing on either unseen examples of words drawn from the same grammar ("consistent" condition, e.g. new words following the AAB pattern); or testing on unseen examples of words from a different grammar ("inconsistent" condition, e.g. new words following the ABA pattern), following the paradigm of Marcus et al. 1999. We plot log-odds ratio of consistent and inconsistent conditions: $\log P(\text{consistent}|\text{train})/P(\text{inconsistent}|\text{train})$. Positive log-odds indicate successful discrimination. Error bars: $2 \times$ stddev over eight test words. We contrast models using program spaces both with and without syllabic representations, which were not used in textbook problems. Syllabic representation (crosses) proves important for few-shot learning, but a model without syllables (slants) can still discriminate successfully given enough examples by learning rules that copy individual phonemes. See Fig. 4-11 for more examples.

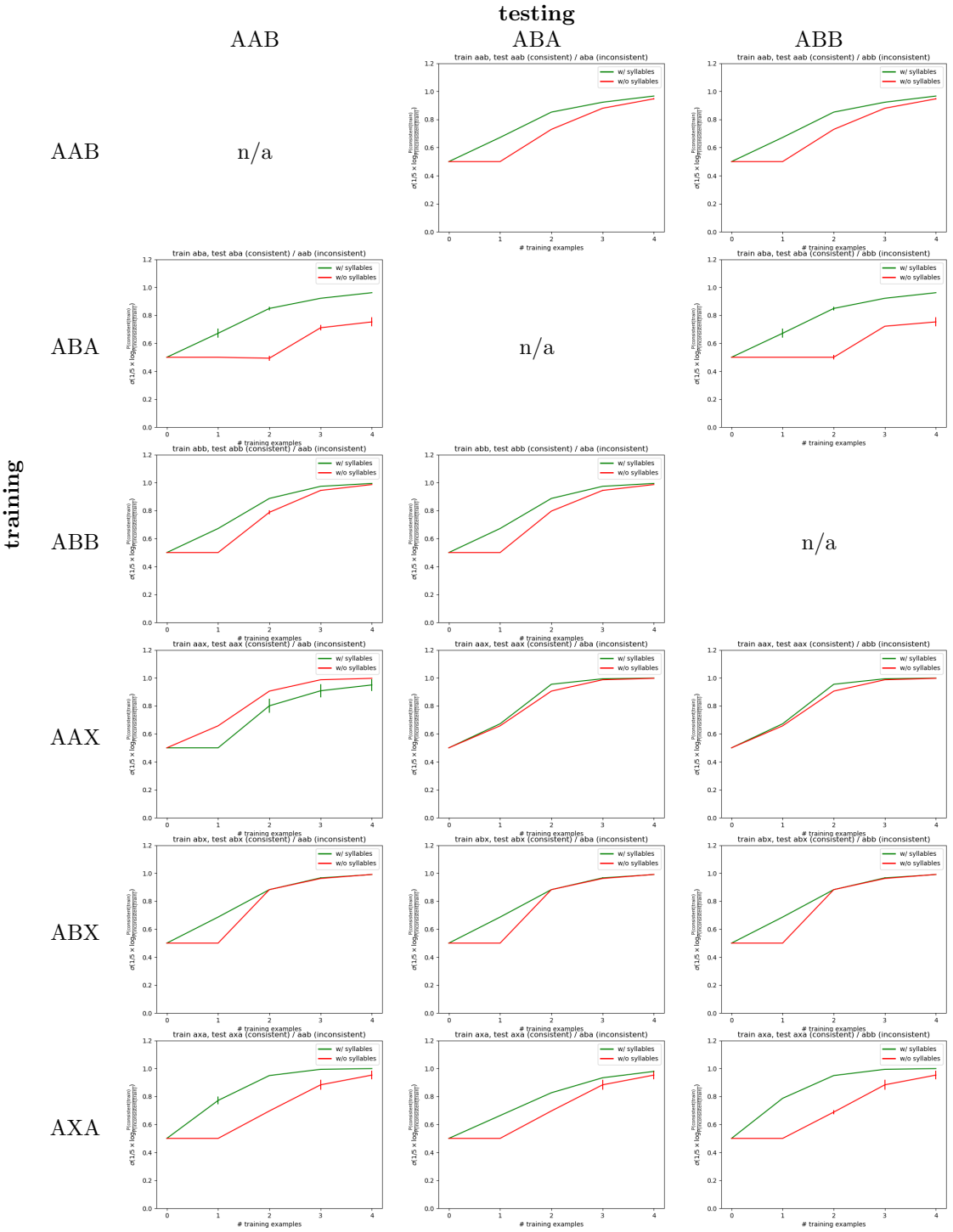


Figure 4-11: Capturing a range of human generalizations found within the artificial grammar learning literature. Horizontal axis is # of examples; Vertical axis is sigmoid of log-odds ratio between positive and negative examples, with a temperature parameter of 5. Compare with Figure 4-10B. Green: w/ syllabic representation. Red: w/o syllabic representation. Error bars represent standard deviation over 4 randomly generated held out words conforming to either the training distribution ("consistent" examples) or the testing distribution ("inconsistent") examples)

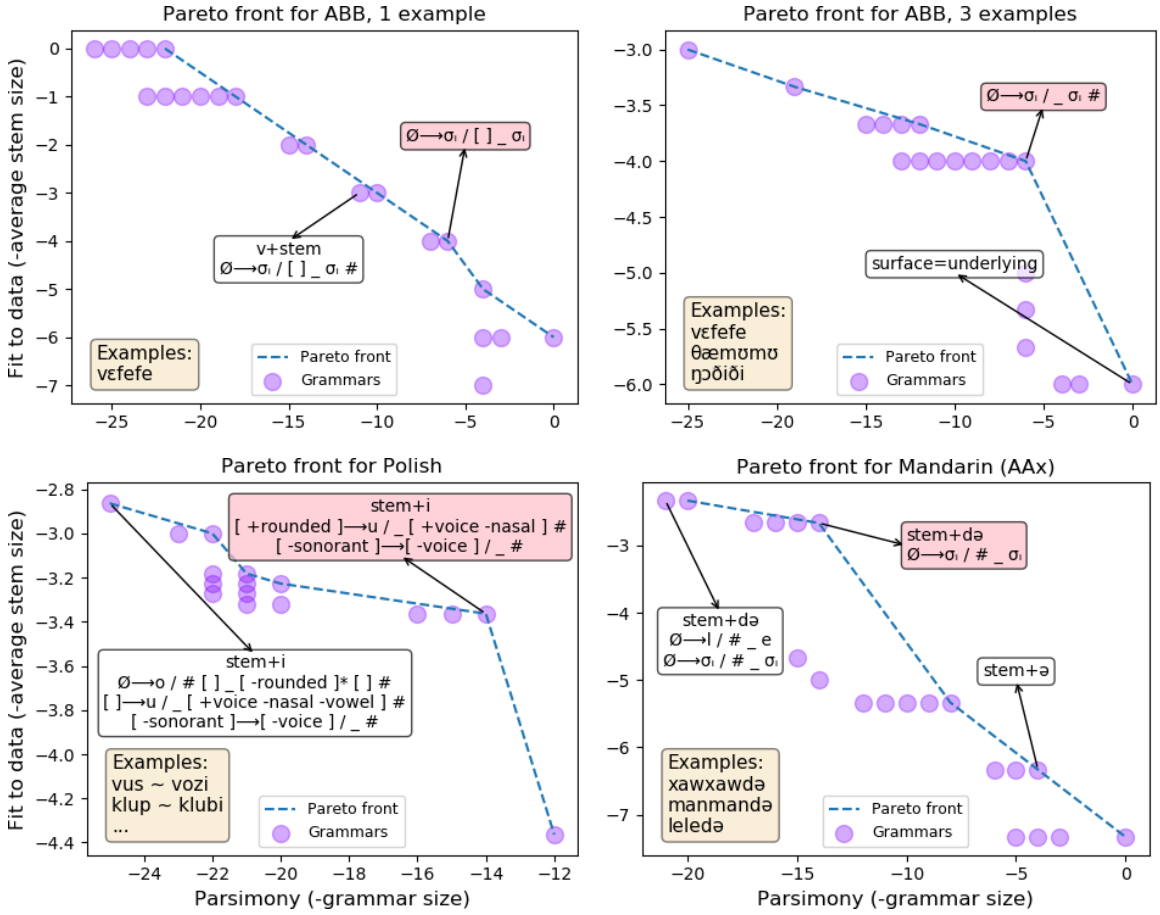


Figure 4-12: Few-shot learning of language patterns can give high ambiguity as to the correct grammar. Here we visualize the geometry of generalization for several natural and artificial grammar learning problems. These visualizations are Pareto frontiers: the set of solutions consistent with the data which optimally trade-off between parsimony and fit to data. We show Pareto fronts for ABB (Marcus 1999; top two) & AAX (Gerken 2006; bottom right, data drawn from isomorphic phenomena in Mandarin) AGL problems for either one example word (upper left) or three example words (right column). In the bottom left we show the Pareto frontier for a textbook Polish morpho-phonology problem. Rightward on x-axis corresponds to more parsimonious grammars (smaller rule size + affix size) and upward on y-axis corresponds to grammars that best fit the data (smaller stem size), so the best grammars live in the upper right corners of these graphs. Pink shade: correct grammar. As the number of examples increases, the Pareto fronts develop a sharp kink around the correct grammar, which indicates a stronger preference for the correct grammar. With one example the kinks can still exist but are less pronounced. The blue lines provably show the exact contour of the Pareto frontier, up to the bound on the number of rules. This precision is owed to our use of exact constraint solvers.

4.3 Synthesizing Higher-Level Theoretical Knowledge

No theory is built from scratch: Instead, researchers borrow concepts from existing frameworks, make analogies with other successful theories, and adapt general principles to specific cases. Through analysis and modeling of many different languages, phonologists (and linguists more generally) develop overarching meta-models that restrict and bias the space of allowed grammars. They also develop the "phonological common-sense" that allows them to infer grammars from sparse data, knowing which rule systems are plausible based on their prior knowledge of human language, and which systems are implausible or simply unattested. For example, many languages *devoice* word-final obstruents, but no language *voices* word-final obstruents. This cross-theory common-sense is found in other sciences. For example, physicists know which potential functions tend to occur in practice: A power-law pair-wise potential is more plausible than an 8-way sinusoidal potential. Thus a key objective for our work is the automatic discovery of a cross-language metamodel capable of imparting "phonological commonsense."

Conceptually, this meta-theorizing corresponds to estimating a prior, $\mathbb{M}$, over language-specific theories, and performing hierarchical Bayesian inference across many languages. Concretely, we think of the meta-theory $\mathbb{M}$ as being a set of schematic, highly reusable phonological-rule templates, encoded as a probabilistic grammar over the structure of phonological rules, and we will estimate both the structure and the parameters of this grammar jointly with the solutions to textbook phonology problems. To formalize a set of meta-theories and define a prior over that set, we use the Fragment Grammars formalism [98], a probabilistic grammar learning set up which caches and reuses pieces, or 'fragments', of commonly used rule subparts.

Assuming we have a collection of D datasets (e.g., from different languages), notated $\{\mathbf{X}_d\}_{d=1}^{D}$, our model constructs D grammars, $\{\langle \mathbb{T}_d, \mathbb{L}_d \rangle\}_{d=1}^{D}$, along with a meta-theory $\mathbb{M}$,

seeking to maximize $P(\mathbb{M}) \prod_{d=1}^{D} P(\mathbb{T}_d, \mathbb{L}_d|\mathbb{M}) P(\mathbf{X}_d|\mathbb{T}_d, \mathbb{L}_d)$ where $P(\mathbb{M})$ is a prior on fragment grammars over SPE-style rules. In practice, jointly optimizing over the space of $\mathbb{M}$s and grammars is intractable, and so we instead alternate between finding high-probability grammars under our current $\mathbb{M}$, and then shifting our inductive bias, $\mathbb{M}$, to more closely match the current grammars. Formally we maximize a variational lower bound on the joint probability of the metamodel and the data sets:

$$\log P(\mathbb{M}, \{\mathbf{X}_d\}_{d=1}^{D}) \geq \log P(\mathbb{M}) + \sum_{d=1}^{D} \log \sum_{\langle \mathbb{T}_d, \mathbb{L}_d \rangle \in \text{support}[Q_d(\cdot)]} P(\mathbf{X}_d|\mathbb{T}_d, \mathbb{L}_d) P(\mathbb{T}_d, \mathbb{L}_d|\mathbb{M})$$

where this bound is written in terms of a set of variational approximate posteriors, $\{Q_d\}_{d=1}^{D}$, whose support we constrain to be small, which ensures that the above objective is tractable. We alternate maximization with respect to $\mathbb{M}$ (i.e., inferring a fragment grammar from the theories in the supports of $\{Q_d\}_{d=1}^{D}$), and maximization with respect to $\{Q_d\}_{d=1}^{D}$ (i.e., finding a small set of theories for each data set that are likely under the current $\mathbb{M}$). Our lower bound most increases when the support of each $\{Q_d\}_{d=1}^{D}$ coincides with the top-k most likely theories, so at each round of optimization, we ask the program synthesizer to find the top k theories maximizing $P(\mathbf{X}_d|\mathbb{T}_d, \mathbb{L}_d) P(\mathbb{T}_d, \mathbb{L}_d|\mathbb{M})$. In practice we find the top $k = 100$ theories for each data set. We estimate $\mathbb{M}$ by applying this procedure to a training subset comprising 30 problems, chosen to exemplify a range of distinct phenomena, and then applied this $\mathbb{M}$ to all 70 problems. Critically this procedure is not given access to any ground-truth solutions to the training subset.

This machine-discovered higher-level knowledge serves two functions. First, it is a form of human understandable knowledge: manually inspecting the contents of the fragment grammar reveals cross-language motifs previously discovered by linguists (Fig. 4-13C, Fig. 4-14). Second, it is critical to actually getting these problems correct: grammars discovered in isolation are frequently wrong (Fig. 4-13A,B, middle column of Fig. 4-13C).

Only with a refined inductive bias, or linguistic commonsense, can language be learned effectively from sparse data.

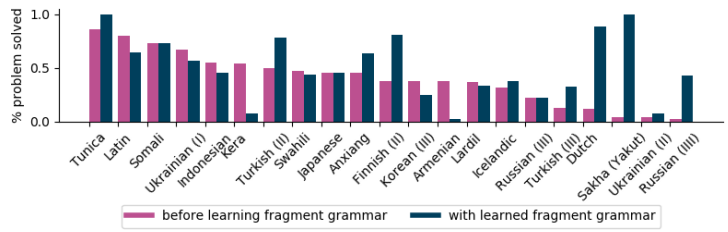
To be clear, our mechanized meta-theorizing is *not* an attempt to "learn universal grammar" (cf. Perfors et al. 2011 [102]). Rather than capture a learning process, our meta-theorizing is analogous to a discovery process: distilling out knowledge of typological tendencies, which aids future model synthesis. We believe however that the child's innate representation of these tendencies [130], through either channel or analytic biases [92], contributes to their skill as language learners, and that the linguist's skill in inferring grammars draws on analogous forms of knowledge.

4.4 Lessons

Our work provides a working demonstration that it is possible for an algorithm to automatically discover human understandable knowledge about the structure of natural language. Like linguists, optimal inference hinges on higher-level biases and constraints; but the toolkit developed here permits systematic probing of these abstract assumptions, and data-driven discovery of cross-language trends. Our work speaks to a long-standing analogy between the problems confronting children and linguists, and computationally cashes out the basic assumptions that underpin infant and child studies of artificial grammar learning.

More broadly, the tools and approaches developed here suggest routes for machines that learn the causal structure of the world, while representing their knowledge in a format that can be reused and communicated to other agents, both natural and artificial. While this goal remains far off, it is worth taking stock of where this work leaves us on the path toward a theory induction machine: what are the prospects for scaling an approach like ours to all of linguistics, or all of science? Scaling to the full linguistic hierarchy—acoustics, phonotactics, syntax, semantics, pragmatics—will require more powerful programming

A



B **Sakha (Yakut)**

w/o meta-model (solves 4% problem)

∅ → o / [-rounded -coronal]_{[+liquid],#}
V → [-back -low] / [-back +vowel] [ ]$_0$_
[ ] → [-rounded] / [-rounded +vowel] [ ]$_0$_

w/ meta-model (solves 100% problem)

l → d / [-lateral -tense]_
C → [-voice] / [-voice]_
V → [+rounded] / [+rounded] [-low]$_0$_
[+continuant -high] → [-rounded] / u C$_0$_
V → [-back -low] / [-back +vowel] [ ]$_0$_
[-sonorant +voice] → [+nasal] / [+nasal]_

C

Discovered rule schema(s)		Rules without discovered meta-model	Rules with learned meta-model
consonant/vowel distinction FM: := [-vowel] FM: := [+vowel] *(FM=featurematrix)*	Tibetan	[-nasal] → ∅ / #_	C → ∅ / # _ C
	Kerewe	[ ] → [+hiTone] / [+hiTone] [ ] _ [ ]	V → [+hiTone] / [+hiTone] C$_0$ _ C
nasal place assimilation [+nasal] → αplace / trigger_[C, αplace]	Bukusu	ñ → αplace / _[αplace]	[+nasal] → αplace / _[C, αplace]
	Koasati	m → αplace / _[C, αplace]	[+nasal] → αplace / _[C, αplace]
vowel harmony V → [-back, -low] / [V, -back] FM$_0$_	Turkish	[ ] → [-back, -low] / [-back, -low] _ [V, -back] [ ]$_0$	V → [-back, -low] / [-back, -low] _ [V, -back] [ ]$_0$
spirantization FM → [+cont] / FM_V	Somali	[-nasal] → [+cont] / V_[-coronal]	[-nasal] → [+cont] / V_V
	Catalan	[+voice, -nasal] → [+cont] / [ ] [+cont]_	[+voice, -sonorant] → [+cont] / [-nasal]_V

Figure 4-13: Discovering and using a cross-language metatheory. **(A)** Re-solving the hardest textbook problems using estimated fragment grammar over phonological rules leads to an average of 25% more of the problem being solved, while **(B)** illustrates a particular case where these discovered tendencies allow the model to find a set of six interacting rules solving the entirety of an unusually complex problem. **(C)** These rule schemas are human understandable and often correspond to motifs previously identified within linguistics. Left column shows four out of 21 induced rule fragments, which encode cross-language tendencies. Learned schemas such as vowel harmony and spirantization (a process whereby stops become fricatives in the vicinity of vowels) are shown. The symbol `FM` means a slot that can hold any feature matrix, and `trigger` means a slot that can hold any rule triggering context. Middle column shows model output when solving each language in isolation: these solutions can be overly specific (Koasati, Bukusu), overly general (Kerewe, Turkish), or even essentially unrelated to the correct generalization (Tibetan). Right column shows model output when solving problems jointly with inferring a metatheory.

