f f f

# DreamCoder: Bootstrapping Domain-Specific Languages for Neurally-Guided Bayesian Program Learning

**Anonymous Authors**[1]

## Abstract

Successful approaches to program induction require a hand-engineered domain-specific language (DSL), constraining the space of allowed programs and imparting prior knowledge of the domain. We contribute a program induction algorithm called DreamCoder that learns a DSL while jointly training a neural network to efficiently search for programs in the learned DSL. We use our model to synthesize functions on lists, edit text, and solve symbolic regression problems, showing how the model learns a domain-specific library of program components for expressing solutions to problems in the domain.

## 1. Introduction

Much of everyday human thinking and learning can be understood in terms of program induction: constructing a procedure that maps inputs to desired outputs, based on observing example input-output pairs. People can induce programs flexibly across many different domains, and remarkably, often from just one or a few examples. For instance, if shown that a text-editing program should map "Jane Morris Goodall" to "J. M. Goodall", we can guess it maps "Richard Erskine Leakey" to "R. E. Leakey"; if instead the first input mapped to "Dr. Jane", "Goodall, Jane", or "Morris", we might have guessed the latter should map to "Dr. Richard", "Leakey, Richard", or "Erskine", respectively.

The FlashFill system (Gulwani, 2011) developed by Microsoft researchers and now embedded in Excel solves problems such as these and is probably the best known practical program-induction algorithm, but researchers in programming languages and AI have built successful program induction algorithms for many applications, such as handwriting recognition and generation (Lake et al., 2015), procedural graphics (Ellis et al., 2017), question answering (Johnson et al., 2017) and robot motion planning (Devlin et al., 2017a), to name just a few. These systems work in different ways, but most hinge upon having a carefully engineered **Domain Specific Language (DSL)**. This is especially true for systems such as FlashFill that aim to induce a wide range of programs very quickly, in a few seconds or less. DSLs constrain the search over programs with strong prior knowledge in the form of a restricted set of programming primitives tuned to the needs of the domain: for text editing, these are operations like appending strings and splitting on characters.

In this work, we consider the problem of building agents that learn to solve program induction tasks, and also the problem of acquiring the prior knowledge necessary to quickly solve these tasks in a new domain. Representative problems in three domains are shown in Table 1. Our solution is an algorithm that grows or boostraps a DSL while jointly training a neural network to help write programs in the increasingly rich DSL. Because any computable learning problem can in principle be cast as program induction, it is important to delimit our focus. In contrast to computer assisted programming (Solar Lezama, 2008) or genetic programming (Koza, 1993), our goal is not to automate software engineering, to learn to synthesize large bodies of code, or to learn complex programs starting from scratch. Ours is a basic AI goal: capturing the human ability to learn to think flexibly and efficiently in new domains — to learn what you need to know about a domain so you don't have to solve new problems starting from scratch. We are focused on the kinds of problems that humans can solve relatively quickly, once they acquire the relevant domain expertise. These correspond to tasks solved by short programs — if you have an expressive DSL. Even with a good DSL, program search may be intractable; so we amortize the cost of search by training a neural network to assist the search procedure.

Our algorithm takes inspiration from several ways that skilled human programmers have learned to code: Skilled coders build libraries of reusable subroutines that are shared across related programming tasks, and can be composed to generate increasingly complex and powerful subroutines. In text editing, a good library should support routines for

