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

Anonymous Authors[1]

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) embedded in Microsoft 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). These systems work in different ways, but most hinge upon 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 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, or to synthesize large bodies of code 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 people solve relatively quickly, once they acquire the relevant domain expertise. These correspond to tasks solved by short programs — if you have an expressive DSL.

[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.

2. Overview of DREAMCODER

We take 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 splitting on characters, but also specialize these routines to split on frequent delimiters such as spaces or commas. 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.

Our algorithm is called DREAMCODER because it is based on a novel kind of "wake-sleep" learning (c.f. (Hinton et al., 1995)), iterating between "wake" and "sleep" phases to achieve three goals: finding programs that solve tasks; creating a DSL by discovering and reusing domain-specific subroutines; and training a neural network that efficiently guides search for programs in the DSL. 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

	List Functions	Text Editing	Symbolic Regression

Programs & Tasks	`[7 2 3]→[7 3]`
	`[1 2 3 4]→[3 4]`
	`[4 3 2 1]→[4 3]`
	$f(\ell)$ = `(`f_1 ℓ `(λ (x) (> x 2)))`
	
	`[2 7 8 1]→8`
	`[3 19 14]→19`
	$f(\ell)$ = `(`f_2 ℓ`)`
	
	`[7 3]→False`
	`[3]→False`
	`[9 0 0]→True`
	`[0]→True`
	`[0 7 3]→True`
	$f(\ell)$ = `(`f_3 ℓ `0)`
		+106 769-438→106.769.438
		+83 973-831→83.973.831
		f(s) = `(`f_0 `"." "-" (`f_0 `"." " " (cdr s)))`
		
		Temple Anna H →TAH
		Lara Gregori→LG
		f(s) = `(`f_2 s`)`
			 f(x) = `(`f_1 x`)`
			  f(x) = `(`f_6 x`)`
			  f(x) = `(`f_4 x`)`
			  f(x) = `(`f_3 x`)`

DSL	$f_1(\ell,\text{p})$ = `(foldr` ℓ `nil (λ (x a) (if (p x) (cons x a) a)))` (f_1: *Higher-order filter function*) $f_2(\ell)$ = `(foldr` ℓ `0 (λ (x a) (if (> a x) a x)))` (f_2: *Maximum element in list* ℓ) $f_3(\ell,\text{k})$ = `(foldr` ℓ `(is-nil` ℓ`) (λ (x a) (if a a (= k x))))` (f_3: *Whether* ℓ *contains* k)	f_0(s,a,b) = `(map (λ (x) (if (= x a) b x))` s`)` (f_0: *Performs character substitution*) f_1(s,c) = `(foldr s s (λ (x a) (cdr (if (= c x) s a))))` (f_1: *Drop characters from* s *until* c *reached*) f_2(s) = `(unfold s is-nil car (λ (z) (`f_1 z `" ")))` (f_2: *Abbreviates a sequence of words*)	f_0(x) = `(+ x real)` f_1(x) = `(`f_0 `(* real x))` f_2(x) = `(`f_1 `(* x (`f_0 x`)))` f_3(x) = `(`f_0 `(* x (`f_2 x`)))` f_4(x) = `(`f_0 `(* x (`f_3 x`)))` (f_4: *4th order polynomial*) f_5(x) = `(/ real x)` f_6(x) = `(`f_4 `(`f_0 x`))` (f_6: *rational function*)

Table 1: Top: Tasks from three domains we apply our algorithm to, 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 under List Functions)

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.

Concretely, our algorithm iterates through three steps. The **Wake** 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 **Sleep-G** step grows the DSL (or **G**enerative model) which allows the agent to more compactly write programs in the domain; it modifies the structure of the DSL by discovering regularities across programs found during waking, compressing them to distill out common code fragments across successful programs. The **Sleep-R** step improves the search procedure by training a neural network (the **R**ecognition model) to write programs in the current DSL, in the spirit of "amortized" or "compiled" inference (Le et al., 2017).

Prior work on program learning has largely assumed a fixed, hand-engineered DSL, both in classic symbolic program learning approaches (e.g., Metagol: (Muggleton et al., 2015), FlashFill: (Gulwani, 2011)), neural approaches (e.g., RobustFill: (Devlin et al., 2017b)), and hybrids of neural and symbolic methods (e.g., Neural-guided deductive search: (Kalyan et al., 2018), DeepCoder: (Balog et al., 2016)). A notable exception is the EC algorithm (Dechter et al., 2013), which also learns a library of subroutines. We find EC motivating, and go beyond it and other prior work through the following contributions:

Contributions. (1) We show how to learn-to-learn programs in an expressive Lisp-like programming language, including conditionals, variables, and higher-order recursive functions; (2) We give an algorithm for learning DSLs, built on a formalism known as Fragment Grammars (O'Donnell, 2015); and (3) We give a hierarchical Bayesian framing enabling joint inference of the DSL and recognition model.