Rule::= FC → FC / Trigger_Trigger
Rule::= [+vowel] → [-highTone] / _[-vowel]* [+highTone +vowel]
Rule::= [-sonorant] → [-voice] / _Trigger
Rule::= [+nasal] → αplace / Trigger_[-vowel]
Rule::= [+vowel] → [+highTone] / [+highTone +vowel] [-vowel]*_Trigger
Rule::= [+vowel] → [-back -low] / [-back +vowel] FeatureMatrix*_
Rule::= [-vowel] → [+voice] / _[-sonorant +voice]
Rule::= FC → [+continuant] / FeatureMatrix_[+vowel]
Rule::= [-sonorant] → FeatureMatrix / _[-sonorant]
Rule::= [+voice] → FC / FeatureMatrix [-vowel]_Trigger
Rule::= [-voice] → [+voice] / [+nasal] [-vowel]*_
Trigger::= #
Trigger::= FeatureMatrix #
Trigger::= FeatureMatrix* FeatureMatrix #
Trigger::= FeatureMatrix FeatureMatrix #
Trigger::= {#,FeatureMatrix}
Trigger::= FeatureMatrix* {#,FeatureMatrix}
Trigger::= FeatureMatrix {#,FeatureMatrix}
Trigger::=
Trigger::= FeatureMatrix
Trigger::= FeatureMatrix* FeatureMatrix
Trigger::= FeatureMatrix FeatureMatrix
Trigger::= [-vowel]* [+highTone +vowel]
FeatureMatrix::= +
FeatureMatrix::= ConstantPhoneme
FeatureMatrix::= [-sonorant]
FeatureMatrix::= [+vowel +highTone]
FeatureMatrix::= [-vowel]
FeatureMatrix::= [-voice]
FeatureMatrix::= [+vowel]
FeatureMatrix::= [-back +vowel]
FeatureMatrix::= [-sonorant +voice]
FeatureMatrix::= [+nasal]
FeatureMatrix::= [+voice]
FeatureMatrix::= [+continuant -high]
FC::= $\varnothing$
FC::= CopyOffset
FC::= αplace
FC::= FeatureMatrix

Figure 4-14: Here we show the learned fragment grammar over SPE-style rules (compare with Figure 4-2, 4-13). Nonterminal symbols include "Rule", "FeatureMatrix", "Trigger", "FC" (nonterminal symbol for the focus and structural change of SPE-style rewrites), and "CopyOffset" (which ranges over integers and is used in copying rules, indexing into which feature matrix in the left/right trigger will be copied over into the focus of the SPE-style rewrite rule).

languages for expressing symbolic rules, and more scalable inference procedures, because although the textbook problems we solve are harder than prior work tackles, actual morphophonology is still larger and more intricate than the problems considered here.

Scaling to real scientific discovery demands fundamental innovations, but holds promise. Unlike language acquisition, genuinely new scientific theories are hard-won, developing over timescales that can span a decade or more. They involve the development of new formal substrates and new vocabularies of concepts, such as 'force' in physics and 'allele' in biology. We suggest three lines of attack. Drawing inspiration from conceptual role semantics [15], future automated theory builders could introduce and define new theoretical objects in terms of their interrelations to other elements of the theory's conceptual repertoire, only at the end grounding out in testable predictions. Drawing on the findings of our work here, the most promising domains are those which are solvable, in some version, by both child learners and adult scientists. This means first investigating sciences with counterparts in intuitive theories, such as classical mechanics (and intuitive physics), or cognitive science (and folk psychology). Building on our findings, a crucial element of theory induction will be the joint solving of many interrelated model building problems, followed by the synthesis of abstract over-hypotheses that encapsulate the core theoretical principles while simultaneously accelerating future induction through shared statistical strength.

Theory induction is a grand challenge for AI, and our work here captures only small slices of the theory building process. Like our model, human theorists do craft models by examining experimental data, but also propose new theories by unifying existing theoretical frameworks, performing 'thought experiments', and inventing new formalisms. Humans also deploy their theories more richly than our model: proposing new experiments to test theoretical predictions, engineering new tools based on the conclusions of a theory, and distilling higher-level knowledge that goes far beyond what our Fragment-Grammar approximation can represent. Continuing to push theory induction along these many

dimensions remains a prime target for future research.

Acknowledgments

This chapter contains material produced in collaboration with Timothy O'Donnell, Adam Albright, Josh Tenenbaum and Armando Solar-Lezama. The dissertation author was the primary researcher on this work, which is presently unpublished.

Chapter 5

DreamCoder: A more generic platform for program induction

While learning has played a key role in the systems described in Chapters 3 and 4, all of these systems are special-purpose program induction engines for a specific class of problems, and differ not just in implementation details, but also in the underlying algorithms and techniques: Neural networks for graphics, SAT solving for linguistics and 2-D images (but not 3-D), a learned language bias for linguistics but only for linguistics, reinforcement learning for 3-D CAD and bias-optimal search for 2-D diagrams. This observation suggests that the next challenge is to generalize these approaches to create a single system which can, with a relatively modest amount of domain-specific knowledge and data, learn to induce programs for a new class of problems.

Returning to the two obstacles laid out at the outset of this thesis, we take as our goal to build a generic system for learning to induce programs that bootstraps its own inductive bias and its own search strategy. 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. 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 [43], and the human expertise literature [22, 23]. 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 [22, 23], 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 [30, 81, 73], and neurally-guided program synthesis [10, 33]. Both these ideas have been separately influential, but have only been brought together in our work starting with the EC^2 algorithm [37], a forerunner of DreamCoder. 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 [61]. 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.

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. 5-21A,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.

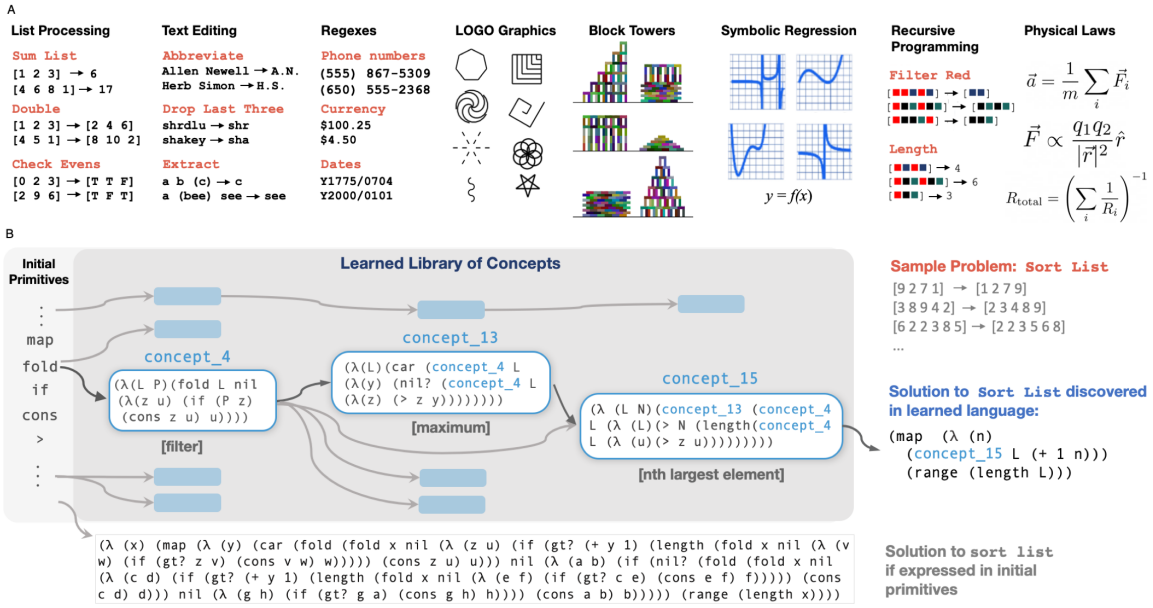


Figure 5-1: **(A)**: Eight problem-solving domains DreamCoder is applied to. **(B)**: An illustration of how DreamCoder learns to solve problems in one domain, processing lists of integers. Problems are specified by input-output pairs exemplifying a target function (e.g., 'Sort List'). Given initial 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.

5.1 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 [61]. Each DreamCoder iteration (Eq. 5.3, Fig. 5-2) 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. 5-2 left). This mechanism increases the breadth and depth of learned libraries like those in Fig. 5-1B and Fig. 5-21, 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. 5-2 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. 5-2 middle). This prior is encoded by a library, written $\mathcal{D}$, which when equipped with a real-valued weight vector, written θ, defines a generative model over programs, written $\mathrm{P}[\rho|\mathcal{D},\theta]$. The library $\mathcal{D}$ is a set of typed λ-calculus expressions. Using this notation, the joint distribution, J, over the observed tasks and the latent variables is

$$J(\mathcal{D},\theta) \triangleq \mathrm{P}[\mathcal{D},\theta] \prod_{x\in X} \sum_{\rho} \mathrm{P}[x|\rho]\mathrm{P}[\rho|\mathcal{D},\theta]$$
$$\mathcal{D}^* = \arg\max_{\mathcal{D}} \int J(\mathcal{D},\theta)\ \mathrm{d}\theta \qquad \theta^* = \arg\max_{\theta} J(\mathcal{D}^*,\theta) \tag{5.1}$$

where $\mathrm{P}[x|\rho]$ scores the likelihood of a task $x \in X$ given a program ρ.[1]

Evaluating Eq. 5.1 entails marginalizing over the infinite set of all programs – which is

[1]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.

impossible. We make a particle-based approximation to Eq. 5.1 and instead marginalize over a finite **beam** of programs, with one beam per task, collectively written $\{\mathcal{B}_x\}_{x\in X}$. This particle-based approximation is written $\mathscr{L}(\mathcal{D},\theta,\{\mathcal{B}_x\})$ and acts as a lower bound on the joint density:[2]

$$J(\mathcal{D},\theta) \geq \mathscr{L}(\mathcal{D},\theta,\{\mathcal{B}_x\}) \triangleq \mathrm{P}[\mathcal{D},\theta] \prod_{x\in X} \sum_{\rho\in\mathcal{B}_x} \mathrm{P}[x|\rho]\mathrm{P}[\rho|\mathcal{D},\theta], \text{ where } |\mathcal{B}_x| \text{ is small} \quad (5.2)$$

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 $(\mathcal{D},\theta)$ (**Abstraction**):

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

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

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

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

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

Putting these ingredients together, Equation 5.3 summarizes DreamCoder's steps. These updates will be further refined, explained, and approximated in Sections 5.1.1-5.1.3.

$$\begin{aligned}
\mathcal{B}_x = & \operatorname*{argtop}_{\substack{\rho: \\ Q(\rho|x) \text{ is large}}} k \;\; \mathrm{P}[\rho|x,(\mathcal{D},\theta)] \propto \mathrm{P}[x|\rho]\mathrm{P}[\rho|(\mathcal{D},\theta)], \text{ for each task } x \in X && \textit{Wake} \\
\mathcal{D} = & \arg\max_{\mathcal{D}} \int \mathrm{P}\left[\mathcal{D},\theta\right] \prod_{x\in X} \sum_{\rho\in\mathcal{B}_x} \mathrm{P}[x|\rho]\mathrm{P}[\rho|\mathcal{D},\theta] \; \mathrm{d}\theta && \textit{Sleep: Abstraction} \\
\text{Train } Q(\rho|x) \approx & \; \mathrm{P}[\rho|x,L], \text{ where } x \sim X \text{ ('replay') or } x \sim L \text{ ('fantasy')} && \textit{Sleep: Dreaming}
\end{aligned} \tag{5.3}$$

Algorithm 1 provides pseudocode for the full DreamCoder algorithm. We interleave an extra wake cycle between each sleep stage, and during this extra waking phase we simply enumerate from the prior. We can also interpret this extra waking phase as a part of dreaming: we are drawing from the prior ("dreaming") and checking if those draws solve any tasks. We favor this interpretation, and indeed, in the actual software implementation this extra wake cycle occurs concurrently the generation of dreams.

Algorithm 1 Full DreamCoder algorithm

function DreamCoder($\mathcal{D}, X$):
Input: Initial library functions $\mathcal{D}$, 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: $\mathcal{B}_x \leftarrow \mathcal{B}_x \cup \{\rho \mid \rho \in \text{enumerate}(\text{P}[\cdot|\mathcal{D}, \theta], T) \text{ if } \text{P}[x|\rho] > 0\}$ ▷ Enumerate from prior
        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 $\text{P}[\cdot|x, \mathcal{D}, \theta]$ ▷ Keep top M programs
        $\mathcal{D}, \theta, \{\mathcal{B}_x\}_{x \in X} \leftarrow$ABSTRACTION$(\mathcal{D}, \theta, \{\mathcal{B}_x\}_{x \in X})$ ▷ Abstraction Sleep
        yield $(\mathcal{D}, \theta)$, Q, $\{\mathcal{B}_x\}_{x \in X}$ ▷ Yield the updated library, recognition model, and solutions found to tasks
    end while
end while

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.[3]

5.1.1 Wake: Program Synthesis

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 ($\mathcal{B}_x$) of the $k = 5$ programs with the highest posterior probability $\mathrm{P}[\rho|x, (\mathcal{D}, \theta)]$, which one can see is marginalized over in the abstraction sleep update of Eq. 5.3. We represent programs as polymorphically

[3]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.

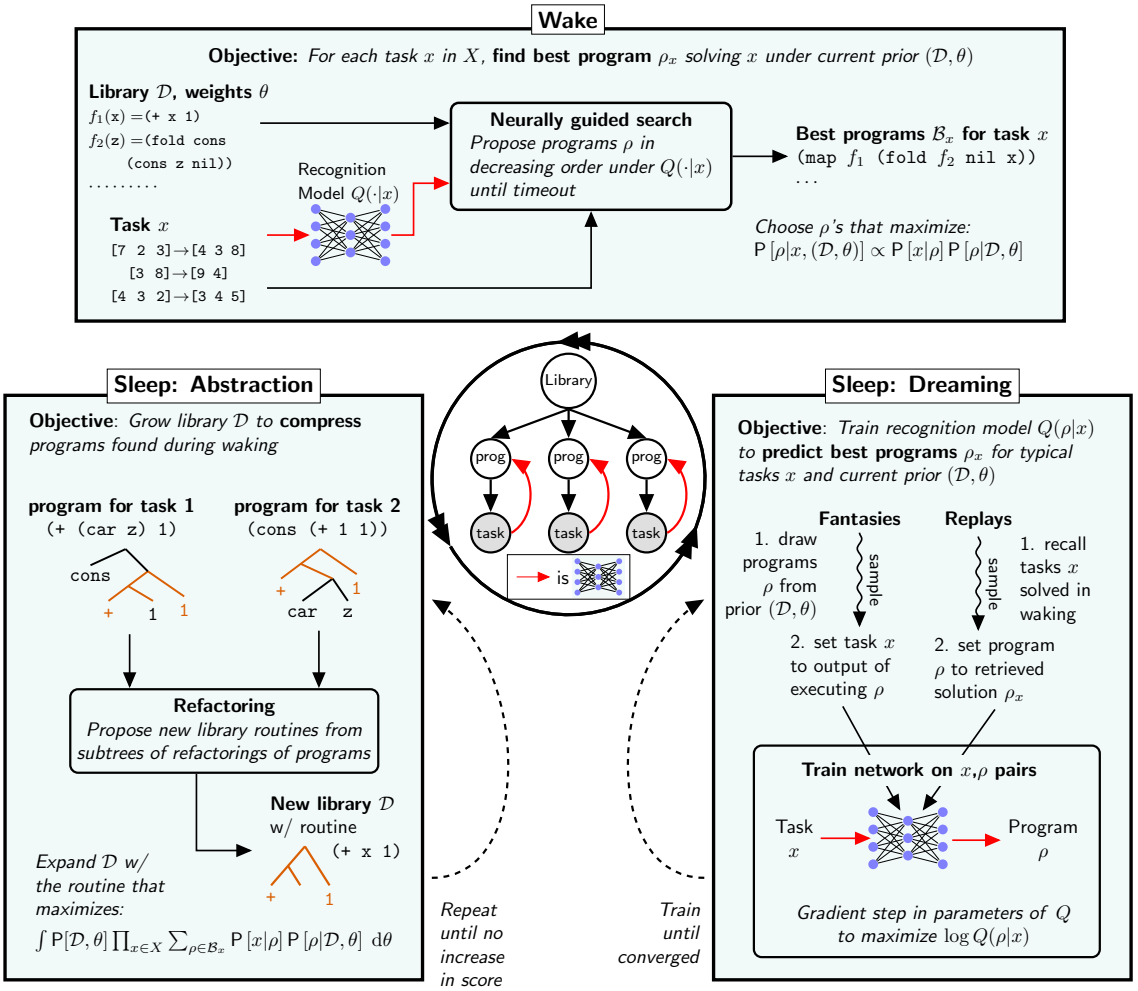


Figure 5-2: DreamCoder's basic algorithmic cycle, which serves to perform approximate Bayesian inference for the graphical model diagrammed in the **middle**. The system observes programming tasks (e.g., input/outputs for list processing or images for graphics programs), which it explains with latent 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).

typed λ-calculus expressions, an expressive formalism including conditionals, variables, higher-order functions, and the ability to define new functions.

Waking is thus a fusion of bottom-up neural guidance (from the recognition model), top-down inductive biases (from the generative model, i.e. the library), and symbolic search (via enumeration). For enhanced scalability, we engineered a parallel enumeration algorithm. We combine two different enumeration strategies, which allowed us to build a this 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 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 recursion's stack (thus logarithmically as a function of run time).

Our parallel enumeration algorithm (Algorithm 2) 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 [112]. Algorithm 2 takes as input a distribution over programs, written here as μ. We define $\mu(\rho)$ when ρ has "holes" as $\sum_{\rho' \in \rho} \mu(\rho')$ where $\rho' \in \rho$ means that ρ' has no holes and is a prolongation of ρ in the sense of [116].

Algorithm 2 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

5.1.2 Abstraction Sleep: Growing the Library

During the abstraction phase of sleep, the model grows its library 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, when suitably approximated, the abstraction sleep objective (Eq. 5.3) is equivalent to minimizing the description length of the library plus the description lengths of programs found during waking. Intuitively, we will "compress out" reused code to maximize a Bayesian criterion. To formalize this Bayesian compression, we define a library-conditioned generative model over programs (Algorithm 3), which then implicitly defines $\mathrm{P}\left[\rho|\mathcal{D},\theta\right]$. A new contribution of DreamCoder's abstraction phase is that rather than compress out reused syntactic structures, we will *refactor* programs to expose reused semantic patterns. The exposition of abstraction sleep will proceed in four steps: first, approximating the objective in Equation 5.3 to avoid an expensive marginalization; second, relaxing the objective to allow refactorings that best compress programs found during waking; third, introducing algorithmic machinery that makes this refactoring efficient; and fourth, tying these pieces together to build the abstraction sleep algorithm.

Algorithm 3 The library-conditioned generative model over programs. This stochastic procedure takes as input the desired type of the program, and performs type inference [29] 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.

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

function sample$'(\mathcal{D}, \theta, \mathcal{E}, \tau)$:
Input: Library $(\mathcal{D}, \theta)$, 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$'(\mathcal{D}, \theta, \{\text{var} : \alpha\} \cup \mathcal{E}, \beta)$ ▷ Recursively sample function body
    return `(lambda (`var`)` body`)`
else ▷ Build an application to give something w/ type τ
    primitives $\leftarrow \{\rho | \rho : \tau' \in \mathcal{D} \cup \mathcal{E}$ 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. $\propto \begin{cases} \theta_e & \text{if } e \in \mathcal{D} \\ \theta_{var}/|\text{variables}| & \text{if } e \in \mathcal{E} \end{cases}$
    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$'(\mathcal{D}, \theta, \mathcal{E}, \alpha_k)$
    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

Approximating the abstraction sleep objective

Following [37], we will approximate the abstraction sleep objective using the Akaike Information Criterion[4] [14], because the marginalization over θ is, in general, intractable:[5]

$$\arg\max_{\mathcal{D}} \left(\underbrace{\log \mathrm{P}[\mathcal{D}]}_{\text{prior}} + \underbrace{\arg\max_{\theta} \left(\sum_{x \in X} \log \sum_{\rho \in \mathcal{B}_x} \mathrm{P}[x|\rho]\mathrm{P}[\rho|\mathcal{D},\theta] + \log \mathrm{P}[\theta|\mathcal{D}] \right)}_{\text{replace marginal with max}} \underbrace{-|\theta|_0}_{\text{AIC}} \right) \tag{5.4}$$

Continuing to follow [37] we define $\mathrm{P}\left[\mathcal{D}\right] \propto \exp\left(-\lambda \sum_{\rho \in \mathcal{D}} \text{size}(\rho)\right)$, where λ is a hyper-parameter and $\text{size}(\rho)$ is the size of ρ's syntax tree, while placing a symmetric Dirichlet prior over the weight vector θ to define $\mathrm{P}\left[\theta|\mathcal{D}\right]$.