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

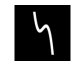
| | List Functions | Text Editing | Symbolic Regression |
|---|---|---|---|
| Programs & Tasks | [7 2 3]→[7 3]<br>[1 2 3 4]→[3 4]<br>[4 3 2 1]→[4 3]<br>$f(\ell) = (f_1\ \ell\ (\lambda\ (x)\ (>\ x\ 2)))$<br>[7 3]→False<br>[3]→False<br>[9 0 0]→True<br>[0]→True<br>[0 7 3]→True<br>$f(\ell) = (f_3\ \ell\ 0)$<br>[2 7 8 1]→8<br>[3 19 14]→19<br>$f(\ell) = (f_2\ \ell)$ | +106 769-438→106.769.438<br>+83 973-831→83.973.831<br>$f(s) = (f_0$ "." "-" $(f_0$ "." " " (cdr s)))<br>Temple Anna H →TAH<br>Lara Gregori→LG<br>$f(s) = (f_2\ s)$ |  $f(x) = (f_1\ x)$<br> $f(x) = (f_6\ x)$<br> $f(x) = (f_4\ x)$<br> $f(x) = (f_3\ x)$ |
| DSL | $f_0(\ell,r)$ = (foldr r ℓ cons)<br>($f_0$: *Append lists* r *and* ℓ)<br>$f_1(\ell,p)$ = (foldr ℓ nil (λ (x a) (if (p x) (cons x a) a)))<br>($f_1$: *Higher-order filter function*)<br>$f_2(\ell)$ = (foldr ℓ 0 (λ (x a) (if (> a x) a x)))<br>($f_2$: *Maximum element in list* ℓ)<br>$f_3(\ell,k)$ = (foldr ℓ (is-nil ℓ) (λ (x a) (if a a (= k x))))<br>($f_3$: *Whether* ℓ *contains* k) | $f_0(s,a,b)$ = (map (λ (x) (if (= x a) b x)) s)<br>($f_0$: *Performs character substitution*)<br>$f_1(s,c)$ = (foldr s s (λ (x a) (cdr (if (= c x) s a))))<br>($f_1$: *Drop characters from* s *until* c *reached*)<br>$f_2(s)$ = (unfold s is-nil car (λ (z) ($f_1$ z " ")))<br>($f_2$: *Abbreviates a sequence of words*)<br>$f_3(a,b)$ = (foldr a b cons)<br>($f_3$: *Concatenate strings* a *and* b) | $f_0(x)$ = (+ x real)<br>$f_1(x)$ = ($f_0$ (* real x))<br>$f_2(x)$ = ($f_1$ (* x ($f_0$ x)))<br>$f_3(x)$ = ($f_0$ (* x ($f_2$ x)))<br>$f_4(x)$ = ($f_0$ (* x ($f_3$ x)))<br>($f_4$: *4th order polynomial*)<br>$f_5(x)$ = (/ real x)<br>$f_6(x)$ = ($f_4$ ($f_0$ x))<br>($f_6$: *rational function*) |

Table 1: Top: Tasks from each domain, each followed by the programs DREAMCODER discovers for them. Bottom: Several examples from learned DSL. Notice that learned DSL primitives can call each other, and that DREAMCODER rediscovers higher-order functions like `filter` ($f_1$ in List Functions)

splitting on characters, but also specialize these routines to split on particular characters such as spaces or commas that are frequently used to delimit substrings across tasks. Skilled coders also learn to recognize what kinds of programming idioms and library routines would be useful for solving the task at hand, even if they cannot instantly work out the details. In text editing, one might learn that if outputs are consistently shorter than inputs, removing characters is likely to be part of the solution; if every output contains a constant substring (e.g., “Dr.”), inserting or appending that constant string is likely to be a subroutine.

Our DREAMCODER algorithm incorporates these insights by iterating through three steps. The **Search** step takes a given set of **tasks**, typically several hundred, and searches for compact programs that solve these tasks guided by the current DSL and neural network. The **Compress** step grows the library (or DSL) of domain-specific subroutines which allow the agent to more compactly write programs in the domain; it modifies the structure of the DSL by discovering regularities across programs found during search, compressing them to distill out common code fragments across successful programs. The **Compile** step improves the search procedure by training a neural network to write programs in the current DSL, in the spirit of “amortized” or “compiled” inference (Le et al., 2017).

The learned DSL effectively encodes a prior on programs likely to solve tasks in the domain, while the neural net looks at the example input-output pairs for a specific task and produces a “posterior” for programs likely to solve that specific task. The neural network thus functions as a **recognition model** supporting a form of approximate Bayesian program induction, jointly trained with a **generative model** for programs encoded in the DSL, in the spirit of the Helmholtz machine (Hinton et al., 1995)). The recognition model ensures that searching for programs remains tractable even as the DSL (and hence the search space for programs) expands.

