

**On the Representation and Learning of Concepts:
Programs, Types, and Bayes**

by

Lucas Eduardo Morales

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2018

© Massachusetts Institute of Technology 2018. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
September 2018

Certified by
Joshua B. Tenenbaum
Professor
Thesis Supervisor

Accepted by
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

**On the Representation and Learning of Concepts:
Programs, Types, and Bayes**

by

Lucas Eduardo Morales

Submitted to the Department of Electrical Engineering and Computer Science
on September 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis develops computational models of cognition with a focus on concept representation and learning. We start with brief philosophical discourse accompanied by empirical findings and theories from developmental science. We review many formal foundations of computation as well as modern approaches to the problem of *program induction* — the learning of structure within those representations. We show our own research on program induction focused on its application for language bootstrapping. We then demonstrate our approach for augmenting a class of machine learning algorithms to enable domain-general learning by applying it to a program induction algorithm. Finally, we present our own computational account of concepts and cognition.

Thesis Supervisor: Joshua B. Tenenbaum
Title: Professor

Acknowledgments

This thesis would not exist if not for my advisor Josh Tenenbaum, who helped bring me into fascination with understanding how people learn and think.

Much of this work was supported by the MIT-IBM Watson AI Lab for their investment in program induction research.

My collaborators Kevin Ellis and Josh Rule had vital roles in turning interesting ideas into real research projects and experiments. They also tolerated and even accepted my obsession with the Rust programming language. (Along those lines, I am also grateful to Mozilla for providing such a powerful programming tool.)

I thank the many folks around the CoCoSci space with whom I had a variety of interesting conversations: Luke Hewitt, Max Nye, Mario Belledonne, Max Siegel, Nathalie Fernandez, Michael Chang, Felix Sosa, and Amir Soltani. These conversations helped make the lab a space where I felt some belonging, as well as promoted many of the thoughts that have been incorporated into this thesis.

I thank my friends for giving me opportunities to engage in many of my beloved hobbies: philosophizing, teaching, optimizing, etc.

My family played a vital role in giving me the liberty to pursue my own interests from a young age while keeping me engaged through countless conversations of politics, economics, and comedic banter.

I have the deepest of appreciation for the many philosophers and scientists who, in the name of understanding, have created the scientific knowledge and practice in which I have found home. I am standing on the shoulders of giants and humbled by everything that I can observe.

Contents

1	Introduction	19
2	Computational Framing of Representation and Learning	21
3	Representations	25
3.1	Overview	25
3.2	Neural networks	26
3.3	Context-free grammars	29
3.4	Combinatory logic	31
3.4.1	Polymorphic typed combinatory logic	33
3.5	Lambda-calculus	35
3.5.1	Simply typed lambda-calculus	38
3.5.2	Polymorphic typed lambda-calculus	38
3.6	Term rewriting	40
3.6.1	Polymorphic typed term rewriting	43
3.6.2	Higher-order rewriting	44
3.7	Pure type systems	48
4	Concept Learning by Design: Program Induction	57
4.1	Overview	57
4.2	Learning tasks	58
4.3	Combinatorial explosion	61
4.4	Learning architectures	62
4.4.1	Overview	62
4.4.2	Enumerative search	62

4.4.3	Constraint satisfaction	64
4.4.4	Deductive search	66
4.4.5	Inductive search	67
4.4.6	Inductive logic programming	68
4.4.7	Markov chain Monte Carlo	70
4.4.8	Genetic programming	73
4.4.9	Neural network optimization	74
4.5	Bayesian program learning	76
4.6	Language bootstrapping	78
4.6.1	Introduction	78
4.6.2	Related work	80
4.6.3	DREAMCODER	81
4.6.4	Experiments with DREAMCODER	92
4.6.5	DREAMCODER: conclusion and future work	96
4.7	Domain-general learning	98
4.7.1	Introduction	98
4.7.2	Related work	99
4.7.3	CONTEXTNET	102
4.7.4	Experiment: CONTEXTNET atop EC	105
4.7.5	CONTEXTNET: conclusion and future work	111
5	Towards Formalized Conceptual Role	113
5.1	Introduction	113
5.2	Representation and role	114
5.3	Learning and conceptual change	118
5.3.1	Synthesis	119
5.3.2	Consolidation and abstraction	121
5.3.3	Conceptual change with discontinuity	122
5.3.4	Construction of concepts	125
5.3.5	Inducing search procedures and probabilistic models	127
5.4	Perception, memory, and cognition	128
5.5	Discussion and Future Work	132