Relaxing the abstraction sleep objective: Introducing refactoring

Equation 5.4 prefers libraries assigning high probability to programs found during waking. However, if we add a new expression e to our library, then, in general, our programs might not be written in terms of e — and so we must *refactor* the programs in terms of the new expression. For example, imagine we wanted to discover a new procedure for doubling numbers, after having found the programs `(cons (+ 9 9) nil)` and `(λ (x) (+ (car x) (car x)))`. As human programmers, we can look at these pieces of code and recognize that,

[4]The AIC might get a lot of flack here because it's obviously not an approximation to the marginal, it's a model selection criterion. Why not use the BIC, which is just as easy to calculate? Well, the BIC is sneaky, because on the surface it looks like it might not be doing something bogus, and someone famous like Laplace derived it. Plus the BIC has a logarithm in it, which is a prerequisite for any serious formula. So it can give you a false sense of security. But the BIC is just as ridiculous as the AIC. At least the AIC owns it.

[5]One could also use variational methods to approximate the marginal, or use nonparametric methods to attain a closed form expression for the joint over $(\mathcal{D}, \{\mathcal{B}_x\})$, as done in [81]. The AIC has the advantage of being exceedingly simple and eminently tractable, but it is possible in principle that these more sophisticated methods would yield better empirical results.

if we define a new procedure called `double`, defined as `(λ (x) (+ x x))`, then we can rewrite the original programs as `(cons (double 9) nil)` and `(λ (x) (double (car x)))`. This process is a kind of refactoring where a new subroutine is defined (`double`) and the old programs rewritten in terms of the new subroutine.

More formally, our aim during abstraction sleep is not necessarily to maximize the probability of programs found during waking that solve tasks. Rather, our goal is to maximize the probability of *any* program which solves a task, or more formally, to maximize the marginal probability of tasks given library, i.e. $\mathrm{P}[x|\mathcal{D},\theta]$. This observation licenses a restatement of our objective which marginalizes over not just the programs found during waking, but over any refactoring of those programs. In a technical sense, by refactoring, we mean any program which is equivalent up to β-reduction (i.e., function application/variable substitution [103]). We write $\rho \longrightarrow_\beta \rho'$ to mean that ρ rewrites to ρ' in one step of β-reduction, and write $\rho \longrightarrow_\beta^* \rho'$ for the transitive reflexive closure of $\rho \longrightarrow_\beta \rho'$. Using this notation, we further refine our objective to

$$\log \mathrm{P}[\mathcal{D}] + \arg\max_\theta \left(\sum_{x \in X} \log \sum_{\rho \in \mathcal{B}_x} \mathrm{P}[x|\rho] \max_{\rho' \longrightarrow_\beta^* \rho} \mathrm{P}[\rho'|\mathcal{D},\theta] + \log \mathrm{P}[\theta|\mathcal{D}] \right) - |\theta|_0 \quad (5.5)$$

But 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. 5-3 diagrams the model using this bounded variant of Equation 5.5 to discover 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 within 4 steps of β-reduction – a quantity that grows exponentially both as a function of program size and as a function of the bound on β-reduction steps. To resolve this exponential growth, the next section introduces a new

data structure for representing and manipulating the set of refactorings, combining ideas from version space algebras [77, 91, 105] and equivalence graphs [132], and we derive a dynamic program for its construction. This results in substantial efficiency gains: A version space with 10^6 nodes, calculated in minutes, can represent the 10^{14} refactorings in Fig. 5-3 that would otherwise take centuries to explicitly enumerate and search.

Efficiently representing and manipulating refactorings

We tame the combinatorial explosion associated with refactoring using machinery that draws primarily on the idea of a *version space*. In this thesis, 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. Our first goal is to define an operator over version spaces which calculates the set of n-step refactorings of a program ρ, e.g., the set of all programs ρ' where $\rho' \underbrace{\longrightarrow_\beta \rho'' \longrightarrow_\beta \cdots \longrightarrow_\beta}_{\leq n \text{ times}} \rho$. We will call this operator $I\beta_n$, because it inverts β-reduction n times. This section begins by gradually building up to a definition of $I\beta_n$, and we prove its correctness in Appendix A.1.

Formally, a version space is either:

- A deBuijn[6] index: written $\$i$, where i is a natural number
- A primitive, such as the number `42`, or the function `map`
- An abstraction: written λv, where v is a version space

[6]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 [103] for more details. Sadly it is not standard in the literature to put a \$ in front of deBuijn indices, but it really should be: otherwise, it is unclear whether `0` means the number 0 or a deBuijn index. Languages like Perl and Bash use \$ in front of variables, and I suggest that we do the same for deBuijn indices.

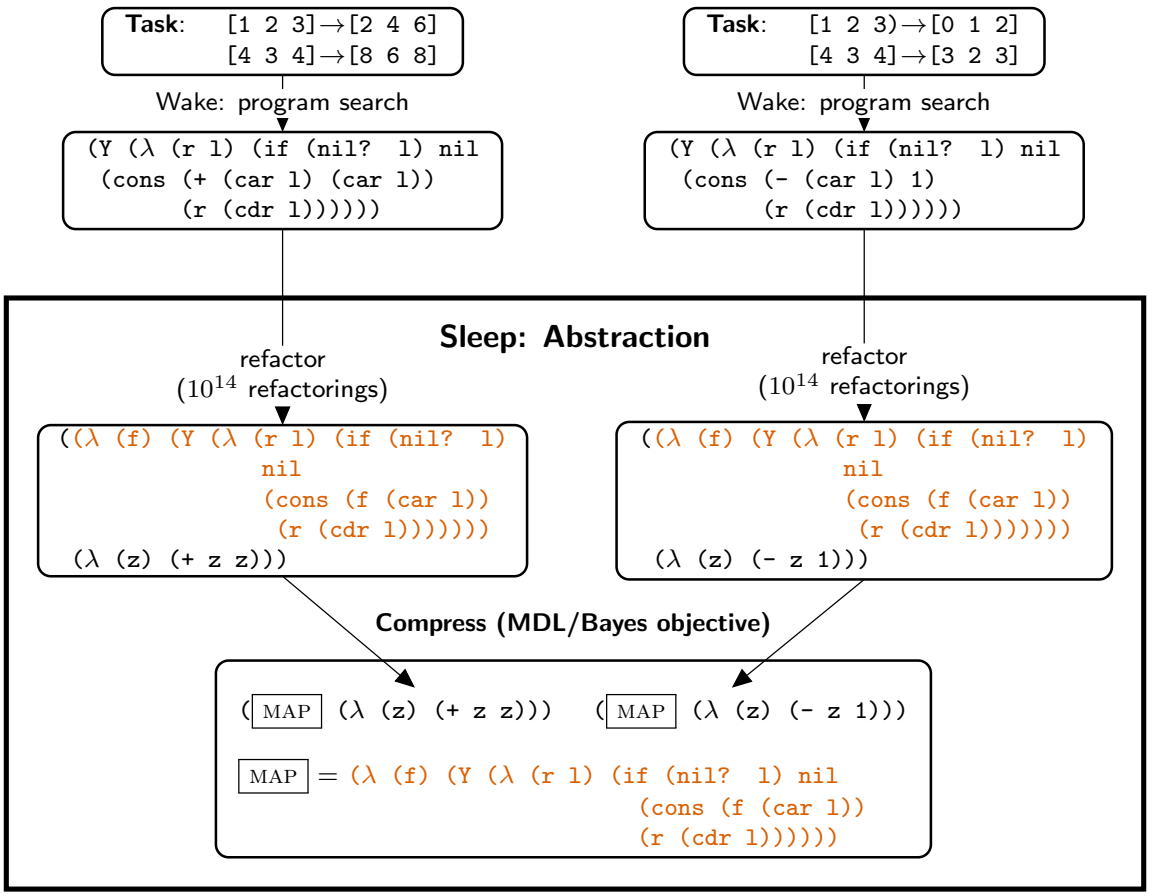


Figure 5-3: Programs found as solutions during waking are refactored – or rewritten in semantically equivalent but syntactically distinct forms – during the sleep abstraction phase, to expose candidate new primitives for growing DreamCoder's learned library. Here, solutions for two simple list tasks (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.

- 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

When we say a **leaf**, we mean either a primitive or a deBuijn index.

The purpose of a version space to compactly represent a set of programs. This compact encoding arises from the union operator ($\uplus$), which, intuitively, represents a nondeterministic choice between a collection of alternatives. For example, the version space $(\lambda \uplus \{\$0, +\})(\uplus \{4, 9\})$ encodes four different expressions: $(\lambda\$0)4$, $(\lambda\$0)9$, $(\lambda+)4$, and $(\lambda+)9$. We refer to the set of expressions encoded by a version space as its **extension**:

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

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

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

EXTRACT recursively as:

$$\textsc{extract}(v|\mathcal{D}) = \begin{cases} e, \text{ if } e \in \mathcal{D} \text{ and } e \in [\![v]\!] \\ \textsc{extract}'(v|\mathcal{D}), \text{ otherwise.} \end{cases} \tag{5.6}$$

$$\textsc{extract}'(e|\mathcal{D}) = e, \text{ if } e \text{ is a leaf} \tag{5.7}$$

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

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

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

Equation 5.6 simply checks whether a library component e is in the extension of the version space v, and if so just returns that component. This is correct because we seek the smallest member of the version space's extension, and members of the library count as having size 1. This is efficient because $e \in [\![v]\!]$ can be efficiently computed. Otherwise, we recurse along the structure of v via Equations 5.7.

Having now defined the structure of version spaces and some basic operations upon them, we return to our objective of constructing version spaces encoding the set of all n-step refactorings. We write $I\beta_n(\rho)$ for a version space encoding n-step refactorings of program ρ. This operator should satisfy:

$$[\![I\beta_n(\rho)]\!] = \left\{ \rho' \ :\ \rho' \underbrace{\longrightarrow_\beta \rho'' \longrightarrow_\beta \cdots \longrightarrow_\beta}_{\leq n \text{ times}} \rho \right\} \tag{5.8}$$

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

$$I\beta_n(v) = \uplus \left\{ \underbrace{I\beta'(I\beta'(I\beta'(\cdots}_{i \text{ times}} v))) \ :\ 0 \leq i \leq n \right\} \tag{5.9}$$

In particular, we want to define $I\beta'$ so that its extension obeys

$$[\![I\beta(v)']\!] = \{\rho' \ : \ \forall \rho \in [\![v]\!] \text{ and } \forall \rho' \text{ where } \rho \longrightarrow_\beta \rho\} \tag{5.10}$$

and so capture exactly and only those programs which β-reduce in one step to a program in the extension of v. For now we will define $I\beta'$ and motivate its definition in intuitively, but in Appendix A we will prove that these definitions are *consistent* (Theorem 3) and *complete* (Theorem 4). *Consistency* means that only valid refactorings are output (every program in $[\![I\beta'(v)]\!]$ does actually β-reduced to a program in $[\![v]\!]$), while *completeness* means that every valid refactoring is output (every program which β-reduces to an expression in $[\![v]\!]$ is present in $[\![I\beta'(v)]\!]$). Consistency and completeness imply Eq. 5.10

Constructing $I\beta'$ is somewhat technical, and depends on careful consideration of the β-reduction operator we aim to invert. To motivate our construction of $I\beta'$ we define β-reduction using deBuijn indices, following [103]. The β-reduction operator is defined as the smallest relation satisfying the following conditions (Equation 5.11-5.12). Each of these conditions is written with a horizontal bar, and means that whenever the premise above the bar holds, the relation below the bar is also satisfied. (Eq. 5.12 has nothing above its bar, and so the relation below the bar always holds.)

$$\frac{f \longrightarrow_\beta f'}{f\ x \longrightarrow_\beta f'\ x} \quad \frac{x \longrightarrow_\beta x'}{f\ x \longrightarrow_\beta f\ x'} \quad \frac{b \longrightarrow_\beta b'}{\lambda b \longrightarrow_\beta \lambda b'} \tag{5.11}$$

$$\frac{}{(\lambda b)v \longrightarrow_\beta \uparrow_0^{-1} [\$0 \mapsto \uparrow_0^1 v]b} \tag{5.12}$$

Eq. 5.11 inductively defines three rules which together say that whenever a subexpression may be β-reduced then the whole expression also β-reduces. Eq. 5.12 says that whenever

the left-hand side of an application is a λ-abstraction then the application β-reduces by substituting the right-hand side of the application into the body of the λ-abstraction. This is notated $\uparrow_0^{-1}\ [\$0 \mapsto \uparrow_0^1 v]b$. This notation involves both a shifting operator ($\uparrow_k^n$), which adds n to all deBuijn indices $\geq k$, as well as a substitution operator, $[\$n \mapsto e]e'$, which substitutes all occurrences of $\$n$ within e' for e. These are defined as:

$$
\begin{aligned}
[\$n \mapsto e]\rho &= \rho, \text{ if } \rho \text{ is a primitive} & \uparrow_k^n \rho &= \rho, \text{ if } \rho \text{ is a primitive} \\
[\$n \mapsto e](f\ x) &= ([\$n \mapsto e]f)\ ([\$n \mapsto e]x) & \uparrow_k^n (f\ x) &= (\uparrow_k^n f)\ (\uparrow_k^n x) \\
[\$n \mapsto e](\lambda b) &= \lambda([\$(n+1) \mapsto \uparrow_0^1 e]b) & \uparrow_k^n \lambda b &= \lambda\ \uparrow_{k+1}^n b \\
[\$n \mapsto e]\$i &= \begin{cases} e & \text{if } n = i \\ \$i & \text{if } n \neq i \end{cases} & \uparrow_k^n \$i &= \begin{cases} i & \text{if } i < k \\ i+n & \text{if } i \geq k \end{cases}
\end{aligned}
\tag{5.13}
$$

Accordingly, we will define $I\beta'$ as an operator which builds both top-level redexes (inverting Eq. 5.12), and which also recurses to build redexes within the body of an expression (inverting Eq. 5.11). Below we define $I\beta'$ as the union of an operator S, which constructs these top-level substitutions, along with recursive invocations of $I\beta'$:

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

Intuitively: every expression which β-reduces in one step to a member of $[\![u]\!]$ either does so via a top-level redex (computed via $S(u)$) or by virtue of the fact that a subexpression

of a member of $[\![u]\!]$ may be rewritten as a redex (computed via the recursive cases above). Because we are inverting exactly one step of β-reduction, the recursive case for function applications ($u = (f\ x)$) considers only the cases where either the function or argument undergoes inverse β-reduction ($(I\beta'(f)\ x)$ and $(f\ I\beta'(x))$, respectively) but not both.

The low-level workhorse of the refactoring algorithm is the S operator, which builds version spaces of new β-substitutions semantically equivalent to the original program's subexpressions. It takes as input a version space u and then returns a version space whose extension contains programs of the form $(\lambda b)v$ where substituting the value v into the body v (i.e. performing β reduction) gives an expression in $[\![u]\!]$. Therefore, constructing S depends on the details of substitution with deBuijn indices, outlined in Eq. 5.13. Just as substitution tracks the deBuijn index being substituted, our inverse substitution operator S keeps track of the substituted index: this index starts out at 0 and is incremented with each λ-abstraction. We define S as S_k with $k = 0$, where k corresponds to this index counter:

$$S(v) = S_0(v)$$

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

where we have defined a new operator, $\downarrow$, whose purpose is to undo the action of $\uparrow$. In particular, this operator satisfies $\downarrow_c^n \uparrow_c^n e = e$, for all programs e and natural numbers n, c (Lemma 2 in Appendix A), and is defined as:

$$\begin{aligned}
\downarrow_c^k \$i &= \begin{cases} \$i, \text{ when } i < c \\ \$(i-k), \text{ when } i \geq c+k \\ \varnothing, \text{ when } c \leq i < c+k \end{cases} \\
\downarrow_c^k \lambda b &= \lambda \downarrow_{c+1}^k b \\
\downarrow_c^k (f\ x) &= (\downarrow_c^k f\ \downarrow_c^k x) \\
\downarrow_c^k \uplus V &= \uplus \left\{ \downarrow_c^k v \mid v \in V \right\} \\
\downarrow_c^k v &= v, \text{ when } v \text{ is a primitive or } \varnothing \text{ or } \Lambda
\end{aligned} \tag{5.14}$$

Implementation note. We have written these definitions recursively, but actually implement them using a dynamic program: we hash cons each version space, and only calculate the operators $I\beta_n$, $I\beta'$, and S_k once per each version space.

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 $I\beta_n(\rho)$, we obtain a version space containing expressions semantically equivalent to program ρ—and also we obtain $I\beta_n$ for any subexpressions of ρ. In order to allow these subexpressions to be refactored independently, we track the equivalences exposed by $I\beta_n$ and compile all these equivalencies together. For example, when refactoring `(* (+ 1 1) (+ 5 5))`, the $I\beta_1$ operator, which inverts 1 step of β-reduction, will identify that `(+ 1 1)` can be rewritten as `(double 1)` while `(+ 5 5)` can be rewritten as `(double 5)`, where `double` = `(lambda (x) (+ x x))`. However, $I\beta_1$ will not discover that `(* (+ 1 1) (+ 5 5))` can

be rewritten as `(* (double 1) (double 5))`, because this requires inverting 2 steps of β-reduction. Yet this rewrite is clearly licensed by the semantic equivalences exposed by $I\beta_1$.

Inspired by the equality saturation approach within program analysis [132], we track the equivalences exposed by $I\beta_n$, and finally return a single structure for each program compiling all of these equivalences. This allows us to, for example, refactor `(* (+ 1 1) (+ 5 5))` into `(* (double 1) (double 5))` without considering more than 1 step of β-reduction. Concretely, for each program ρ, we calculate a version space $I\beta(\rho)$ defined as

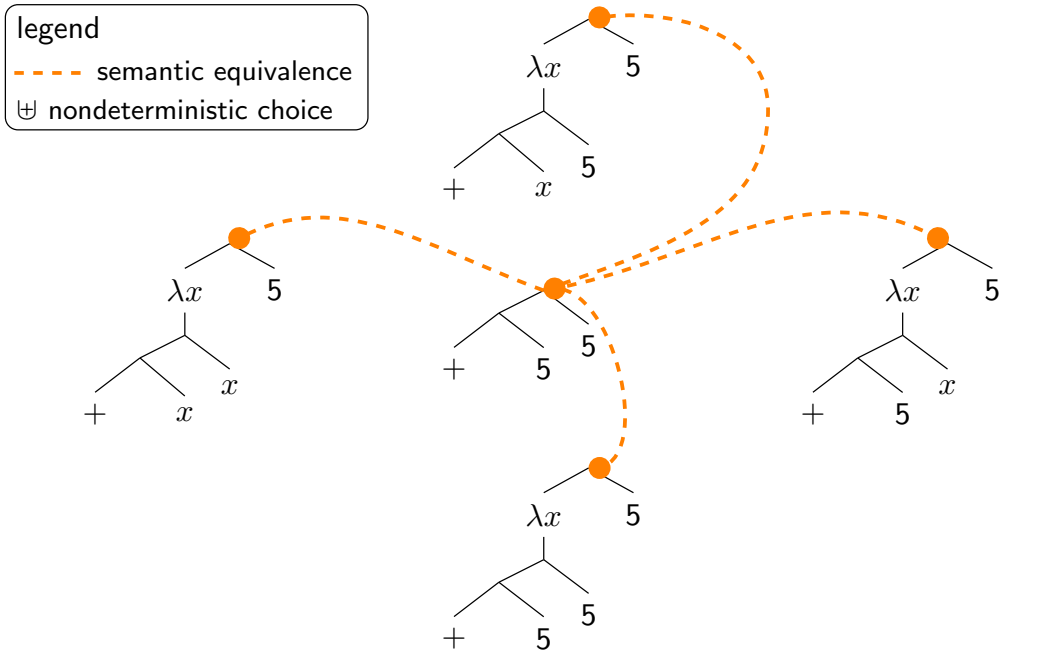
$$I\beta(\rho) = I\beta_n(\rho) \uplus \begin{cases} \text{if } \rho = (f\ x)\text{: } I\beta(f)\ I\beta(x) \\ \text{if } \rho = \lambda b\text{: } \lambda I\beta(b) \\ \text{if } \rho \text{ is an index or primitive: } \varnothing \end{cases}$$

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

Putting together the pieces

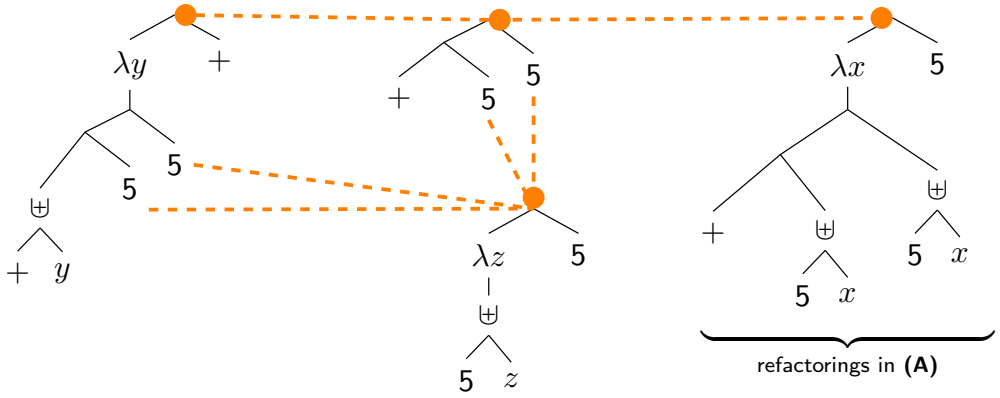
Algorithm 4 specifies our library learning procedure. This integrates two ideas: the machinery of version spaces and equivalence graphs, along with a probabilistic objective favoring compressive libraries. As described previously, the functions $I\beta(\cdot)$ and EXTRACT construct a version space from a program and extract the shortest program from a version space, respectively (Algorithm 4, lines 5-6, 14).