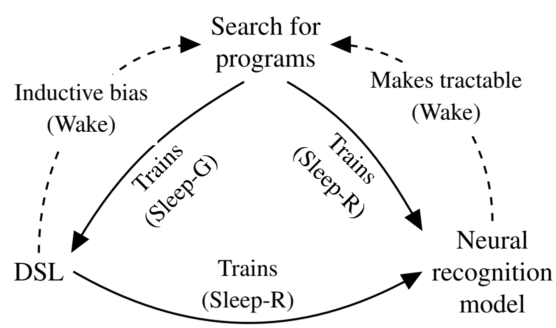



Figure 1: DREAMCODER solves for programs, the DSL, and a recognition model. Each of these steps bootstrap off of the others in a Helmholtz-machine inspired wake/sleep inference algorithm.

We apply DREAMCODER to three domains: list processing; text editing (in the style of FlashFill (Gulwani, 2011)); and symbolic regression. For each of these we initially provide a generic set of programming primitives. Our algorithm then constructs its own DSL for expressing solutions in the domain (Tbl. 1).

## 2. The DreamCoder Algorithm

## 3. Programs that manipulate sequences

We apply DreamCoder to list processing (Section 3.1) and text editing (Section 3.2). For both these domains we use a bidirectional GRU (Cho et al., 2014) for the recognition model, and initially provide the system with a generic set of list processing primitives: `foldr`, `unfold`, `if`, `map`, `length`, `index`, `=`, `+`, `−`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`.

### 3.1. List Processing

Synthesizing programs that manipulate data structures is a widely studied problem in the programming languages community (Feser et al., 2015). We consider this problem within the context of learning functions that manipulate lists, and which also perform arithmetic operations upon lists of numbers.

| Name | Input | Output |
|---|---|---|
| repeat-2 | [7 0] | [7 0 7 0] |
| drop-3 | [0 3 8 6 4] | [6 4] |
| rotate-2 | [8 14 1 9] | [1 9 8 14] |
| count-head-in-tail | [1 2 1 1 3] | 2 |
| keep-mod-5 | [5 9 14 6 3 0] | [5 0] |
| product | [7 1 6 2] | 84 |

Table 2: Some tasks in our list function domain. See the supplement for the complete data set.

We created 236 human-interpretable list manipulation tasks, each with 15 input/output examples (Tbl. 2). Our data set is interesting in three major ways: many of the tasks require complex solutions; the tasks were not generated from some latent DSL, and the agent must learn to solve these complicated problems from only 236 tasks. Our data set assumes arithmetic operations as well as sequence operations, so we additionally provide our system with the following arithmetic primitives: `mod`, `*`, `>`, `is-square`, `is-prime`.

We evaluated DreamCoder on random 50/50 test/train split. Interestingly, we found that the recognition model provided little benefit for the training tasks. However, it yielded faster search times on held out tasks, allowing more tasks to be solved before timing out. The system composed 38 new subroutines, yielding a more expressive DSL more closely matching the domain (left of Tbl. 1, right of Fig. **??**). See the supplement for a complete list of DSL primitives discovered by DreamCoder.

### 3.2. Text Editing

Synthesizing programs that edit text is a classic problem in the programming languages and AI literatures (Menon et al., 2013; Lau, 2001), and algorithms that learn text editing programs ship in Microsoft Excel (Gulwani, 2011). This prior work presumes a hand-engineered DSL. We show DreamCoder can instead start out with generic sequence manipulation primitives and recover many of the higher-level building blocks that have made these other text editing systems successful.

Because our enumerative search procedure cannot generate string constants, we instead enumerate programs with string-valued parameters. For example, to learn a program that prepends "Dr.", we enumerate $(f_3$ `string s`$)$ – where $f_3$ is the learned appending primitive (Fig. 1) — and then define $\mathbb{P}[x|p]$ by approximately marginalizing out the string parameters via a simple dynamic program. In Sec. 4, we will use a similar trick to synthesize programs containing real numbers, but using gradient descent instead of dynamic programming.