3. The DREAMCODER Algorithm

DREAMCODER takes as input a set of *tasks*, written X, each of which is a program synthesis problem. It has at its disposal a domain-specific *likelihood model*, written $\mathbb{P}[x|p]$, which scores the likelihood of a task $x \in X$ given a program p. Its goal is to solve each of the tasks by writing a program, and also to infer a DSL, written $\mathcal{D}$, which is a generative model over programs, written $\mathbb{P}[p|\mathcal{D}]$. We frame our goal as maximum a posteriori (MAP) inference of $\mathcal{D}$ given X:

$$\arg\max_{\mathcal{D}} \mathbb{P}[\mathcal{D}] \prod_{x \in X} \sum_{p} \mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D}]$$

Even evaluating the above objective is intractable due to the sum over all programs, and so we instead marginalize over finite sets of programs, called **frontiers**, one for each task $x \in X$, written $\mathcal{F}_x$, giving the following lower bound on the marginal, which we call $\mathscr{L}$:

$$\mathscr{L} \triangleq \mathbb{P}[\mathcal{D}] \prod_{x \in X} \sum_{p \in \mathcal{F}_x} \mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D}] \qquad (1)$$

We alternate maximization w.r.t. $\{\mathcal{F}_x\}_{x \in X}$ (**Wake**) and $\mathcal{D}$ (**Sleep-G**):

Wake: Maxing $\mathscr{L}$ w.r.t. the programs. Here $\mathcal{D}$ is fixed and we want to find new programs to add to the frontiers

Name	Input	Output
repeat-2	[7 0]	[7 0 7 0]
rotate-2	[8 14 1 9]	[1 9 8 14]
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.

so that $\mathscr{L}$ increases the most. $\mathscr{L}$ most increases by finding programs where $\mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D}]$ is large. Here we use a simple and generic enumerative program synthesis algorithm.

Sleep-G: Maxing $\mathscr{L}$ w.r.t. the DSL. Here $\{\mathcal{F}_x\}_{x\in X}$ is held fixed, and so we can evaluate $\mathscr{L}$. Now the problem is that of searching the discrete space of DSLs and finding one maximizing $\mathcal{D}$. To represent DSLs and search the space of DSLs, we use Fragment Grammars (O'Donnell, 2015).

Searching for programs is hard because of the large combinatorial search space. We ease this difficulty by training a neural recognition model, $q(\cdot|\cdot)$, during the **Sleep-R** phase: q is trained to approximate the posterior over programs, $q(p|x) \approx \mathbb{P}[p|x,\mathcal{D}] \propto \mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D}]$, thus amortizing the cost of finding programs with high posterior probability.

Sleep-R: tractably maxing $\mathscr{L}$ w.r.t. the frontiers. Here we train $q(p|x)$ to assign high probability to programs p where $\mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D}]$ is large, because including those programs in the frontiers will most increase $\mathscr{L}$. Concretely, the loss for $q(\cdot|x)$ is:

$$\mathbb{E}_X \left[\mathrm{KL}\left(\mathbb{P}[\cdot|x,\mathcal{D}] \;||\; q(\cdot|x)\right)\right]$$

which minimizes the KL between the distribution predicted by q and the true posterior over programs solving task x.

4. Programs that manipulate sequences

We apply DREAMCODER to list processing (Section 4.1) and text editing (Section 4.2), using a bidirectional GRU (Cho et al., 2014) for the recognition model, and initially providing the system with generic sequence manipulation primitives: `foldr`, `unfold`, `if`, `map`, `length`, `index`, `=`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`.

4.1. 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 also perform arithmetic operations upon lists (Table 2). We created 236 human-interpretable list manipulation tasks, each with 15 input/output examples. In solving these tasks, the system composed 38 new subroutines, and rediscovered the higher-order function `filter` (Tbl. 1, left).

4.2. Text Editing

Synthesizing programs that edit text is a classic problem in the programming languages and AI literatures (Lau, 2001), and algorithms that learn text editing programs ship in Microsoft Excel (Gulwani, 2011). This prior work presumes a hand-engineered DSL. Here we will 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.

We trained our system on 109 synthetic tasks, with 4 input/output examples each. After three iterations, it assembles a DSL containing a dozen new functions (Fig. 1, center) solving all 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*.Here we learned a single DSL that applied generically to all of the tasks, and perform comparably to the best prior work.

5. 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 or rational function. The recognition model is a convnet that observes an image of the target function's graph (Fig. 1) — visually, different kinds of polynomials and rational functions produce different kinds of graphs, and so the recognition model can look at a graph and predict what kind of function best explains it. These programs can contain real numbers, which we set with gradient descent, and penalize continuous parameters by incorporating a BIC penalty into the likelihood model $\mathbb{P}[x|p]$. We learn a DSL containing 13 new functions, mainly templates for polynomials of different orders or ratios of polynomials. It also learns to find programs minimizing the number of continuous parameters — learning to represent linear functions with `(* real (+ x real))`, which has two continuous parameters, and represents quartic functions using f_4 in the rightmost column of Fig. 1 which has five continuous parameters. This phenomenon arises from our Bayesian framing.