To appropriately score each proposed $\mathcal{D}$ 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



(A) Four refactorings of `(+ 5 5)` w/ common structure

(B) Those refactorings reexpressed w/ version space ⊎, plus more refactorings



Example program encoded by data structure in **(B)**:
((λ (y) (y 5 ((λ (z) z) 5))) +)

Figure 5-4: An illustration of refactoring with the cross-product version space operator ⊎ and equivalence tracking when inverting β-reduction by one step. We show four refactorings of a simple expression, `(+ 5 5)`, shown centered in **(A)**. Each of these refactorings is semantically equivalent to the original expression, notated with a dashed orange line. Each refactoring abstracts out the number `5` but differs in which of the two leaves it abstracts out. Because there are two leaves, and each can be either left as `5` or replaced with a variable, we have four possibilities. As shown in the rightmost tree in panel **(B)**, we can collapse all four of these refactorings into a single tree using the ⊎ operator, which can be thought equivalently as either a cross product or a nondeterministic choice. Panel **(B)** shows several other portions of the data structure (rightmost and bottom), but ellides others for clarity. At bottom we show a program encoded by the data structure in **(B)** which relies both on the semantic equivalencies and ⊎ operator.

Inside/Outside algorithm [71], our distribution over programs is context-sensitive due to the presence of variables in the programs and also due to the polymorphic typing system. Our earlier work [37] derives a tractable MAP estimator for θ under these conditions, and we follow this approach verbatim, and we summarize it in Appendix A.3.

Algorithm 4 Abstraction Sleep

Input: Set of beams $\{\mathcal{B}_x\}$
Output: Library $\mathcal{D}$, weight vector θ
$\mathcal{D} \leftarrow$ every primitive (leaf in syntax tree) in $\{\mathcal{B}_x\}$
while true **do**
    $\forall \rho \in \bigcup_x \mathcal{B}_x : v_\rho \leftarrow I\beta(\rho)$ ▷ Construct a version space for each program
    Define $L(\mathcal{D}', \theta) = \prod_x \sum_{\rho \in \mathcal{B}_x} \mathrm{P}[x|\rho]\mathrm{P}[\textsc{extract}(v_\rho|\mathcal{D}')|\mathcal{D}', \theta]$ ▷ Likelihood under $(\mathcal{D}', \theta)$
    Define $\theta^*(\mathcal{D}') = \arg\max_\theta \mathrm{P}[\theta|\mathcal{D}']L(\mathcal{D}', \theta)$ ▷ MAP estimate of θ
    Define $\text{score}(\mathcal{D}') = \log \mathrm{P}[\mathcal{D}'] + L(\mathcal{D}', \theta^*(\mathcal{D})) - |\theta^*(\mathcal{D})|_0$ ▷ objective function; Equation 5.4
    $\text{components} \leftarrow \{\textsc{extract}(v|\mathcal{D}) \ : \ \forall x, \forall \rho \in \mathcal{B}_x, \forall v \in \text{children}(v_\rho)\}$ ▷ Propose many new components
    $\text{proposals} \leftarrow \{\mathcal{D} \cup \{c\} \ : \ \forall c \in \text{components}\}$ ▷ Propose many new libraries
    $\mathcal{D}' \leftarrow \arg\max_{\mathcal{D}' \in \text{proposals}} \text{score}(\mathcal{D}')$ ▷ Get highest scoring new library
    if $\text{score}(\mathcal{D}') < \text{score}(\mathcal{D})$ **return** $\mathcal{D}, \theta^*(\mathcal{D})$ ▷ No changes to library led to a better score
    $\mathcal{D} \leftarrow \mathcal{D}'$ ▷ Found better library. Update library.
    $\forall x \ : \ \mathcal{B}_x \leftarrow \{\textsc{extract}(v_\rho|\mathcal{D}) \ : \ \rho \in \mathcal{B}_x\}$ ▷ Refactor beams in terms of new library
end while

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 $S_k(f\ x)$),[7] 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 5-5). 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 $\mathcal{D}$ to a shortest program in $[\![v]\!]$ using primitives in $\mathcal{D}$. 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 EXTRACT. The following recursive equations define how we calculate these beams; the set 'proposals' is

[7] 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.

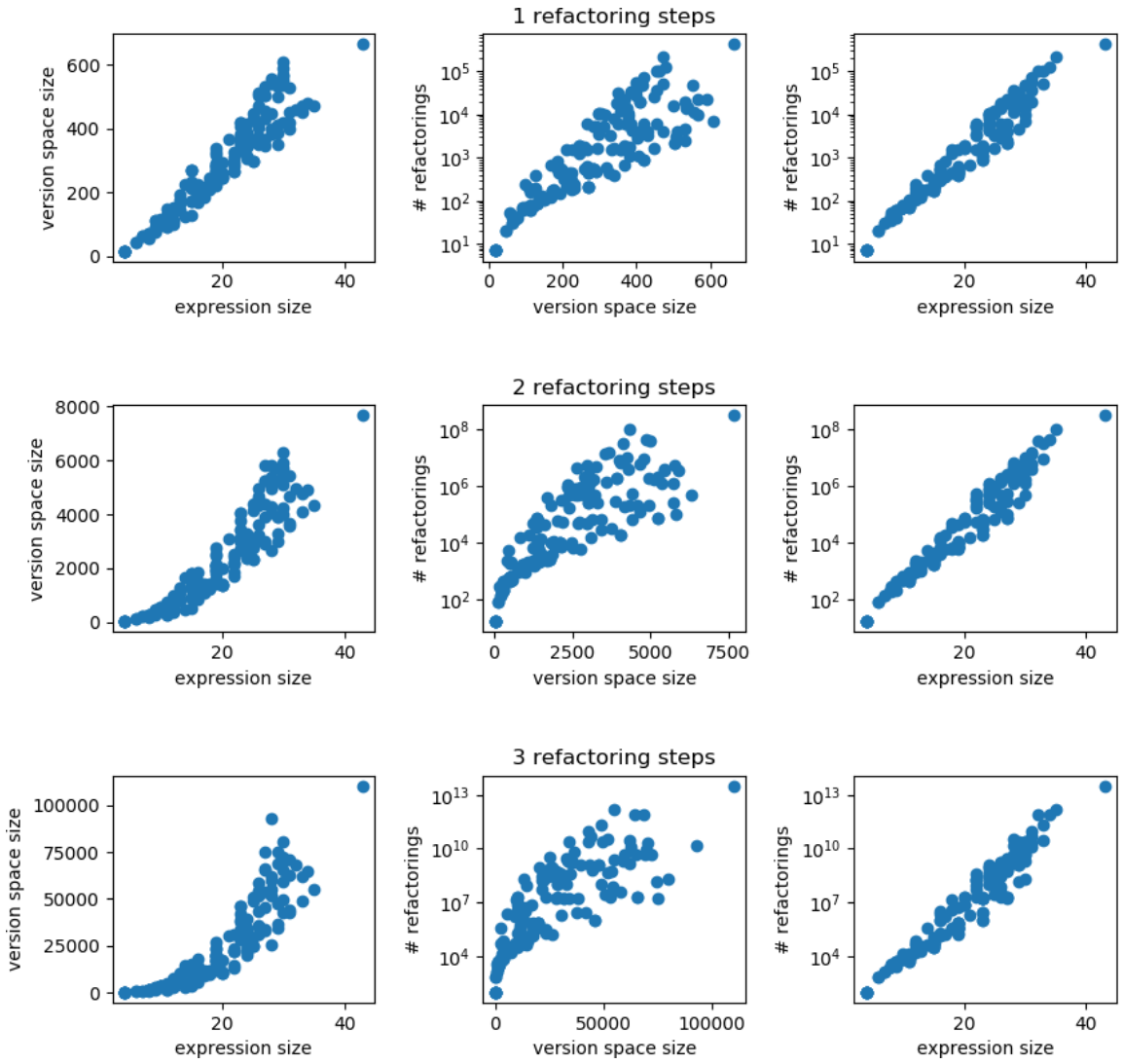


Figure 5-5: Library learning involves constructing a version space of refactorings of each program. Time and space complexity is proportional to # version spaces, which grows polynomialy with program size and exponentially on the bound on the number of refactoring steps ($\text{size}^{\text{bound}}$). 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.

defined in line 10 of Algorithm 4, and $\mathcal{D}$ is the current library:

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

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

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

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 $-\sum_x \min_{p \in \mathcal{B}_x} \text{beam}_{v_p}(\mathcal{D}')$. 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.

5.1.3 Dream Sleep: Training a Neural Recognition Model

During dreaming the system trains its recognition model, which later speeds up problem-solving during waking by guiding program search. 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 [61] 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.

To sharpen up this distinction between our dream phase and a conventional wake/sleep algorithm, we can define a pair of alternative objectives for the recognition model, $\mathcal{L}^{\text{posterior}}$ and $\mathcal{L}^{\text{MAP}}$, which (respectively) train Q to perform full posterior inference (as in classic wake/sleep), or MAP inference (as in DreamCoder). These objectives combine replays and

fantasies:

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

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

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

Evaluating $\mathcal{L}_{\text{Fantasy}}$ involves drawing programs from the current library, running them to get their outputs, and then training Q to regress from the input/outputs to the program. Since these programs map inputs to outputs, we need to sample the inputs as well. Our solution is to sample the inputs from the empirical observed distribution of inputs in X. The $\mathcal{L}^{\text{MAP}}_{\text{Fantasy}}$ objective also involves finding the MAP program solving a task drawn from the library. To make this tractable, rather than *sample* programs as training data for $\mathcal{L}^{\text{MAP}}_{\text{Fantasy}}$, we *enumerate* programs in decreasing order of their prior probability, tracking, for each dreamed task x, the set of enumerated programs maximizing $\mathrm{P}[x, p|\mathcal{D}, \theta]$. Algorithm 5 precisely specifies this process.

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

function generateDreams($\mathcal{D}, \theta, X, T$)
Input: Prior over programs $(\mathcal{D}, \theta)$, 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($\text{P}\left[\cdot|\mathcal{D}, \theta\right], T, \text{CPUs} = 1$) **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] : $\text{P}[\rho|\mathcal{D}, \theta] > \text{P}[\rho'|\mathcal{D}, \theta]$ **then** ▷ Dream supersedes old ones?
        observedBehaviors[b] $\leftarrow \{\rho\}$ ▷ Record new MDL program for task
    else if $\forall \rho' \in$ observedBehaviors [b] : $\text{P}[\rho|\mathcal{D}, \theta] = \text{P}[\rho'|\mathcal{D}, \theta]$ **then** ▷ Dream is equivalent MDL solution
        observedBehaviors[b] $\leftarrow \{\rho\} \cup$ observedBehaviors [b]
    else if $\forall \rho' \in$ observedBehaviors [b] : $\text{P}[\rho|\mathcal{D}, \theta] < \text{P}[\rho'|\mathcal{D}, \theta]$ **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

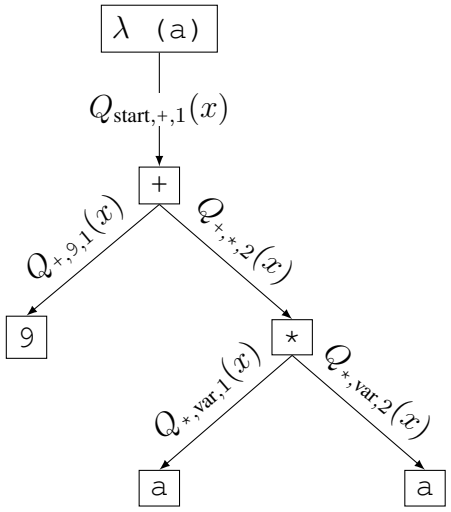
We chose to have DreamCoder maximize $\mathcal{L}^{\text{MAP}}$ rather than $\mathcal{L}^{\text{posterior}}$ for two reasons: $\mathcal{L}^{\text{MAP}}$ prioritizes the shortest program solving a task, thus more strongly accelerating enumerative search during waking; and, combined with our parameterization 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 5-6 (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 [10, 37, 89] – thus we can fall back on fast enumeration if the target program is unlike the training programs; (2) the recognition model provides fine-grained information about the structure of the target program, similar to [33, 150]; 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. 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[8] parameterization. Figure 5-6 (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.

[8]In the unigram variant Q predicts a $|\mathcal{D}| + 1$-dimensional vector: $Q(\rho|x) = \text{P}[\rho|\mathcal{D}, \theta_i = Q_i(x)]$, and was used in EC^2 [37]



	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 5-6: **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`.

In formal terms, Q predicts a $(|\mathcal{D}| + 2) \times (|\mathcal{D}| + 1) \times A$-dimensional tensor, where A is the maximum arity[9] of any primitive in the library. Slightly abusing notation, we write this tensor as $Q_{ijk}(x)$, where x is a task, $i \in \mathcal{D} \cup \{\text{start}, \text{var}\}$, $j \in \mathcal{D} \cup \{\text{var}\}$, 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 $Q(\cdot|X)$.

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

Algorithm 6 Drawing from distribution over programs predicted by recognition model. Compare w/ Algorithm 3

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

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

In the appendix we prove that any global optimizer of $\mathcal{L}^{\text{MAP}}$ breaks symmetries (Theorem 6), and give a concrete worked out example contrasting the symmetry-breaking behavior of $\mathcal{L}^{\text{MAP}}$ and $\mathcal{L}^{\text{posterior}}$ (Appendix A.2).

While the discussion of the recognition model has so far been independent of the problem domain, we can take advantage of the domain's structure when designing the architecture of the neural network. We implement recognition models via a domain-specific encoder which inputs a task, followed by a domain-general architecture which outputs the Q_{ijk} bigram transition matrix. For example, for domains with sequential structure (list processing, text editing, regular expressions) the recognition model's initial task encoding is produced by a recurrent neural network. We use a bidirectional GRU [24] 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, and MaxPool the final hidden unit activations in the GRU along both passes of the bidirectional GRU. For domains with 2D visual structure (LOGO graphics, tower building, and symbolic regression) we use a convolutional neural network. We take our convolutional architecture from [123]. We follow the RNN/CNN by an MLP with 128 hidden units and a ReLU activation which then outputs the Q_{ijk} matrix. See Figure 5-7. We train our recognition models using Adam [69] with a learning rate of 0.001 for 5×10^3 gradient steps or 1 hour, whichever comes first.

5.2 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,

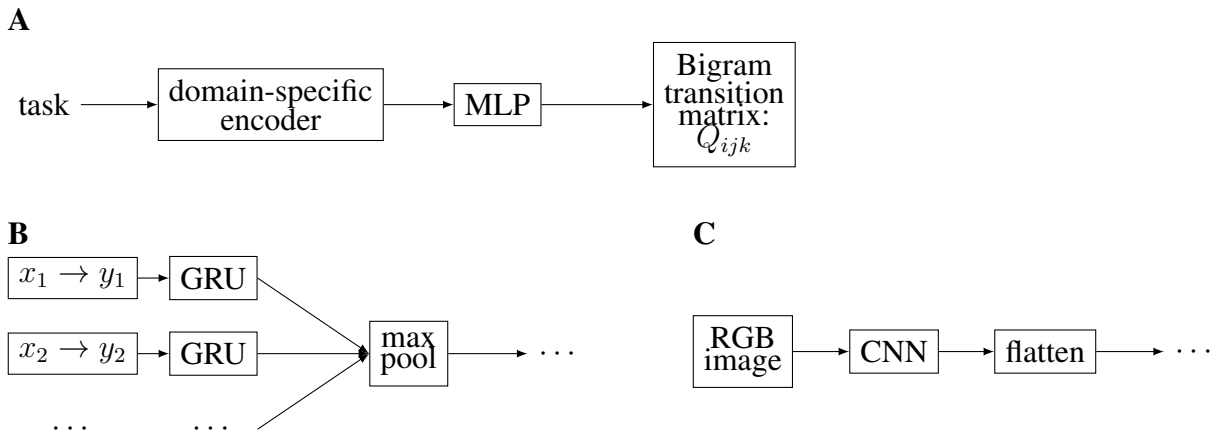


Figure 5-7: Schematic overview of neural network architectures used by DreamCoder. **(A)** For each domain, the recognition model inputs a task, converts it to a feature vector using a domain-specific encoder, and then transforms this vector into the bigram transition matrix using an MLP with a single hidden layer having 128 units and a ReLU activation. **(B)** Domain-specific encoder for domains were the task input is a set of input/outputs (list processing, text editing, regexes (empty input)). We concatenate each input/output pair with a special delimiter symbol ($\to$) and pass each to an identical bidirectional GRU with 64 hidden units. The GRU outputs–concatenated from both directions–are maxpooled prior to being input to the downstream MLP. **(C)** Domain-specific encoder for domains where the task input is an image (logo graphics, tower building, and symbolic regression). We transform the input image using a convolutional network whose architecture is identical to that used in prototypical networks [123], before flattening the CNN output to be passed to the MLP.

including routines like `map`, `fold`, `cons`, `car`, `cdr`, etc. Our list processing tasks comprise 218 problems taken from [37], split 50/50 test/train, each with 15 input/output examples. In solving these problems, DreamCoder composed around 20 new library routines, and rediscovered higher-order functions such as `filter`. Each round of abstraction built on concepts discovered in earlier sleep cycles — for example the model first learns `filter`, then uses it to learn to take the maximum element of a list, then uses that routine to learn a new library routine for extracting the n^{th} largest element of a list, which it finally uses to sort lists of numbers (Fig. 1B).

Synthesizing programs that edit text is a classic problem in the programming languages and AI literatures [77], and algorithms that synthesize text editing programs ship in Microsoft Excel [55]. 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 [3] program synthesis competition.[10] 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

[10]We compare with the 2017 benchmarks because 2018 onward introduced non-string manipulation problems; custom string solvers such as FlashFill [55] and the latest custom SyGuS solvers are at ceiling for these newest problems.

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 [16], to handwritten characters [73, 62], to Raven's progressive matrices [109] — 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 [134], tasking our model with drawing a corpus of 160 images (split 50/50 test/train; Fig. 5-8A) while equipping it with control over a 'pen', along with imperative control flow, and arithmetic operations on angles and distances. After training DreamCoder for 20 wake/sleep cycles, we inspected the learned library and found interpretable parametric drawing routines corresponding to the families of visual objects in its training data, like polygons, circles, and spirals (Fig. 5-8B) – 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. 5-8C).