We trained our system on a corpus of 109 automatically generated text editing tasks, with 4 input/output examples each. After three iterations, it assembles a DSL containing a dozen new functions (center of Fig. 1) that let it solve all of the training tasks. But, how well does the learned DSL generalized to real text-editing scenarios? We tested, but did not train, on the 108 text editing problems from the SyGuS (Alur et al., 2016) program synthesis competition. Before any learning, DreamCoder solves 3.7% of the problems with an average search time of 235 seconds. After learning, it solves 74.1%, and does so much faster, solving them in an average of 29 seconds. As of the 2017 SyGuS competition, the best-performing algorithm solves 79.6% of the problems. But, SyGuS comes with a different hand-engineered DSL *for each text editing problem.*[1] Here we learned a single DSL that applied generically to all of the tasks, and perform comparably to the best prior work.

## 4. Symbolic Regression: Programs from visual input

We apply DreamCoder to symbolic regression problems. Here, the agent observes points along the curve of a function, and must write a program that fits those points. We initially equip our learner with addition, multiplication, and division, and task it with solving 100 symbolic regression problems, each either a polynomial of degree 1–4 or a rational function. The recognition model is a convolutional network that observes an image of the target function's

[1] SyGuS text editing problems also prespecify the set of allowed string constants for each task. For these experiments, our system did not use this assistance.

graph (Fig. 2) — visually, different kinds of polynomials and rational functions produce different kinds of graphs, and so the recognition model can learn to look at a graph and predict what kind of function best explains it. A key difficulty, however, is that these problems are best solved with programs containing real numbers. Our solution to this difficulty is to enumerate programs with real-valued parameters, and then fit those parameters by automatically differentiating through the programs the system writes and use gradient descent to fit the parameters. We define the likelihood model, $\mathbb{P}[x|p]$, by assuming a Gaussian noise model for the input/output examples, and penalize the use of real-valued parameters using the BIC (Bishop, 2006).

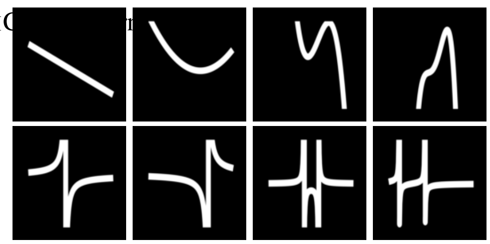


Figure 2: Recognition model input for symbolic regression. DSL learns subroutines for polynomials (top row) and rational functions (bottom row) while the recognition model jointly learns to look at a graph of the function (above) and predict which of those subroutines best explains the observation.

DreamCoder learns a DSL containing 13 new functions, most of which are templates for polynomials of different orders or ratios of polynomials. It also learns to find programs that minimize the number of continuous degrees of freedom. For example, it learns to represent linear functions with the program `(* real (+ x real))`, which has two continuous degrees of freedom, and represents quartic functions using the invented DSL primitive $f_4$ in the rightmost column of Fig. 1 which has five continuous parameters. This phenomenon arises from our Bayesian framing — both the implicit bias towards shorter programs and the likelihood model's BIC penalty.

## 5. Quantitative Results

We compare with ablations of our model on held out tasks. The purpose of this ablation study is both to examine the role of each component of DreamCoder, as well as to compare with prior approaches in the literature: a head-to-head comparison of program synthesizers is complicated by the fact that each system, including ours, makes idiosyncratic assumptions about the space of programs and the statement of tasks. Nevertheless, much prior work can be modeled within our setup. We compare with the following ablations (Tbl 3; Fig 3):

| | Ours | No NN | SE | NPS | PCFG | Enum |
|---|---|---|---|---|---|---|
| | | *List Processing* | | | | |
| % solved | **79%**[2] | 76% | 71% | 35% | 62% | 37% |
| Solve time | 4.1s | 5.8s | 10.6s | 34.7s | 43.4s | 20.2s |
| | | *Text Editing* | | | | |
| % solved | **74%** | 43% | 30% | 33% | 0% | 4% |
| Solve time | 29s | 65s | 38s | 80s | – | 235s |
| | | *Symbolic Regression* | | | | |
| % solved | **84%** | 75% | 62% | 38% | 38% | 37% |
| Solve time | 24s | 40s | 28s | 31s | 55s | 29s |

Table 3: % held-out test tasks solved. Solve time: averaged over solved tasks.