List of Figures

3-1	A multilayer perceptron with three layers. Weights for multiplication are assumed in edges leading into summation.	27
3-2	Common differentiable nonlinearities. (a) The sigmoid logistic function acts like an “on/off” switch for the neuron. (b) The hyperbolic tangent function is like a “good/bad” switch because it sends negative signal where the sigmoid gives no signal. (c) The rectified linear unit has an “off” state like the sigmoid, but provides almost-linear activation when there is sufficiently large input. .	28
3-3	A recurrent neural network.	29
3-4	A language over all strings with an odd number of 1s. (a) A finite-state automaton which accepts strings in this language; (b) the regular grammar which produces sentences in this languages.	30
3-5	(a) A fragment of the English language as a CFG, and (b) the parse tree for a production in that language. The acronyms of each nonterminal in order of rule definition are: Sentence, Noun Phrase, Verb Phrase, Prepositional Phrase, Complex Noun, Complex Verb, Article, Noun, Verb, Preposition. .	30
3-6	An example of computing with combinatory logic: Boolean logic. The symbol definitions are <i>not</i> a feature of combinatory logic: their purpose here is to illustrate a way combinators may be used to provide meaningful computation. <i>A</i> and <i>B</i> are illustrative placeholders for combinators. This is built atop the BCIKS system of Figure 3-7.	32
3-7	The Schönfinkel BCIKS system of combinatory logic. These rules describe how to perform reduction of a combinator.	32

3-8	Binary tree representations for $f(x) = x^2 + 1$, corresponding to the combinatory logic expression $(\mathbf{C}(\mathbf{B} + (\mathbf{S} * \mathbf{I}))1)$: (a) with primitive combinators as node labels and arrows to visualize the path of an input; (b) primitive combinators as terminals; (c) reduction of the combinator when supplied argument x	33
3-9	Type schemas for the Schönfinkel BCIKS combinators.	35
3-10	A procedure in Scheme, a dialect of LISP, based on the abstractions provided by λ -calculus. Here we've defined a function for the greatest common denominator (<code>gcd</code>) using <code>(define (gcd a b) (...))</code> . This corresponds to the expression in λ -calculus of $(\lambda G \langle \text{rest} \rangle)(\lambda a(\lambda b(\dots)))$ where $\langle \text{rest} \rangle$ is an expression that may use G as a procedure for <code>gcd</code>	36
3-11	Arithmetic in pure λ -calculus. Arrows represent encoding. Bold lowercase letters starting from a denote unused variable names.	37
3-12	de Bruijn index notation. Colors and arrows indicate distinct variables and the abstractions which introduce them.	38
3-13	List operations in polymorphic λ -calculus. (a) Types for various list operations; (b) definition of the map primitive in λ -calculus, in standard syntax and with omitted parentheses, given <code>fold</code> , <code>cons</code> , and <code>nil</code> ; (c) definition of map in the Lisp programming language.	39
3-14	(a) Functions in Haskell for determining whether a natural number is even or odd. These could be viewed as procedure calls, however they may instead be viewed as rewrites to a term. It is trivial to convert this Haskell code into a term rewriting system. (The fourth rule, which decrements a natural number in the right-hand side, may instead take the successor of a natural number on the left-hand side so the rewrite system does not have to implement subtraction). (b) The rewriting of a term in this system.	40
3-15	Term rewriting system for (a) arithmetic on natural numbers and (b) combinatory logic where \circ is the symbol for application of two combinators.	41
3-16	(a,b) Term trees under the arithmetic TRS of Figure 3-15a. (c) A tree diagram showing positions associated with a term tree.	42
3-17	A reduction of $1 + 2 \times 1$ in the arithmetic TRS of Figure 3-15a. This is not the only reduction from $A(S(Z), M(S(S(Z)), S(Z)))$ to $S(S(S(Z)))$	43