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. 5-8D), offering limited value for training the recognition model. After learning, the system's dreams are richly structured (Fig. 5-8E), 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 [136].

Inspired by the classic AI 'copy demo' – where an agent looks at a tower made of toy blocks then re-creates it [144] – we next gave DreamCoder 107 tower 'copy tasks' (split

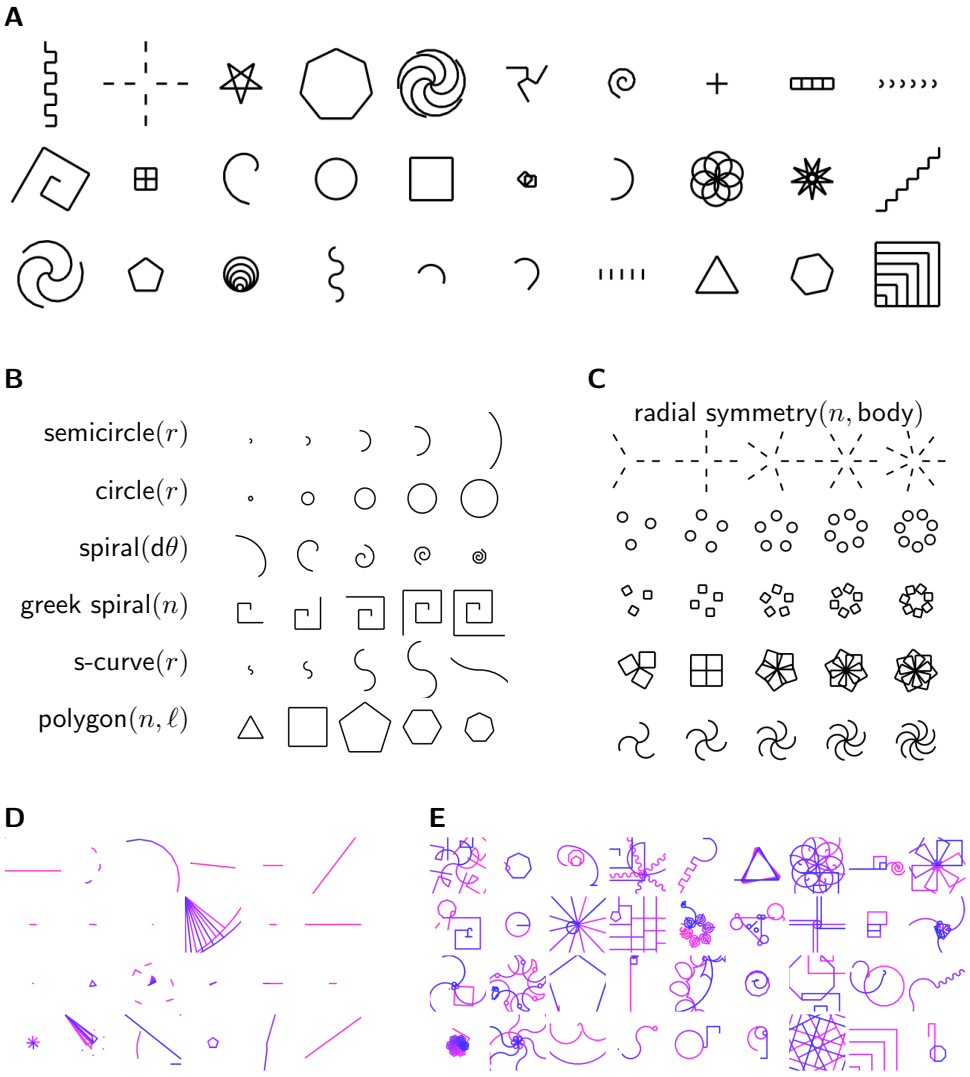


Figure 5-8: **(A):** 30 (out of 160) LOGO graphics tasks. The model writes programs controlling a 'pen' that draws the target picture. **(B-C):** Example learned library routines include both parametric routines for drawing families of curves (**B**) as well as primitives that take entire programs as 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.

50/50 test/train, Fig. 5-9A), 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' [131] for building blocks towers (Fig. 5-9B), including concepts like arches, staircases, and bridges, which one also sees in the model's dreams (Fig. 5-9C,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 [86], to learning routines for symbols and signs [73], to learning new motor routines for producing words [72]. 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 [60], 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. 5-10). It can explain many real-world text patterns and use its explanations as a probabilistic generative model to imagine new examples of these concepts. For instance, 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 trajecto-

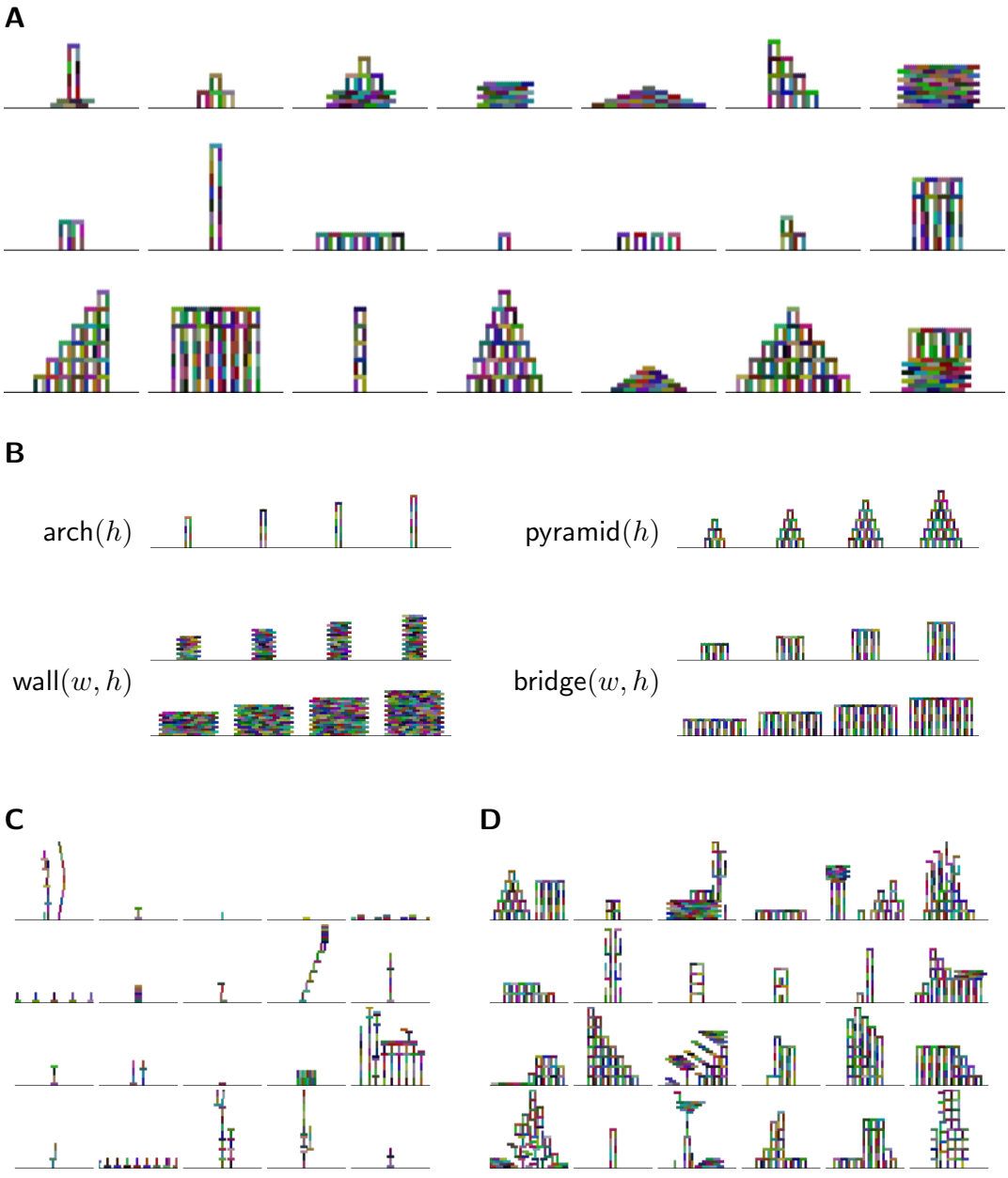


Figure 5-9: **(A)** 21 (out of 107) tower building tasks. The model writes a program controlling a 'hand' that builds the target tower. **(B)** Four learned library routines. These components act like parametric options [131], giving human-understandable, higher-level building blocks that the agent can 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.

Input		MAP program	Samples	Input		MAP program	Samples
(210) (220) (41) (635) (38)	Full	(dd(d)*)	(220) (461)	Y2015/1093 Y2013/1010 Y2014/1017 Y2015/1421 Y2017/1162	Full	Y201d/dddd	Y2010/3308 Y2010/1163
	No Library	(dd(d)*.	(14u (2040)		No Library	Y201d/dddd	Y2011/1131 Y2015/7116
	No Rec	(dd(d)*)	(68) (308)		No Rec	Y201(d)*d/(d)*	Y20127/20411 Y2011214/1
$5.70 $3.40 $2.80 $5.40 $3.70	Full	$d.d0	$2.40 $3.30	-00:16:05.9 -00:19:52.9 -00:33:24.7 -00:44:02.3 -00:24:25.0	Full	-00:dd.dd.d	-00:93:53.2 -00:23=43.3
	No Library	$d.d0	$5=50 $7#40		No Library	-00:dd.dd.d	-00:16g22:5 -00:22.53\t2
	No Rec	$(d)*.(d)*0	$.0 $873.30		No Rec	(-00:)?(.)*dd.d	-00:i47.5 -00:r59.0
(715) 967-2697 (608) 819-2220 (920) 988-2524 (608) 442-0253 (262) 723-4043	Full	(ddd) ddd-dddd	(099) 242-2029 (948) 452-9842	L - ?? L - 31.0 lbs. L - 10.0 lbs. S - 8.6 lbs. L - 25.2 lbs.	Full	u - (dd.(d)*)*(.)*	L - 13.05 ssb\t L - 12.3 02.1 s
	No Library	.ddd) ddd.dddd	?773) 726-6866 m627) 674,0602		No Library	. - (d(.)*)*..	. - . tY - E5
	No Rec	.(d)*) (d)*dd.(d)*	z40192) 51(8 =2) 279-876273		No Rec	u - (d)*(.)*	L - 5208s. S - 5533..

Figure 5-10: 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)

ries (Fig. 1A, 'Symbolic Regression', and EC^2 [37]). 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) [14].[11] 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* [30], 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 [33] model on samples from the initial library; and *Enumeration*, which performs type-directed enumeration [44] for 24 hours per task, generating and testing up to 400 million programs for each task.

[11] Again eagle-eyed readers may notice an inconsistency: we use the BIC here but the AIC previously when deriving an objective for the library learner. The AIC is more appropriate when performing model selection, i.e. when a single model is desired. The BIC is more appropriate when approximating a marginal probability. Here because we represent an approximate posterior over program solving each task, we actually want the BIC because we aren't picking a single program, but instead finding a weighted set of candidate programs. These weights should approximate the posterior probability, and correct calculation of the probability would involve marginalizing out the continuous parameters, hence the BIC. Thank you Roger Levy for this observation and for not making me remove these footnotes!

Across domains, our model always solves the most held-out tasks (Fig. 5-11A) 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 while DreamCoder learns to solve nearly 100% of them. The time needed to train DreamCoder to the points of convergence shown in Fig. 5-11A 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. 5-11B). 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

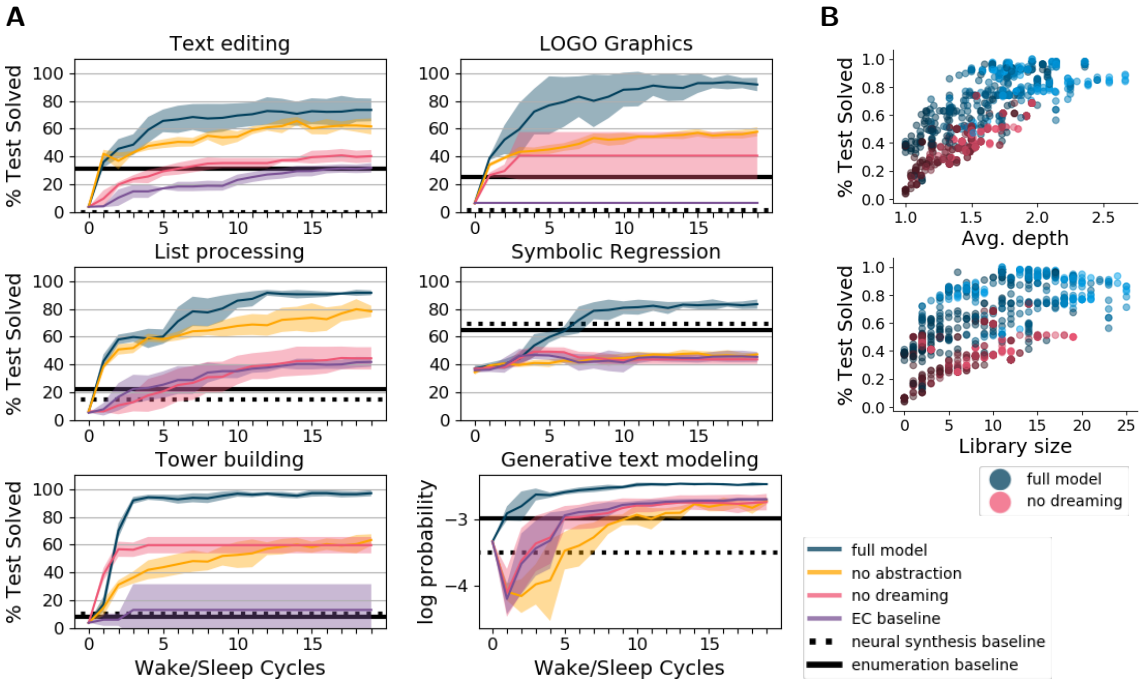


Figure 5-11: Quantitative comparisons of DreamCoder performance with ablations and baseline program induction methods. **(A)** Held-out test set accuracy, across 20 iterations of wake/sleep learning for six domains. Generative text modeling plots show posterior predictive likelihood of held-out strings 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.

network's activation space and the similarity of code components used to solve these problems (see Fig. 5-20; $p < 10^{-4}$ 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. 5-19) – analogous to how human experts learn to classify problems by the underlying principles that govern their solution rather than superficial similarities [22, 23].

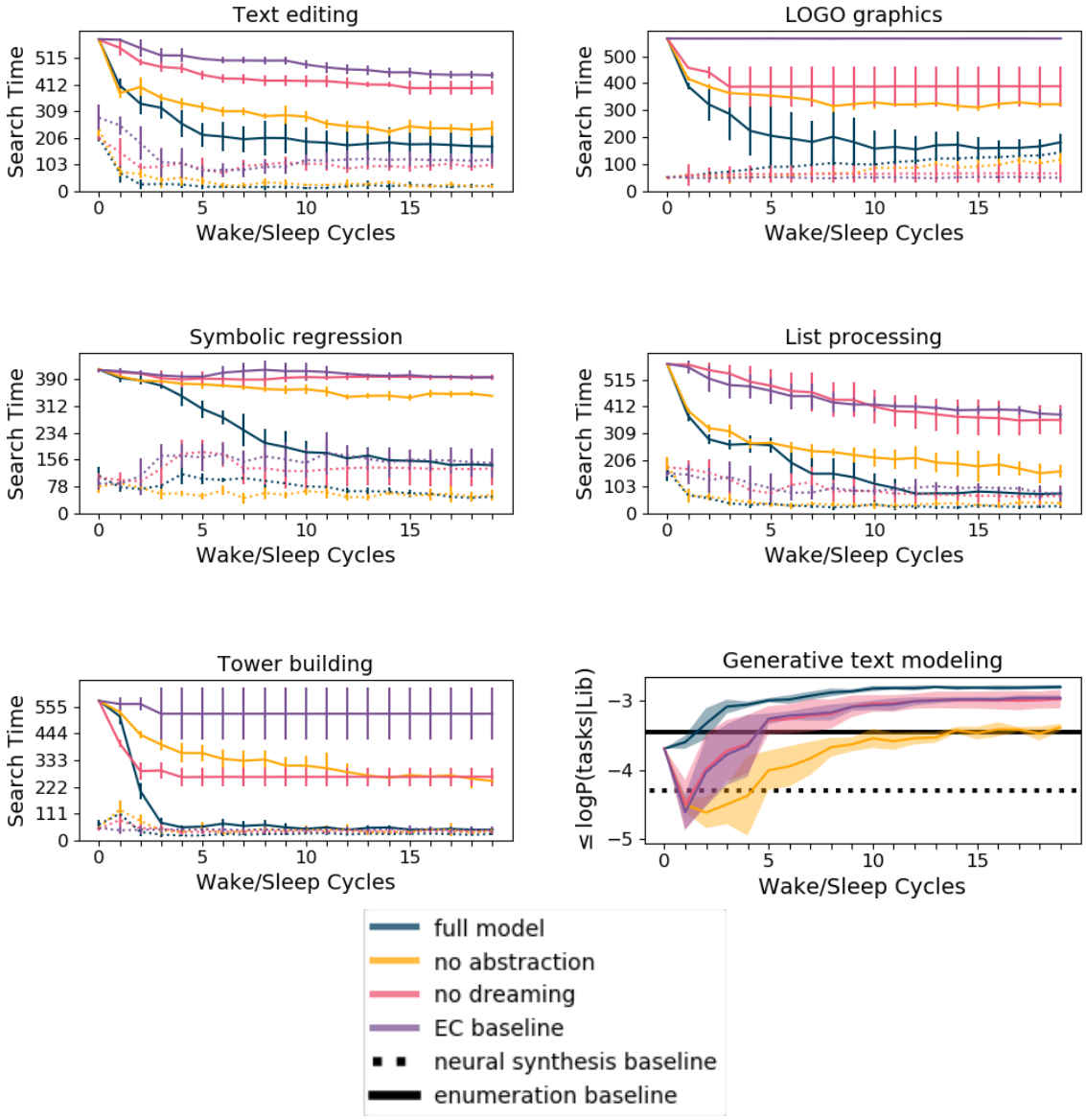


Figure 5-12: Test set performance across wake/sleep iterations. Error bars over five random seeds. Teal: Full model. Yellow: Dreaming only (no library learning). Pink: Abstraction only (no recognition model). Purple: EC baseline. Search time plots show solid lines (time averaged over all tasks) and dotted 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).

List processing learned libraries

Figure 5-13: Learned libraries for list processing diagrammed as graphs. Lines (left-or-right) go from a learned concept to all other concepts that use it. We show four libraries from four different runs, showing the variability in learned library structure.

Tower building learned libraries

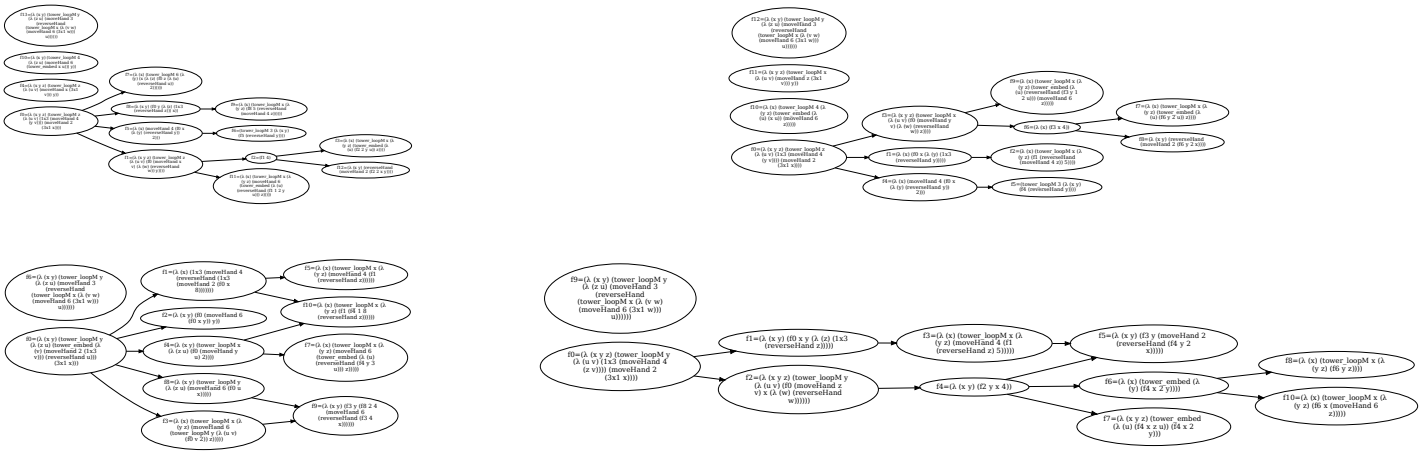


Figure 5-14: Learned libraries for tower building diagrammed as graphs. Lines (left-or-right) go from a learned concept to all other concepts that use it. We show four libraries from four different runs, showing the variability in learned library structure.