**No NN:** lesions the recognition model.

**NPS**, which does not learn the DSL, instead learning the recognition model from samples drawn from the fixed DSL. We call this NPS (Neural Program Synthesis) because this is closest to how RobustFill (Devlin et al., 2017b) and DeepCoder (Balog et al., 2016) are trained.

**SE**, which lesions the recognition model and restricts the DSL learning algorithm to only add **S**ub**E**xpressions of programs in the frontiers to the DSL. This is how most prior approaches have learned libraries of functions (Dechter et al., 2013; Liang et al., 2010; Lin et al., 2014).

**PCFG**, which lesions the recognition model and does not learn the DSL, but instead learns the parameters of the DSL ($\theta$), learning the parameters of a PCFG while not learning any of the structure.

**Enum**, which enumerates a frontier without any learning — equivalently, our first search step.

## 6. Related Work

Our work is far from the first for learning to learn programs, an idea that goes back to Solomonoff (Solomonoff, 1989):

**Deep learning:** Much recent work in the ML community has focused on creating neural networks that regress from input/output examples to programs (Devlin et al., 2017b;a; Menon et al., 2013; Balog et al., 2016). DreamCoder's recognition model draws heavily from this line of work, particularly from (Menon et al., 2013). We see these prior works as operating in a different regime: typically, they train with strong supervision (i.e., with annotated ground-truth programs) on massive data sets (i.e., hundreds of millions (Devlin et al., 2017b)). Our work considers a weakly-supervised regime where ground truth programs are not provided and the agent must learn from at most a few hundred tasks, which is facilitated by our "Helmholtz machine" style recognition model.

**Inventing new subroutines for program induction:** Several program induction algorithms, most prominently the EC algorithm (Dechter et al., 2013), take as their goal to

[2] Due to a last-minute bug we ran the list processing experiments without training the recognition model on "Helmholtz machine" style data. We are rerunning this experiment and anticipate improved quantitative results for list processing — notice the recognition model helps much more for text editing and symbolic regression.

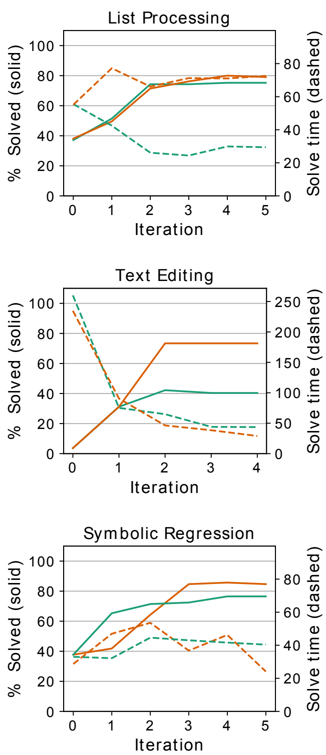



Figure 3: Learning curves for DREAMCODER both with (in orange) and without (in teal) the recognition model. Solid lines: % holdout testing tasks solved. Dashed lines: Average solve time.

learn new, reusable subroutines that are shared in a multi-task setting. We find this work inspiring and motivating, and extend it along two dimensions: (1) we propose a new algorithm for inducing reusable subroutines, based on Fragment Grammars (O'Donnell, 2015); and (2) we show how to combine these techniques with bottom-up neural recognition models. Other instances of this related idea are (Liang et al., 2010), Schmidhuber's OOPS model (Schmidhuber, 2004), and predicate invention in Inductive Logic Programming (Lin et al., 2014).