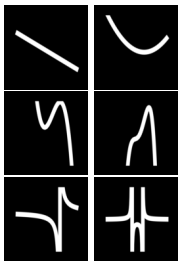

Figure 1: Recognition model input for symbolic regression. DSL learns subroutines for polynomials (top rows) and rational functions (bottom) while the recognition model jointly learns to look at a graph of the function (left) and predict which learned subroutines best explain the observation.

6. Learning from Scratch

A long-standing dream within the program induction community is "learning from scratch": starting with a *minimal* Turing-complete programming language, and then learning to solve a wide swath of induction problems (Solomonoff, 1964). All existing systems, including ours, fall far short of this dream, and it is unclear (and we believe unlikely) that this dream could ever be realized in its fullness. How far can we in this direction? "Learning from scratch" is subjective, but a reasonable starting point is the set of primitives provided in 1959 Lisp (McCarthy, 1960): these include conditionals, recursion, arithmetic, and the list operators `cons`, `car`, `cdr`, and `nil`. A basic first goal is to start with these primitives, and then recover a DSL that more closely resembles modern functional languages like Haskell and OCaml. Recall (Sec. 4) that we initially provided our system with functional programming routines like `map` and `fold`.

We ran the following experiment: DREAMCODER was provided with a subset of the 1959 Lisp primitives, and tasked with solving 22 functional programming exercises. After running for 93 hours on 64 CPUs, our algorithm solves these exercises, along the way assembling a DSL with a modern repertoire of functional programming idioms and subroutines, including `map`, `fold`, `zip`, `unfold`, `index`, `length`, and simple arithmetic operations like building lists of natural numbers between an interval.

To be clear, we believe that program learners should *not* start from scratch, but instead should start from a rich, domain-agnostic basis like those embodied in the standard libraries of modern programming languages. What this experiment shows is that DREAMCODER doesn't *need* to start from a rich basis, and can in principle recover many of the amenities of modern programming systems, provided it is given enough computational power and a suitable spectrum of programming exercises.

7. Learning Generative Models

We apply DREAMCODER to learning generative models for images (Figure 2) and text (Figure **??**). For images, we learn programs in a LOGO/Turtle-like representation, and the task is to look at an image and explain it in terms of a graphics program. For text, we learn probabilistic regular expressions – a simple probabilistic program for which inference is always tractable – and the task is to infer a regex from a set of strings.

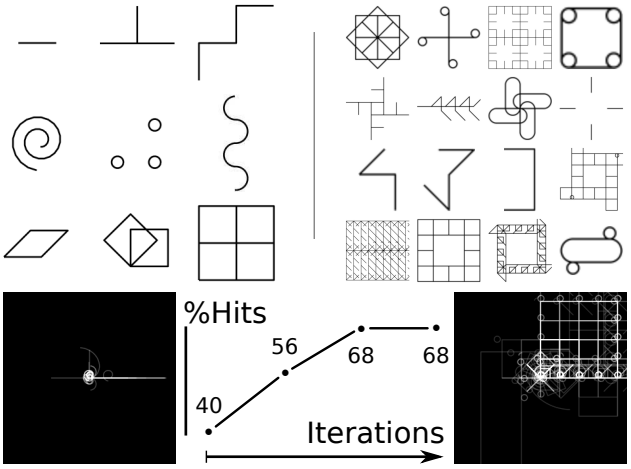

Figure 2: Top left: Example training tasks. Top right: samples from the learned DSL. Bottom: % holdout testing tasks solved (middle); on the sides are averaged samples from the DSL before any training (left) and after last iteration (right).

8. Quantitative Results

We evaluate on held-out testing tasks, measuring how many tasks are solved and how long it takes to solve them 3. Prior to any learning, the system cannot find solutions for most of the tasks, and those it does solve take a long time; with more wake/sleep iterations, we converge upon DSLs and recognition models that more closely match the domains.

9. 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 — can help 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. 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?

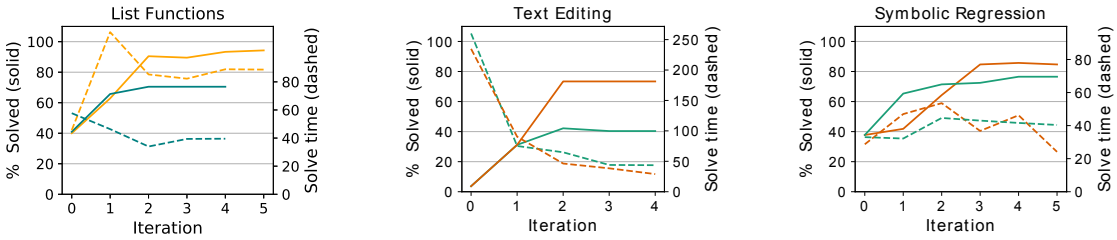

Figure 3: Learning curves for DREAMCODER both with (in orange) and without (in teal) the recognition model. Solid lines: % holdout testing tasks solved w/ 10m timeout. Dashed lines: Average solve time, averaged only over tasks that are solved.

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.

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.

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.

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.

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

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.

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

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