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 [120, 75, 117], we first task DreamCoder with learning equations describing 60 different physical

Text editing learned libraries

Figure 5-15: Learned libraries for text editing diagrammed as graphs. Lines (left-or-right) go from a learned concept to all other concepts that use it. We show four libraries from four different runs, showing the variability in learned library structure.

Symbolic regression learned libraries

Figure 5-16: Learned libraries for symbolic regression diagrammed as graphs. Lines (left-or-right) go from a learned concept to all other concepts that use it. We show four libraries from four different runs, showing the variability in learned library structure.

Generative regex learned library

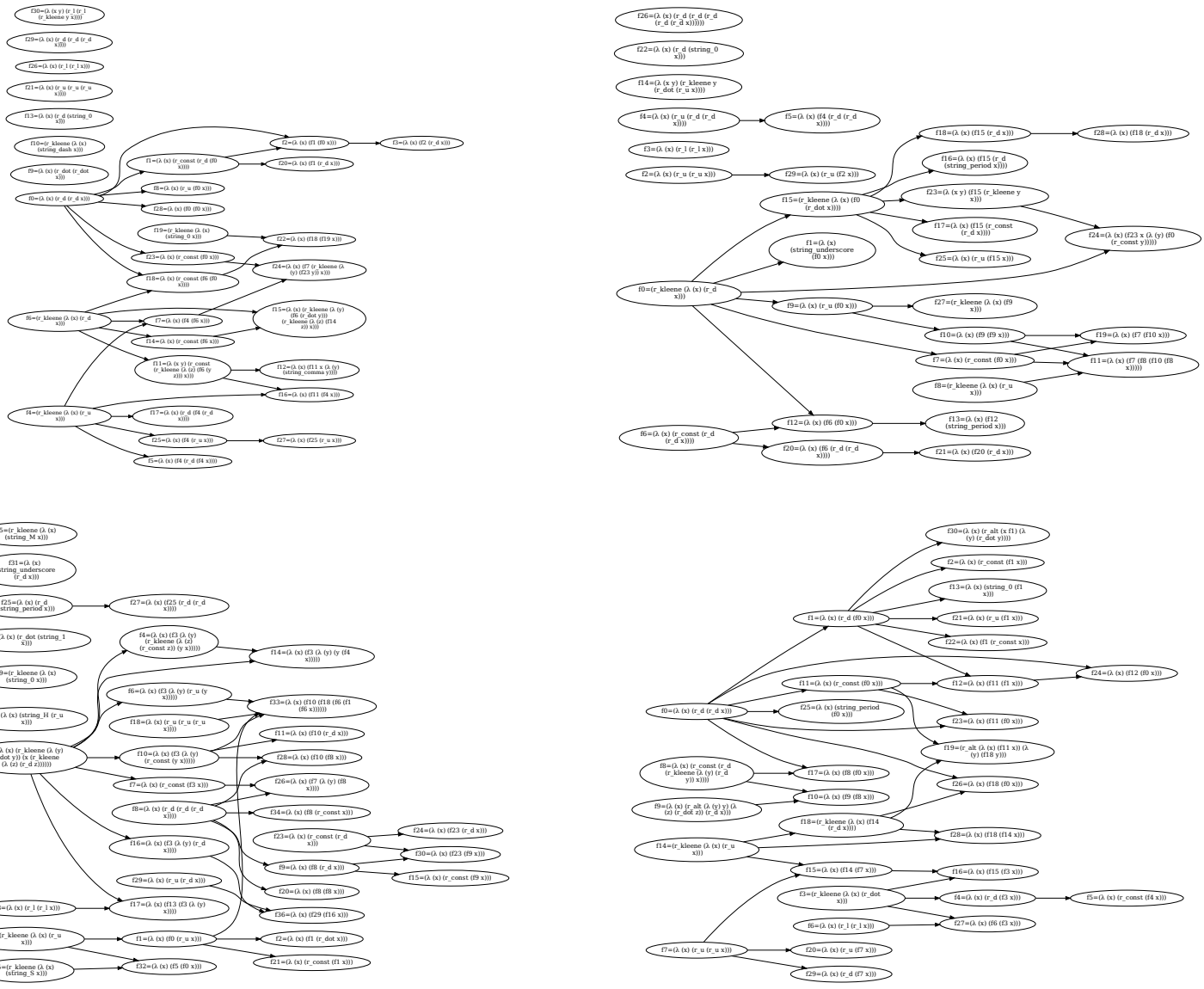


Figure 5-17: Learned libraries for generative regular expressions diagrammed as graphs. Lines (left-or-right) go from a learned concept to all other concepts that use it. We show four libraries from four different runs, showing the variability in learned library structure.

Figure 5-18: Learned libraries for logo graphics diagrammed as graphs. Lines (left-or-right) go from a learned concept to all other concepts that use it. We show four libraries from four different runs, showing the variability in learned library structure.

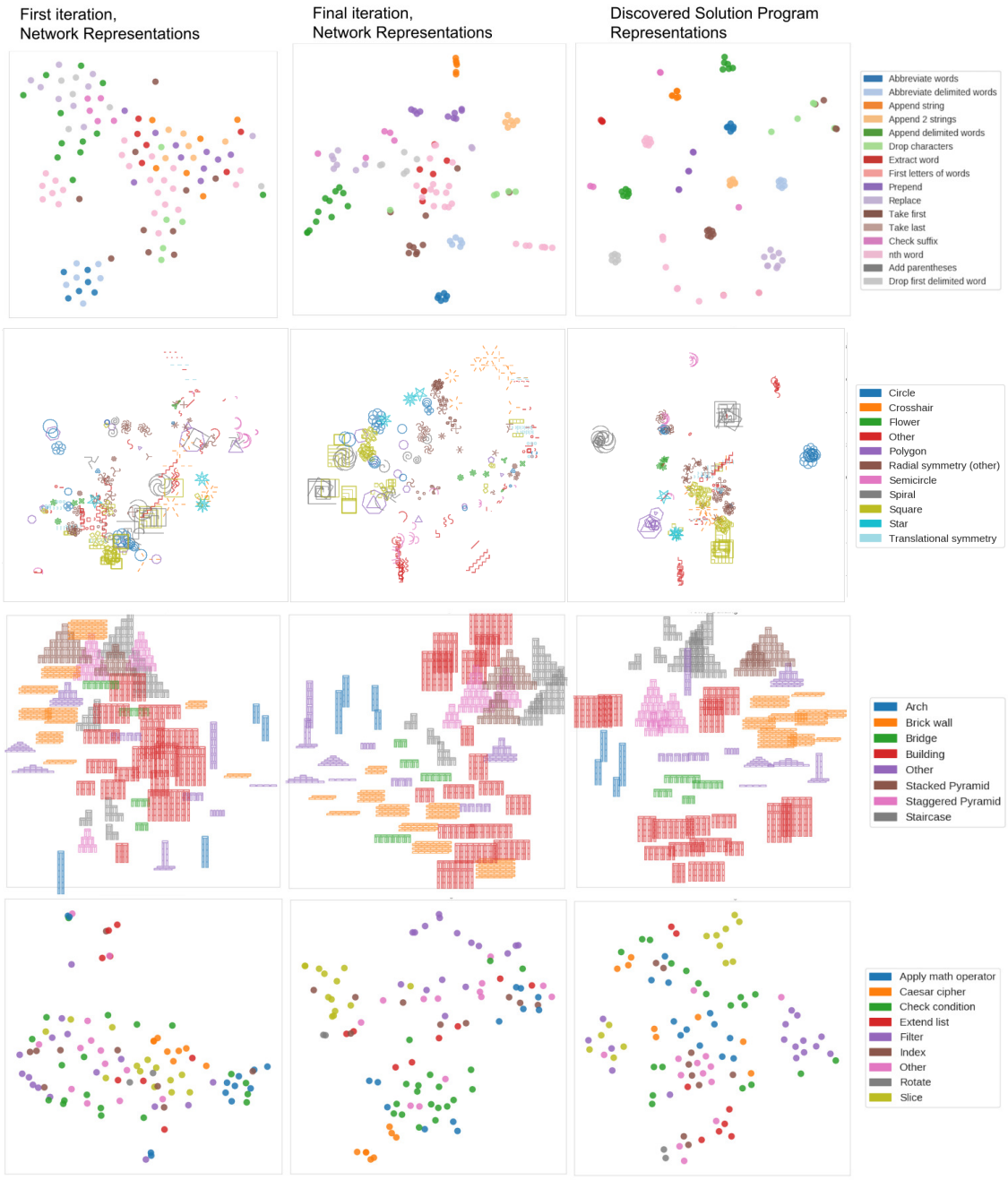


Figure 5-19: Left: TSNE embedding of model's 'at-a-glance' (before searching for solution) organization of tasks (neural net activations), after first wake/sleep cycle. Middle: at-a-glance organization after last wake/sleep cycle. Right: TSNE visualization of programs actually used to solve tasks; intuitively, 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.

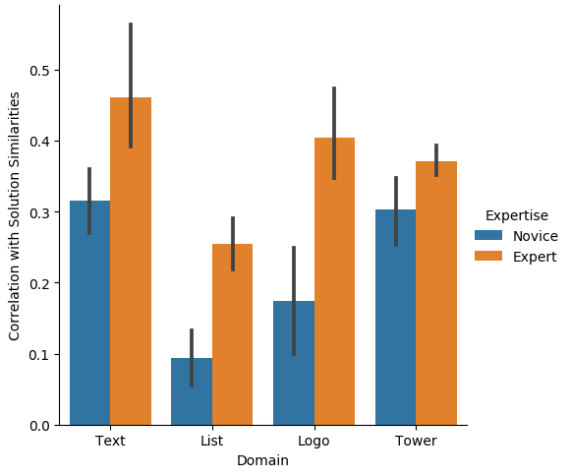


Figure 5-20: Correlation between similarity matrices of neural net activations and program solution feature vectors, both after the first wake/sleep cycle (Novice) and at the last cycle (Expert). Error bars represent SEM over 5 random seeds. Both correlations and similarities computed using cosine distance.

laws and mathematical identities taken from AP and MCAT physics "cheat sheets", based on numerical examples of data obeying each equation (Figure 5.1). 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. We minibatch only unsolved tasks during each wake cycle, owing to the small size of the training set, and the difficulty of the problems. Specifically, as the set of unsolved problems shrinks, we spend more CPU time per problem, because the CPU time per waking cycle is fixed but the batch size shrinks as the number of solved tasks increases. After 8 such 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. 5-21A). 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.

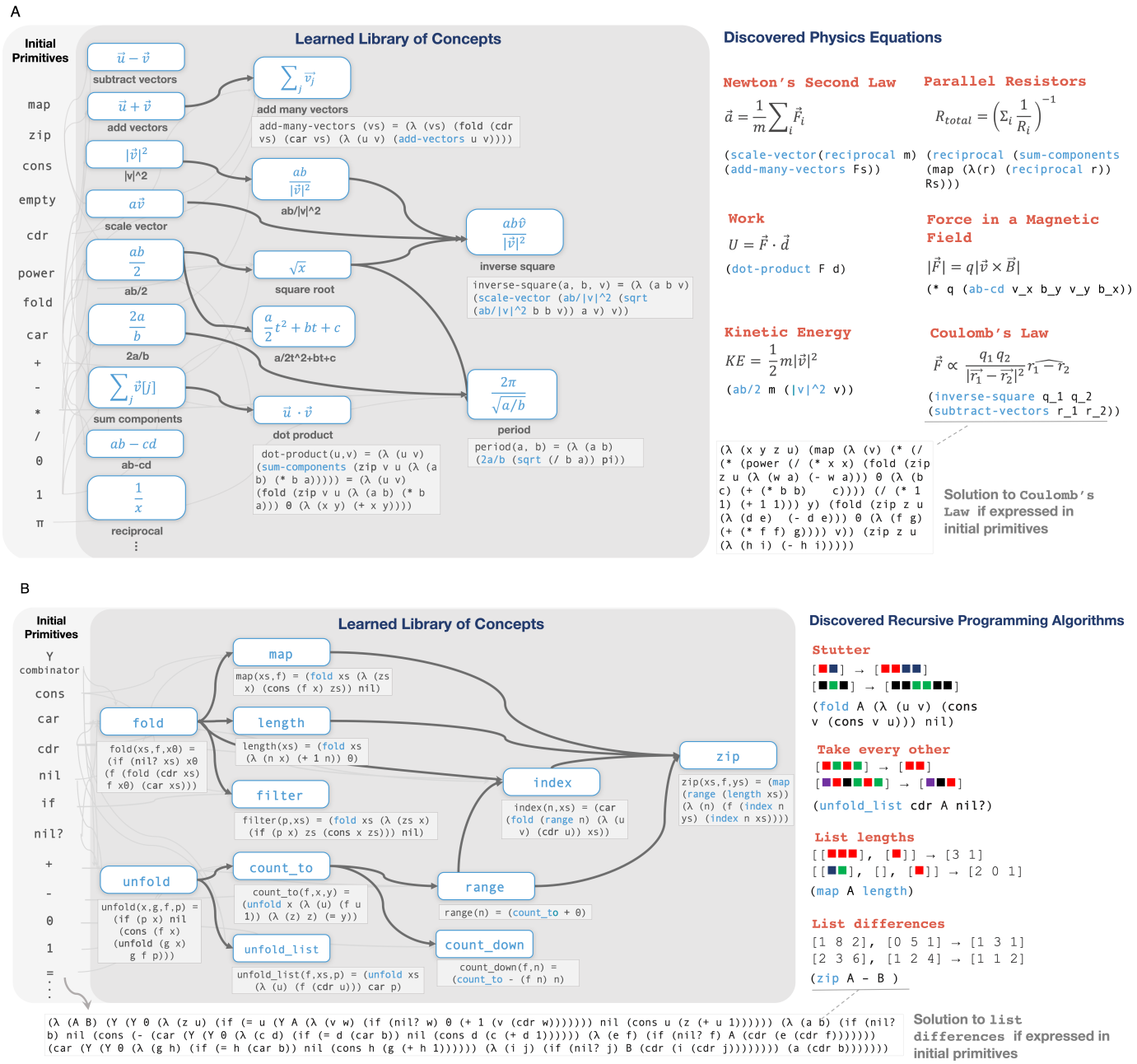


Figure 5-21: DreamCoder develops languages for physical laws (starting from recursive functions) and recursion patterns (starting from the Y-combinator, `cons`, `if`, etc.) **(A)** Learning a language for physical laws starting with recursive list routines such as `map` and `fold`. DreamCoder observes 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.

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

Could DreamCoder also learn this recursive sequence manipulation language? We initialized the system with a minimal subset of 1959 Lisp primitives (`if`, `=`, `>`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`, all present in some form in McCarthy's 1959 Lisp [88].[12]) and asked it to solve 20 basic programming tasks, like those used in introductory computer science classes (Figure 5-22). Crucially the initial language also includes primitive recursion (the Y combinator), which in principle allows learning to express any recursive function, but no other recursive function is given to start; previously we had sequestered recursion within higher-order functions (`map`, `fold`, ...) given to the learner as primitives. We did not use the recognition model for this experiment: a bottom-up pattern recognizer is of little use for acquiring this abstract knowledge from less than two dozen problems.

With enough compute time (roughly five days on 64 CPUs), DreamCoder learns to solve all 20 problems, and in so doing assembles a library equivalent to the modern repertoire of functional programming idioms, including `map`, `fold`, `zip`, `length`, and arithmetic operations such as building lists of natural numbers between an interval (see Fig. 5-21B). 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" [51]. 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.

[12] 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.

Programs & Tasks

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

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

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

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

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

drop last element
```
[2 1 4]→[2 1]
[9 8]→[9]
f(ℓ)=(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(ℓ)=(fold + ℓ 0)
```

double elements
```
[4 2 6 4]→[8 4 12 8]
[2 3 0 7]→[4 6 0 14]
f(ℓ)=(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(ℓ)=(map (- 0) ℓ)
```