**Bayesian Program Learning:** Our work is an instance of Bayesian Program Learning (BPL; see (Lake et al., 2015; Dechter et al., 2013; Ellis et al., 2016; Liang et al., 2010)). Previous BPL systems have largely assumed a fixed DSL (but see (Liang et al., 2010)), and our contribution here is a general way of doing BPL with less hand-engineering of the DSL.

## 7. Discussion

We contribute an algorithm, DREAMCODER, that learns to program by bootstrapping a DSL with new domain-specific primitives that the algorithm itself discovers, together with a neural recognition model that learns how to efficiently deploy the DSL on new tasks. We believe this integration of top-down symbolic representations and bottom-up neural networks — both of them learned — helps make program induction systems more generally useful for AI. Many directions remain open. Two immediate goals are to integrate more sophisticated neural recognition models (Devlin et al., 2017b) and program synthesizers (Solar Lezama, 2008), which may improve performance in some domains over the generic methods used here. We are in the process of applying our algorithm to generative programs and prototyped this with a turtle-like domain: Figure 4 gives some preliminary results for turtle graphics — see Section 1 of the supplementary material for more details. Another direction is to explore DSL meta-learning: can we find a *single* universal primitive set that could effectively bootstrap DSLs for new domains, including the three domains considered, but also many others?

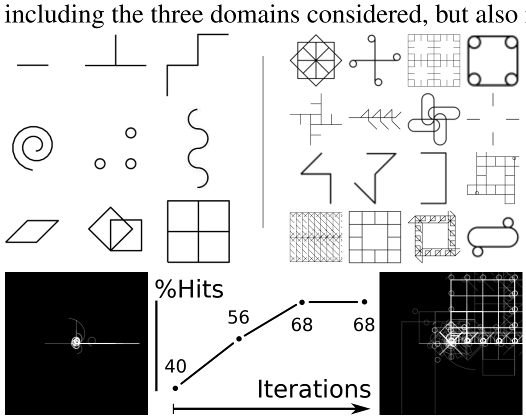



Figure 4: Top left: Example training tasks. Top right: samples from the learned DSL. Bottom: % holdout testing tasks solved (middle), on sides are the averaged samples from the DSL before any training (left) and after last iteration (right).

## References

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

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

Bishop, Christopher M. *Pattern Recognition and Machine Learning*. 2006.

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

Cohn, Trevor, Blunsom, Phil, and Goldwater, Sharon. Inducing tree-substitution grammars. *JMLR*.

Dayan, Peter, Hinton, Geoffrey E, Neal, Radford M, and Zemel, Richard S. The helmholtz machine. *Neural computation*, 7(5):889–904, 1995.

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

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

Devlin, Jacob, Uesato, Jonathan, Bhupatiraju, Surya, Singh, Rishabh, Mohamed, Abdel-rahman, and Kohli, Pushmeet. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*, 2017b.

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

Ellis, Kevin, Ritchie, Daniel, Solar-Lezama, Armando, and Tenenbaum, Joshua B. Learning to infer graphics programs from hand-drawn images. *arXiv preprint arXiv:1707.09627*, 2017.

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

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

Henderson, Robert John. Cumulative learning in the lambda calculus.

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

Johnson, Justin, Hariharan, Bharath, van der Maaten, Laurens, Fei-Fei, Li, Zitnick, C Lawrence, and Girshick, Ross. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pp. 1988–1997. IEEE, 2017.

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

Koza, John R. *Genetic programming - on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, 1993. ISBN 978-0-262-11170-6.

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

Lau, Tessa. *Programming by demonstration: a machine learning approach*. PhD thesis, 2001.

Le, Tuan Anh, Baydin, Atlm Gne, and Wood, Frank. Inference Compilation and Universal Probabilistic Programming. In *AISTATS*, 2017.

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

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

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

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

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

Osera, Peter-Michael and Zdancewic, Steve. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, volume 50, pp. 619–630. ACM, 2015.

Pierce, Benjamin C. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

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

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

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

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

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