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

take every other
```
[1 5 2 9]→[1 2]
[3 8 1 3 1 2]→[3 1 1]
f(ℓ)=(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 5-22: 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 5-21B.

5.3 Lessons

This concluding chapter suggests 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 performance hinges on learning explicit declarative knowledge– a prior implemented by a language bias–together with the implicit procedural skill to use it, i.e. an inference strategy. 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.

5.3.1 Connections to 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.

5.3.2 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 [137], 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 [126, 127, 116, 63, 34]. 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 [74, 128, 19], 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, we have advocated approaches that share similar philosophical motivations to the sketching approach to program synthesis [124]. 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.

Acknowledgments

This chapter contains material taken from "Library learning for neurally-guided Bayesian program induction" (NeurIPS 2018, by Kevin Ellis, Lucas Morales, Armando Solar-Lezama, Joshua Tenenbaum), as well as material taken from "DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning" (arXiv 2020, by Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua Tenenbaum). The dissertation author was the primary researcher on this work.

Chapter 6

Open directions

Here we take stock and look forward, considering the weaknesses and strengths of the work given in this thesis, and what new directions open up as a result. Our intention is to lay down research directions at the intersection of program synthesis and artificial intelligence, and connect those directions to the work here.

Neurosymbolic Program Induction: Not everything's program-*like*. This thesis focuses on domains where the solution space is largely captured by crisp symbolic forms. This includes classic program synthesis areas as well as domains where the task inputs are pixel images, voxel arrays, text patterns containing exceptions and irregularities, and search spaces containing continuous parameters. Nonetheless, much real-world data is far messier than considered here. Within vision, considered modeling visual textures, or organic shapes; within natural language semantics, consider modeling the meaning of a noun such as "dog" or slippery concepts such as "dog walking" [90][1], or the loose association between nostrils and a word starting with "s" (snore, smell, snort, snout, ...); or when

[1] "Dog walking" is the prototypical example of a slippery concept [90]. There are many unintuitive yet undeniable examples of dog walking, such as a person on a Segway leading a dog with a treat. Here there is neither any walking nor any leashes involved, but it is definitely an example of dog walking. I thank Melanie Mitchell for this family of examples.

modeling real-world complicated physical systems, the ever-present residuals which depart from the clean equations we might write down to model that system.

Aspects such as these are probably not well modeled using only a program-*like* representation. Instead, opaque statistical approaches might do better: we can use distributional semantics for the meaning of nouns, for example. Although not investigated in this thesis, we believe that neurosymbolic representation learning–which interleaves combinatorial structure with reusable neural components–can offer a productive path forward, as exemplified in the HOUDINI system [140], neural module networks [7], and others [84, 148, 54], which could be especially valuable when combined with the bootstrapping, library learning, and REPL-guided search techniques introduced here.

Further pushing the scalability of program induction. At its best, the algorithms introduced here can synthesize nontrivial, lengthy programs: a dozen lines of graphics code (Section 3.2.3), a set of six interacting phonological processes (Figure 4-5), or a sorting algorithm with 32 function calls (Figure 5-1B). Yet these demonstrations pale in complexity when compared to the size and sophistication of real-world code. If program induction is to evolve into a standard part of the AI toolkit, then the programs we will need to synthesize for AI tasks will likely rival the code written by human software engineers.

While we see the principles developed in this thesis as necessary for this scaling path, we suspect that library learning and REPL guidance are both insufficient and underdeveloped: these techniques must be pushed farther, and supplemented with other methods. A missing method, unexploited in this thesis, is *divide and conquer*, or recursively breaking a large problem down into smaller pieces and solving each piece independently. This method has proved important in recent program induction works [55, 138, 4], and could be especially synergistic with the recognition model techniques introduced here when used to propose ways of subdividing problem specifications.

The algorithms introduced here also suggest specific next steps. In particular, library

learning needs to confront the problem of not just synthesizing new functions, but also discovering new types and data structures. Synthesizing new types is important both because datatypes are a core component of real-world libraries, but also because it would allow our AI systems to grow out their own ontologies: uncovering the *what* of a domain, rather than just the *how*. REPL guidance techniques need to fully tap the pre-existing idea of self-play found within AlphaGo [119], ExIt [8], and TDGammon [133] by aligning their train and test time inference strategies. We also must generalize the REPL to accommodate intermediate program states which cannot be fully executed, which would allow a REPL system to induce programs with variables, control flow, and recursion. Two complemently approaches seem promising here: using neural networks to approximately represent these intermediate program states;[2] and exposing a debugger-like interface to the policy and value networks which reveals the intermediate execution trace as much as possible. Fully exploiting the insight of self-play for REPL synthesizers could be especially useful in conjunction with algorithms for generating good tasks, discussed next.

Beyond training sets: Bootstrapping by making your own problems. The conceptual cores of this thesis come together in the last chapter with the DreamCoder system. This system's knowledge grows gradually, with dynamics related to but different from earlier developmental proposals for "curriculum learning" [13] 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

[2]Maxwell Nye is actively exploring this direction—stay tuned!

generate their own problems in these human-like ways is a necessary next step. Indeed, one of the most distinctive features of human problem-solving is our ability to make up our own problems. In a sense, almost all of artificial intelligence research consists of making up our own problems: Fully tackling the challenge of building intelligent machines is not on the horizon, and so we instead create smaller-scale, tractable steppingstones. We hope these steppingstones nudge us toward machines that are intelligent in all the ways humans are, and our hope is that this thesis acts as one such steppingstone.

Developing a theory of program induction. The algorithms developed here rest on a heterogeneous set of techniques: neural nets, graphical models, type systems, reinforcement learning, etc. When can we expect these algorithmic contraptions to work? For now, this thesis offers only intuitions, and can prove little outside Appendix A. Intuitively, bootstrap learning systems such as DreamCoder will work when there is a spectrum of problem difficulties, and empirically seems to require on the order of 50-100 tasks; bootstrapping works best when inductive biases and search strategies can mutually train each other. Intuitively, REPL-guidance works when intermediate program states can be rendered in a manner suitable for a neural network. Intuitively, program induction works when problems admit program-*like* solutions. A theoretical challenge is to sharpen up these intuitions by deriving necessary or sufficient conditions for when similar systems will work in the future.

At a glance, this challenge might seem hopeless: given our failure to get a theoretical handle on SAT solvers (which are purely symbolic), why should we expect to reach a satisfying understanding of these heterogeneous program induction systems? History gives optimism. Program induction did not start with building software systems, but with proving theorems about the limits of program learning [126]. Intriguing, this theoretical work considers a setting where the learner has a single, very difficult problem to solve. As shown in this thesis, program induction works better when we instead have a training set of interrelated induction tasks. We suggest that, if those tasks share enough algorithmic

mutual information [80], then they should be much easier to solve jointly, and ideas around resource-bounded Kolmogorov complexity may be useful (cf. [115, 116]). We should revisit this early theoretical work in the multitask setting advocated here, both to understand when and why our methods work, and because it might help spur the next wave of program induction systems.

How do we build a program induction community?[3] Program induction won't take off without a strong community of researchers. Much as AlphaGo and AlexNet [70] catalyzed the modern deep learning renaissance, a similar catalyst—solving a practical unsolved engineering challenge—is needed to build interest in program induction. What kind of problem is most amenable to program induction, while also being an important outstanding challenge?

We believe solving certain kinds of model structure learning would serve as this catalyst. One version of this is sample-efficient model-based reinforcement learning, such as solving Atari from minutes of gameplay with modest prior knowledge. But this would still be a small-scale demonstration, and the real long-term vision is to scale these kinds of automatic theory induction, like in Chapter 4, toward actual models of the real world. A program induction system which could synthesize interpretable causal models of poorly-understood physical phenomena, where those models were useful for human scientists, is one such example. The goal would not be to automate away scientists—which is not on the horizon—but instead to provide a theory-building analog of computer algebra systems. Rather than model our research after the metaphor of the "robot scientist," we should adopt a new metaphor: the "inductive calculator." Outside science, one of the most useful model building activities for an embodied agent is to construct causal theories of the physical world, of objects and their physical properties. The program induction toolkit developed here has matured to the point where it can make real traction on problems like these: we

[3] We thank Kliment Serafimov for asking this question.

can infer structured 3D programs from perceptual input, grow reusable libraries of physical concepts like 'table', 'gear' or 'pulley', and learn-to-learn these models from less data by sharing statistical strength across related problems.

What we want for the future of machine intelligence. We believe these next steps are within reach. More broadly, what should the future of machine intelligence hold? We want learning algorithms which generalize strongly and abstractly, even on the basis of limited data. Yet, from larger amounts of data, we want systems which bootstrap, or learn-to-learn, and where this learning-to-learn is driven by representation learning. One way of seeing library learning is as a kind of hierarchical representation learning, but where the internal representation is built from symbolic code. Most importantly, we want machines which do not only learn to make predictions, but which discover something closer to knowledge. We want such knowledge to be usable by both humans and machines alike. To be clear, we do *not* claim that program induction alone will achieve this. Many ideas, both old, new, and yet to be discovered, are needed. But we view this thesis as part of an argument that program induction can be a core part of the story going forward, and that the steps we are taking are headed in these directions.

Appendix A

Proofs and Derivations

A.1 Version spaces

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 $\uparrow^{-1}_{n+c}\uparrow^{n+1}_{c} e = \uparrow^{n}_{c} e$, and in particular $\uparrow^{-1}_{n}\uparrow^{n+1} e = \uparrow^{n} e$.

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

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

Proof. Suppose $b = \$n$ and therefore, by the definition of S_n, also $v = \downarrow^{n}_{0} u$. Invoking Lemmas 1 and 2 we know that $u = \uparrow^{-1}_{n}\uparrow^{n+1} v$ and so for every $v' \in [\![v]\!]$ we have $\uparrow^{-1}_{n}\uparrow^{n+1} v' \in [\![u]\!]$. Because $b = \$n = b'$ we can rewrite this to $\uparrow^{-1}_{n} [\$n \mapsto \uparrow^{n+1} v']b' \in [\![u]\!]$.

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 $S_n(u) \subset \{(\lambda\lambda b)v \mid (\lambda b)v \in S_{n+1}(a)\}$. Let $(\lambda\lambda b)v \in S_n(u)$. 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:

$$S_n(u) \subset \{(\lambda b_f\ b_x)(v_f \cap v_x) \mid (\lambda b_f)v_f \in S_n(f),\ (\lambda b_x)v_x \in S_n(x)\}$$

  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 $(\lambda b)v \in S_n(u)$ arbitrarily. By the definition of S_n there is a z such that $(\lambda b) \in S_n(z)$, 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* $(\lambda b)v \in S_n(u)$ *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 $(\lambda\$n)(\downarrow^n u) \in S_n(u)$ 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 $(\lambda\$i)\Lambda \in S_n(u)$, 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 $(\lambda\$(i+1))\Lambda \in S_n(u)$, 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 $(\lambda u)\Lambda \in S_n(u)$ 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 $(\lambda c)v \in S_{n+1}(a)$ satisfying $v' \in [\![v]\!]$ and $c' \in [\![c]\!]$. By the definition of S_n we also know that $(\lambda\lambda c)v \in S_n(u)$, 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 $(\lambda b_f)v_f \in S_n(f)$ satisfying $v' \in [\![v_f]\!]$, $b'_f \in [\![b_f]\!]$ and a $(\lambda b_x)v_x \in S_n(x)$ satisfying $v' \in [\![v_x]\!]$, $b'_x \in [\![b_x]\!]$. By the definition of S_n we know that $(\lambda b_f\ b_x)(v_f \cap v_x) \in S_n(u)$ 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 $(\lambda b)v \in S_n(z)$ such that $v' \in [\![v]\!]$ and $b' \in [\![v]\!]$. By the definition of S_n we know that $(\lambda b)v$ 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* $p \longrightarrow_\beta p'$.

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 $(\lambda b)v \in S_0(u)$: From the definition of β-reduction we know that $p \longrightarrow_\beta \uparrow^{-1}\ [\$0 \mapsto \uparrow^1\ v']b'$. 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 $b' \longrightarrow_\beta b''$. So $p \longrightarrow_\beta \lambda b''$. 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 $f' \longrightarrow_\beta f''$. So $(f'\ x') \longrightarrow_\beta (f''\ x')$. 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 $p' \longrightarrow_\beta p$. 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 $p \longrightarrow_\beta p'$ *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 $p \longrightarrow_\beta p'$ 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 $(\lambda b)v \in S_0(u)$ 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 $b \longrightarrow_\beta b'$: 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 $f \longrightarrow_\beta f'$: 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 $x \longrightarrow_\beta x'$: 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* $p \underbrace{\longrightarrow_\beta q \longrightarrow_\beta \cdots \longrightarrow_\beta}_{\leq n \text{ times}} p'$ *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 $p \underbrace{\longrightarrow_\beta q \longrightarrow_\beta \cdots \longrightarrow_\beta}_{\leq n \text{ times}} p'$ then $q \underbrace{\longrightarrow_\beta \cdots \longrightarrow_\beta}_{\leq n-1 \text{ times}} p'$; induction on n gives $q \in [\![I\beta_{n-1}(p')]\!]$. Combined with $p \longrightarrow_\beta q$ 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 $p \longrightarrow_\beta q$ for a program $q \in [\![\underbrace{I\beta'(I\beta'(\cdots}_{i-1 \text{ times}} p'))]\!] \subseteq [\![I\beta_{i-1}(p')]\!]$. By induction $q \underbrace{\longrightarrow_\beta \cdots \longrightarrow_\beta}_{\leq i-1 \text{ times}} p'$, which combined with $p \longrightarrow_\beta q$ gives $p \underbrace{\longrightarrow_\beta q \longrightarrow_\beta \cdots \longrightarrow_\beta}_{\leq i \leq n \text{ times}} p'$. □

A.2 Symmetry breaking

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

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

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

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

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 $\mathcal{B}_x$, $p_1 = \texttt{(+ (+ 1 1) 1)}$ and $p_2 = \texttt{(+ 1 (+ 1 1))}$, where

$$Q_1(p_1|x) = \frac{\epsilon}{2} \qquad Q_1(p_2|x) = \frac{\epsilon}{2}$$
$$Q_2(p_1|x) = 0 \qquad Q_2(p_2|x) = \epsilon$$

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 $\text{P}[p_1|\mathcal{D}, \theta] = \text{P}[p_2|\mathcal{D}, \theta]$ (Algorithm 3), we have

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

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.

A.3 Estimating the continuous weights of a learned library

Within the DreamCoder algorithm, we use an EM approach to estimate the continuous parameters of the generative model, i.e. θ. Suppressing dependencies on $\mathcal{D}$, the EM updates are

$$\theta = \arg\max_{\theta} \log \mathrm{P}\left[\theta\right] + \sum_{x} \mathrm{E}_{q_x}\left[\log \mathrm{P}\left[\rho|\theta\right]\right] \tag{A.1}$$

$$q_x(\rho) \propto \mathrm{P}[x|\rho]\mathrm{P}[\rho|\theta]\mathbb{1}\left[\rho \in \mathcal{B}_x\right] \tag{A.2}$$

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 [14].

We write $c(e, \rho)$ to mean the number of times that library component e was used in program ρ; $c(\rho) = \sum_{e \in \mathcal{D}} c(e, \rho)$ to mean the total number of library routines used in program ρ; $c(\tau, \rho)$ to mean the number of times that type τ was the input to sample in Algorithm 3 while sampling program ρ. Jensen's inequality gives a lower bound on the

likelihood:

$$\begin{aligned}
&\sum_x \mathrm{E}_{q_x}\left[\log \mathrm{P}[\rho|\theta]\right] = \\
&\sum_{e\in\mathcal{D}} \log\theta_e \sum_x \mathrm{E}_{q_x}\left[c(e,\rho_x)\right] - \sum_\tau \mathrm{E}_{q_x}\left[\sum_x c(\tau,\rho_x)\right] \log \sum_{\substack{e:\tau'\in\mathcal{D}\\ \text{unify}(\tau,\tau')}} \theta_e \\
=&\sum_e C(e)\log\theta_e - \beta \sum_\tau \frac{\mathrm{E}_{q_x}\left[\sum_x c(\tau,\rho_x)\right]}{\beta} \log \sum_{\substack{e:\tau'\in\mathcal{D}\\ \text{unify}(\tau,\tau')}} \theta_e \\
\geq&\sum_e C(e)\log\theta_e - \beta \log \sum_\tau \frac{\mathrm{E}_{q_x}\left[\sum_x c(\tau,\rho_x)\right]}{\beta} \sum_{\substack{e:\tau'\in\mathcal{D}\\ \text{unify}(\tau,\tau')}} \theta_e \\
=&\sum_e C(e)\log\theta_e - \beta \log \sum_\tau \frac{R(\tau)}{\beta} \sum_{\substack{e:\tau'\in\mathcal{D}\\ \text{unify}(\tau,\tau')}} \theta_e
\end{aligned}$$

where we have defined

$$\begin{aligned}
C(e) &\triangleq \sum_x \mathrm{E}_{q_x}\left[c(e,\rho_x)\right] \\
R(\tau) &\triangleq \mathrm{E}_{q_x}\left[\sum_x c(\tau,\rho_x)\right] \\
\beta &\triangleq \sum_\tau \mathrm{E}_{q_x}\left[\sum_x c(\tau,\rho_x)\right]
\end{aligned}$$

Crucially it was defining β that let us use Jensen's inequality. Recalling that the prior over θ is a Dirichlet distribution, i.e. $\mathrm{P}\left[\theta\right] \triangleq \mathrm{Dir}(\alpha)$, we have the following lower bound on M-step objective:

$$\sum_e (C(e)+\alpha)\log\theta_e - \beta\log\sum_\tau \frac{R(\tau)}{\beta} \sum_{\substack{e:\tau'\in\mathcal{D}\\ \text{unify}(\tau,\tau')}} \theta_e \tag{A.3}$$

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

$$\frac{C(e) + \alpha}{\theta_e} \propto \sum_{\tau'} \mathbb{1}\left[\text{unify}(\tau, \tau')\right] R(\tau') \tag{A.4}$$

$$\theta_e \propto \frac{C(e) + \alpha}{\sum_{\tau'} \mathbb{1}\left[\text{unify}(\tau, \tau')\right] R(\tau')} \tag{A.5}$$

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 [71]. 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.

Appendix B

Hyperparameters, neural architecture details, and training details

B.1 Learning to infer graphics programs from hand-drawn images

B.1.1 Neural network for inferring specifications from hand drawings

High-level overview

For the model in Fig. 3-3, the distribution over the next drawing command factorizes as:

$$\mathrm{P}_\theta[t_1 t_2 \cdots t_K | I, S] = \prod_{k=1}^{K} \mathrm{P}_\theta \left[t_k | a_\theta \left(f_\theta(I, \mathrm{render}(S)) | \{t_j\}_{j=1}^{k-1} \right), \{t_j\}_{j=1}^{k-1} \right] \tag{B.1}$$

where $t_1 t_2 \cdots t_K$ are the tokens in the drawing command, I is the target image, S is a spec, θ are the parameters of the neural network, $f_\theta(\cdot, \cdot)$ is the image feature extractor

(convolutional network), and $a_\theta(\cdot|\cdot)$ is an attention mechanism. The distribution over specs factorizes as:

$$\mathrm{P}_\theta[S|I] = \prod_{n=1}^{|S|} \mathrm{P}_\theta[S_n|I, S_{1:(n-1)}] \times \mathrm{P}_\theta[\texttt{STOP}|I, S] \tag{B.2}$$

where $|S|$ is the length of spec S, the subscripts on S index drawing commands within the spec (so S_n is a sequence of tokens: $t_1 t_2 \cdots t_K$), and the `STOP` token is emitted by the network to signal that the spec explains the image.

Convolutional network

The convolutional network takes as input 2 256×256 images represented as a $2 \times 256 \times 256$ volume. These are passed through two layers of convolutions separated by ReLU nonlinearities and max pooling:

- Layer 1: 20 8×8 convolutions, 2 16×4 convolutions, 2 4×16 convolutions. Followed by 8×8 pooling with a stride size of 4.

- Layer 2: 10 8×8 convolutions. Followed by 4×4 pooling with a stride size of 4.

Autoregressive decoding of drawing commands

Given the image features f, we predict the first token (i.e., the name of the drawing command: `circle`, `rectangle`, `line`, or `STOP`) using logistic regression:

$$\mathrm{P}[t_1] \propto \exp\left(W_{t_1} f + b_{t_1}\right) \tag{B.3}$$

where W_{t_1} is a learned weight matrix and b_{t_1} is a learned bias vector.

Given an attention mechanism $a(\cdot|\cdot)$, subsequent tokens are predicted as:

$$\mathrm{P}[t_n|t_{1:(n-1)}] \propto \mathrm{MLP}_{t_1,n}(a(f|t_{1:(n-1)}) \oplus \bigoplus_{j<n} \mathrm{oneHot}(t_j)) \tag{B.4}$$

Thus each token of each drawing primitive has its own learned MLP. For predicting the coordinates of lines we found that using 32 hidden nodes with sigmoid activations worked well; for other tokens the MLP's are just logistic regression (no hidden nodes).

We use Spatial Transformer Networks [65] as our attention mechanism. The parameters of the spatial transform are predicted on the basis of previously predicted tokens. For example, in order to decide where to focus our attention when predicting the y coordinate of a circle, we condition upon both the identity of the drawing command (`circle`) and upon the value of the previously predicted x coordinate:

$$a(f|t_{1:(n-1)}) = \mathrm{AffineTransform}(f, \mathrm{MLP}_{t_1,n}(\bigoplus_{j<n} \mathrm{oneHot}(t_j))) \tag{B.5}$$

So, we learn a different network for predicting special transforms *for each drawing command* (value of t_1) and also *for each token of the drawing command*. These networks ($\mathrm{MLP}_{t_1,n}$ in equation B.5) have no hidden layers and output the 6 entries of an affine transformation matrix; see [65] for more details.

Training takes a little bit less than a day on a Nvidia TitanX GPU. The network was trained on 10^5 synthetic examples.

LSTM Baseline

We compared our deep network with a baseline that models the problem as a kind of image captioning. Given the target image, this baseline produces the program spec in one shot by using a CNN to extract features of the input which are passed to an LSTM which finally

predicts the spec token-by-token. This general architecture is used in several successful neural models of image captioning (e.g., [143]).

Concretely, we kept the image feature extractor architecture (a CNN) as in our model, but only passed it one image as input (the target image to explain). Then, instead of using an autoregressive decoder to predict a single drawing command, we used an LSTM to predict a sequence of drawing commands token-by-token. This LSTM had 128 memory cells, and at each time step produced as output the next token in the sequence of drawing commands. It took as input both the image representation and its previously predicted token.

Generating synthetic training data

We generated synthetic training data for the neural network by sampling LaTeX code according to the following generative process: First, the number of objects in the scene are sampled uniformly from 1 to 12. For each object we uniformly sample its identity (circle, rectangle, or line). Then we sample the parameters of the circles, than the parameters of the rectangles, and finally the parameters of the lines; this has the effect of teaching the network to first draw the circles in the scene, then the rectangles, and finally the lines. We furthermore put the circle (respectively, rectangle and line) drawing commands in order by left-to-right, bottom-to-top; thus the training data enforces a canonical order in which to draw any scene.

To make the training data look more like naturally occurring figures, we put a Chinese restaurant process prior [49] over the values of the X and Y coordinates that occur in the execution spec. This encourages reuse of coordinate values, and so produces training data that tends to have parts that are nicely aligned.

In the synthetic training data we excluded any sampled scenes that had overlapping drawing commands. As shown in the main paper, the network is then able to generalize to

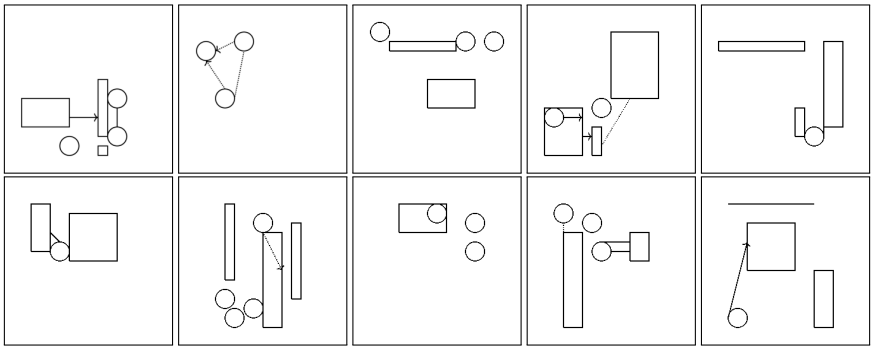


Figure B-1: Example synthetic training data

scenes with, for example, intersecting lines or lines that penetrate a rectangle.

When sampling the endpoints of a line, we biased the sampling process so that it would be more likely to start an endpoint along one of the sides of a rectangle or at the boundary of a circle. If n is the number of points either along the side of a rectangle or at the boundary of a circle, we would sample an arbitrary endpoint with probability $\frac{2}{2+n}$ and sample one of the "attaching" endpoints with probability $\frac{1}{2+n}$.

See figure B-1 for examples of the kinds of scenes that the network is trained on.

For readers wishing to generate their own synthetic training sets, we refer them to our source code at: `https://github.com/ellisk42/TikZ`.

B.1.2 End-to-End baseline

Recall that we factored the graphics program synthesis problem into two components: (1) a perception-facing component, whose job is to go from perceptual input to a set of commands that must occur in the execution of the program (**spec**); and (2) a program synthesis component, whose job is to infer a program whose execution contains those commands. This is a different approach from other recent program induction models (e.g., [33]), which regress directly from a program induction problem to the source code of

the program.

Experiment. To test whether this factoring is necessary for our domain, we trained a model to regress directly from images to graphics programs. This baseline model, which we call the *no-spec baseline*, was able to infer some simple programs, but failed completely on more sophisticated scenes.

Baseline model architecture: The model architecture is a straightforward, image-captioning-style CNN→LSTM. We keep the same CNN architecture from our main model (Section B.1.1), with the sole difference that it takes only one image as input. The LSTM decoder produces the program token-by-token: so we flatten the program's hierarchical structure, and use special "bracketing" symbols to convey nesting structure, in the spirit of [142]. The LSTM decoder has 2 hidden layers with 1024 units. We used 64-dimensional embeddings for the program tokens.

Training and evaluation: The model was trained on 10^7 synthetically generated programs – 2 orders of magnitude more data than the model we present in the main paper. We then evaluated the baseline on *synthetic renders* of our 100 hand drawings (the testing set used throughout the paper). Recall that our model was evaluated on noisy real hand drawings. We sample programs from this baseline model conditioned on a synthetic render of a hand drawing, and report only the sampled program whose output most closely matched the ground truth spec spec, as measured by the symmetric difference of the two sets. We allow the baseline model to spend 1 hour drawing samples per drawing – recall that our model finds 58% of programs in under a minute. Together these training and evaluation choices are intended to make the problem as easy as possible for the baseline.

Results: The no-spec baseline succeeds for trivial programs (a few lines, no variables, loops, etc.); occasionally gets small amounts of simple looping structure; and fails utterly for most of our test cases. See Figure B-2.

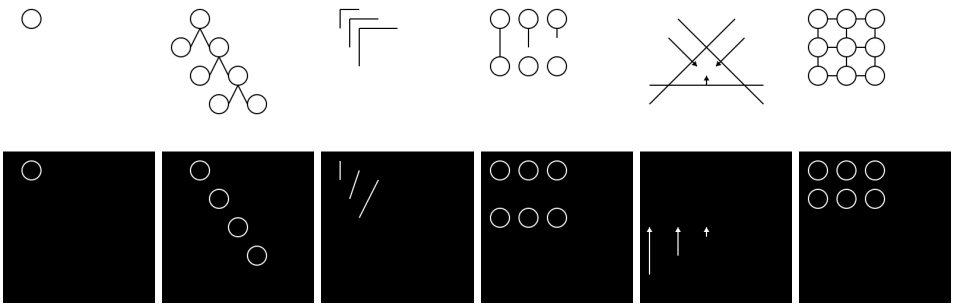


Figure B-2: Top, white: synthetic rendering of a hand drawing. Bottom, black: output of best program found by no-spec baseline.

B.1.3 DeepCoder-style baseline

In addition to the end-to-end baseline, we compared with a DeepCoder-style baseline. DeepCoder (DC) [10] is an approach for learning to speed up program synthesizers. DC models are neural networks that predict, starting from a spec, the probability of a DSL component being in a minimal-cost program satisfying the spec. Writing $\text{DC}(S)$ for the distribution predicted by the neural network, DC is trained to maximize the following objective:

$$\text{E}_{S\sim\mathcal{D}}\left[\min_{p\in\textsc{Best}(S)}\sum_{x\in\text{DSL}}\log\left(\mathbb{1}\left[x\in p\right]\text{DC}(S)_x+\mathbb{1}\left[x\notin p\right]\left(1-\text{DC}(S)_x\right)\right)\right] \tag{B.6}$$

where x ranges over DSL components and $\text{DC}(S)_x \in [0,1]$ is the probability predicted by the DC model for component x for spec S.

We provided our DC model with the same features given to our bias optimal search policy, and trained using the same 20-fold cross validation splits. To evaluate the DC baseline on held out data, we used the *Sort-and-Add* policy described in the DeepCoder paper [10].

B.2 Program synthesis with a REPL

B.2.1 Network architecture

The code-writing policy is a CNN followed by a pointer network which decodes into the next line of code. A pointer network [141] is an RNN that uses a differentiable attention mechanism to emit pointers, or indices, into a set of objects. Here the pointers index into the set of partial programs pp in the current state, which is necessary for the union and difference operators. Because the CNN takes as input the current REPL state – which may have a variable number of objects in scope – we encode each object with a separate CNN and sum their activations, i.e. a 'Deep Set' encoding [149]. The value function is an additional 'head' to the pooled CNN activations.

Concretely the neural architecture has a *spec encoder*, which is a CNN inputting a single image, as well as a *canvas encoder*, which is a CNN inputting a single canvas in the REPL state, alongside the spec, as a two-channel image. The canvas encoder output activations are summed and concatenated with the spec encoder output activations to give an embedding of the state:

$$\text{stateEncoding}(spec, pp) = \text{specEncoder}(spec) \otimes \sum_{p \in pp} \text{canvasEncoder(spec,p)} \qquad \text{(B.7)}$$

for W_1, W_2 weight matrices.

For the policy we pass the state encoding to a pointer network, implemented using a GRU with 512 hidden units and one layer, which predicts the next line of code. To attend to canvases $p \in pp$, we use the output of the canvas encoder as the 'key' for the attention mechanism.

For the value function we passed the state in coding to a MLP with 512 hidden units w/ a hidden ReLU activation, and finally apply a negated 'soft plus' activation to the output to

ensure that the logits output by the value network is nonpositive:

$$v(spec, pp) = -\text{SoftPlus}(W_2\text{ReLU}(W_1\text{stateEncoding}(spec, pp))) \tag{B.8}$$

$$\text{SoftPlus}(x) = \log(1 + e^x) \tag{B.9}$$

2-D CNN architecture: The 2D spec encoder and canvas encoder both take as input 64×64 images, passed through 4 layers of 3x3 convolution, with ReLU activations after each layer and 2x2 max pooling after the first two layers. The first 3 layers have 32 hidden channels and the output layer has 16 output channels.

3-D CNN architecture: The 3D spec encoder and canvas encoder both take as input $32 \times 32 \times 32$ voxel arrays, passed through 3 layers of 3x3 convolution, with ReLU activations after each layer and 4x4 max pooling after the first layer. The first 2 layers have 32 hidden channels and the output layer has 16 output channels.

No REPL baseline: Our "No REPL" baselines using the same CNN architecture to take as input the *spec*, and then use the same pointer network architecture to decode into the program, with the sole difference that, rather than attend over the CNN encodings of the objects in scope (which are hidden from this baseline), the pointer network attends over the hidden states produced at the time when each previously constructed object was brought into scope.

B.2.2 Training details

We train our models on randomly generated scenes with up to 13 objects, for approximately three days with one p100 GPU. We use the Adam optimizer [] wiht default settings. Over three days the 3D model saw approximately 1.7 million examples and the 2D model saw approximately 5 million examples. We fine-tuned the policy using REINFORCE and

trained the value network for approximately 5 days on one p100 GPU. For each gradient step during this process we sampled $B_1 = 2$ random programs and performed $B_2 = 16$ rollouts for a total batch size of $B = B_1 \times B_2 = 32$. During reinforcement learning the 3D model saw approximately 0.25 million examples and the 2D model saw approximately 9 million examples.

We generate a scene with up to k objects by sampling a number between 1 to k, and then sampling a random CSG tree with that many objects. We then remove any subtrees that do not affect the final render (e.g., subtracting pixels from empty space). Our held-out test set is created by sampling 100 random scenes with up to $k = 20$ objects for 3D and $k = 30$ objects for 2D.

B.3 DreamCoder

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 (Sec. 5.1.2)	α (Sec. A.3)	Beam size (Sec. 5.1.1)
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 unsolved (Sec. 5-21)	1	30	5
Recursive functions	4h	64	batch unsolved (Sec. 5-21)	1	30	5

Bibliography

[1] Adam Albright and Bruce Hayes. Rules vs. analogy in english past tenses: A computational/experimental study. *Cognition*, 90(2):119–161, 2003.

[2] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.

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

[4] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 319–336. Springer, 2017.

[5] David Alvarez-Melis and Tommi S Jaakkola. Tree-structured decoding with doubly-recurrent neural networks. 2016.

[6] David Andre and Stuart J Russell. State abstraction for programmable reinforcement learning agents. 2002.

[7] 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.

[8] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, pages 5360–5370, 2017.

[9] Richard N Aslin and Elissa L Newport. Statistical learning from acquiring specific items to forming general rules. *Current directions in psychological science*, 21(3):170–176, 2012.

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

[11] Shraddha Barke, Rose Kunkel, Polikarpova Nadia, and Leon Bergen. Constraint-based learning of phonological processes. *Under review at EMNLP*, 2019.

[12] Clark Barrett, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (smt-lib). www. *SMT-LIB. org*, 15:18–52, 2010.

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

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

[15] Ned Block. Conceptual role semantics. 1998.

[16] M. M. Bongard. *Pattern Recognition*. Spartan Books, 1970.

[17] Rudy Bunel, Alban Desmaison, M Pawan Kumar, Philip HS Torr, and Pushmeet Kohli. Learning to superoptimize programs. *arXiv preprint arXiv:1611.01787*, 2016.

[18] Susan Carey. *Conceptual Change In Childhood*. MIT Press, 1985.

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

[20] Susan E. Carey. *The Origin of Concepts*. Oxford University Press, 2009.

[21] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. 2018.

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

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

[24] 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.

[25] N. Chomsky. *Current Issues in Linguistic Theory*. Janua Linguarum. Series Minor. De Gruyter, 1988.

[26] N. Chomsky and M. Halle. *The sound pattern of English*. Studies in language. Harper & Row, 1968.

[27] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[28] Ryan Cotterell, Nanyun Peng, and Jason Eisner. Modeling word forms using latent underlying morphs and phonology. *Transactions of the Association for Computational Linguistics*, 3:433–447, 2015.

[29] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

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

[31] A. P. Dempster, M. N. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 39:1–22, 1977.

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

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

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

[35] Arnaud Doucet, Nando De Freitas, and Neil Gordon, editors. *Sequential Monte Carlo Methods in Practice*. Springer, 2001.

[36] Kevin Ellis and Sumit Gulwani. Learning to learn programs from examples: Going beyond program structure. *IJCAI*, 2017.

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

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

[39] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Unsupervised learning by program synthesis. In *NIPS*, pages 973–981, 2015.

[40] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Sampling for bayesian program learning. In *Advances in Neural Information Processing Systems*, 2016.

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

[42] SM Eslami, N Heess, and T Weber. Attend, infer, repeat: Fast scene understanding with generative models. arxiv preprint arxiv:..., 2016. *URL http://arxiv.org/abs/1603.08575*.

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

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

[45] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR. org, 2017.

[46] Michael C Frank and Joshua B Tenenbaum. Three ideal observer models for rule learning in simple languages. *Cognition*, 120(3):360–371, 2011.

[47] Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, SM Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. *ICML*, 2018.

[48] LouAnn Gerken. Infants use rational decision criteria for choosing among models of their input. *Cognition*, 115(2):362–366, 2010.

[49] Samuel J Gershman and David M Blei. A tutorial on bayesian nonparametric models. *Journal of Mathematical Psychology*, 56(1):1–12, 2012.

[50] Tobias Gerstenberg and Joshua B Tenenbaum. Intuitive theories.

[51] Jeremy Gibbons. *Origami programming*. 2003.

[52] Daniel Gildea and Daniel Jurafsky. Learning bias and phonological-rule induction. *Computational Linguistics*, 22(4):497–530, 1996.

[53] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[54] 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.

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

[56] Sumit Gulwani, Jose Hernandez-Orallo, Emanuel Kitzelmann, Stephen Muggleton, Ute Schmid, and Ben Zorn. Inductive programming meets the real world. *Commun. ACM*, 2015.

[57] Morris Halle and George N Clements. *Problem book in phonology: a workbook for introductory courses in linguistics and in modern phonology*. MIT Press, 1983.

[58] Jeffrey Heinz. Computational phonology–part i: foundations. *Language and Linguistics Compass*, 5(4):140–152, 2011.

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

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

[61] 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.

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

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

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

[65] Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. Spatial transformer networks. In *NIPS*, 2015.

[66] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *ICLR*, 2018.

[67] Ronald M. Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, 1994.

[68] 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.

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

[70] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

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

[72] 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.

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

[74] 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.

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

[76] Pat Langley, Gary L Bradshaw, and Herbert A Simon. Bacon. 5: The discovery of conservation laws. In *IJCAI*, volume 81, pages 121–126. Citeseer, 1981.

[77] 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.

[78] 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.

[79] Leonid Anatolevich Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

[80] Ming Li and Paul M.B. Vitnyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Publishing Company, Incorporated, 3 edition, 2008.

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

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

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

[84] 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.

[85] 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.

[86] 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.

[87] Christopher A Mattson and Achille Messac. Pareto frontier based concept selection under uncertainty, with visualization. *Optimization and Engineering*, 6(1):85–115, 2005.

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

[89] 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.

[90] Melanie Mitchell. *Artificial Intelligence: A Guide for Thinking Humans*. Farrar, Straus and Giroux, 2019.

[91] 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.

[92] Elliott Moreton. Analytic bias and phonological typology. *Phonology*, 25(1):83–127, 2008.

[93] Stephen Muggleton and Wray L. Buntine. Machine Invention of First Order Predicates by Inverting Resolution. In *Proceedings of the 5th International Conference on Machine Learning*, pages 339–352. Morgan Kaufman, 1988.

[94] 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.

[95] Stephen H Muggleton, Ute Schmid, Christina Zeller, Alireza Tamaddoni-Nezhad, and Tarek Besold. Ultra-strong machine learning: comprehensibility of programs learned with ilp. *Machine Learning*, 107(7):1119–1140, 2018.

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

[97] David Odden. *Introducing phonology*. Cambridge university press, 2005.

[98] Timothy J. O'Donnell. *Productivity and Reuse in Language: A Theory of Linguistic Computation and Storage*. The MIT Press, 2015.

[99] Brooks Paige and Frank Wood. Inference networks for sequential monte carlo in graphical models. In *International Conference on Machine Learning*, pages 3040–3049, 2016.

[100] Judea Pearl. *Causality: models, reasoning and inference*, volume 29. Springer.

[101] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *ACM Sigplan Notices*, volume 49, pages 408–418. ACM, 2014.

[102] Amy Perfors, Joshua B Tenenbaum, and Terry Regier. The learnability of abstract syntactic principles. *Cognition*, 118(3):306–338, 2011.

[103] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[104] 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).

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

[106] Yewen Pu, Zachery Miranda, Armando Solar-Lezama, and Leslie Kaelbling. Selecting representative examples for program synthesis. In *International Conference on Machine Learning*, pages 4158–4167, 2018.

[107] E. S. Raimy. *Representing Reduplication*. PhD thesis, University of Delaware, 1999.

[108] Ezer Rasin and Iddo Berger. Learning rule-based morpho-phonology. 2015.

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

[110] Daniel Ritchie, Anna Thomas, Pat Hanrahan, and Noah Goodman. Neurally-guided procedural models: Amortized inference for procedural graphics programs using neural networks. In *Advances In Neural Information Processing Systems*, pages 622–630, 2016.

[111] Iggy Roca and Wyn Johnson. *A workbook in phonology*. Oxford, UK: Blackwell, 1991.

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

[113] Feras A. Saad, Marco F. Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages*, 3(POPL):37:1–37:32, January 2019.

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

[115] Jürgen Schmidhuber. The speed prior: a new simplicity measure yielding near-optimal computable predictions. In *COLT*, pages 216–228. Springer, 2002.

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

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

[118] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. Csgnet: Neural shape parser for constructive solid geometry. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5515–5523, 2018.

[119] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

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

[121] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *CAV*, pages 398–414. Springer, 2015.

[122] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.

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

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

[125] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ACM Sigplan Notices*, volume 41, pages 404–415. ACM, 2006.

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

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

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

[129] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. From program verification to program synthesis. In *ACM Sigplan Notices*, volume 45, pages 313–326. ACM, 2010.

[130] David Stampe. *How I spent my summer vacation*. PhD thesis, University of Chicago, 1973.

[131] 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.

[132] 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.

[133] Gerald Tesauro. Temporal difference learning and td-gammon. *CACM*, 1995.

[134] David D. Thornburg. Friends of the turtle. *Compute!*, March 1983.

[135] Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T Freeman, Joshua B Tenenbaum, and Jiajun Wu. Learning to infer and execute 3d shape programs. *ICLR*, 2019.

[136] 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.

[137] Alan M Turing. Computing machinery and intelligence. *Mind*, 1950.

[138] Silviu-Marian Udrescu and Max Tegmark. Ai feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020.

[139] 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.

[140] 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.

[141] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2674–2682, 2015.

[142] Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. In *Advances in Neural Information Processing Systems*, pages 2773–2781, 2015.

[143] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3156–3164, 2015.

[144] Patrick Winston. The MIT robot. *Machine Intelligence*, 1972.

[145] David H Wolpert. The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390, 1996.

[146] Jiajun Wu, Joshua B Tenenbaum, and Pushmeet Kohli. Neural scene de-rendering. In *CVPR*, 2017.

[147] Pengcheng Yin and Graham Neubig. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *EMNLP*, 2018.

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

[149] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan R Salakhutdinov, and Alexander J Smola. Deep sets. In *Advances in neural information processing systems*, pages 3391–3401, 2017.

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

[151] Amit Zohar and Lior Wolf. Automatic program synthesis of long programs with a learned garbage collector. In *Advances in Neural Information Processing Systems*, pages 2094–2103, 2018.