3-18 (a) A term rewriting systems for lists and natural numbers and (b) the type of each symbol in its signature.	44
3-19 A polymorphic combinatory reduction system to demonstrate the map list operation. For legibility, symbols are written in blue, meta-variables in orange, and variables in red.	45
3-20 A reduction of an applied map operation for doubling onto a list of numbers [1, 2] in the PCRS of Figure 3-19.	47
3-21 (a) A schematic of conceptual relations that are representable in a declarative manner. It is straightforward to convert this schematic as illustrated here into types of a pure type system. (b) Sort defined with a dependent type: given a type T which is orderable (signified by the constraint <code>Ord T</code>), the procedure's input i is a value of type <code>List T</code> , and the output type depends on the input value — it is the set of values v of type <code>List T</code> that satisfy the conditions that the elements of v are equivalent to the elements of i and that v is a non-decreasing list.	49
3-22 The Barendregt λ -cube with labeled axes on the right [Barendregt, 1991]. Each vertex is an enhancement from λ_{\rightarrow} (front-bottom-left), the simply-typed λ -calculus we described in Section 3.5.1 Simply typed lambda-calculus. System F (front-top-left) is polymorphic λ -calculus, a variant of which we presented in Section 3.5.2 Polymorphic typed lambda-calculus. The calculus of constructions (<i>CC</i> ; Coquand and Huet [1986]) has all three features indicated by the three axes of the cube: polymorphism, type operators, and dependent types.	50
3-23 Product rules for the pure type system of each corner of the λ -cube.	53
3-24 Product rules for the propositional logics, where λPROP , $\lambda\text{PROP}2$, and $\lambda\text{PROP}\omega$ are isomorphic to first-, second-, and higher-order propositional logic, respectively.	54
3-25 Product rules for predicate logics, where λPRED and $\lambda\text{PRED}\omega$ are isomorphic to first-order and higher-order predicate logic, respectively.	54
3-26 λHOL , a pure type system isomorphic to constructive higher-order logic.	54

4-1	Learning from examples. (a) String transformation problems as present in spreadsheets [Gulwani, 2011]. Noisy cases of string transformation problems have also been explored [Devlin et al., 2017b]. The string transformation problems are typically presented as program <i>induction</i> problems, where one only needs to compute outputs given new inputs directly rather than synthesize a program. (b) Analogy-making, a form of one-shot learning that is based on program <i>induction</i> , as it need not explain the analogy to complete the problem [Hofstadter and Mitchell, 1995]. (c) The Bongard problems provide positive and negative examples for a program which discriminates objects in the left-hand-side from the right-hand-side [Bongard, 1970]. This is a program <i>synthesis</i> problem, as the solution is intended to be succinctly explainable.	58
4-2	(a) A context-free grammar for addition, in which we can enumerate numbers. (b) a more sophisticated grammar over the same space, designed for more efficient enumeration by breaking symmetries: (i) a production of <i>Plus</i> cannot have 0 as one of its children and (ii) a production of <i>Plus</i> cannot have another <i>Plus</i> production as its left child (only as its right child). (c) Enumerating from the simple grammar. (d) Enumerating from the sophisticated symmetry-breaking grammar.	63
4-3	Type-directed synthesis of <code>sort</code> . (a) The task is to find an inhabitant of the <code>sort</code> type. Terms of the form $\{\tau P(v)\}$ refer to a <i>refined</i> type, whose inhabitants are every value v of type τ that satisfies the predicate $P(v)$. (b) The program synthesized by SYNQUID [Polikarpova et al., 2016].	67
4-4	λ^2 learns the function <code>dropmins</code> from four examples [Feser et al., 2015]. .	68
4-5	The target concept represents the <i>grandfather</i> relation. Lowercase letters a, b, \dots represent people. \mathcal{B} is the background knowledge, \mathcal{P} is the set of positive examples, and all other ground atoms involving the target concepts are the negative examples. The clauses on the bottom are learned. The “invented” predicate may be invented to provide a more general rule for the target.	69

4-6	A mutually recursive procedure for the <i>even</i> relation, learned with meta-interpretive learning. The “invented1” predicate corresponds to the <i>predecessor</i> relation, and the “invented2” predicate corresponds with the <i>odd</i> relation.	70
4-7	Demonstration of MCMC search with the Metropolis-Hastings algorithm, plotted in the hypothesis space. Scatter points are sampled hypotheses. The line indicates progression in the chain: purple refers to the beginning while yellow refers to the end. It is probable that after many iterations, these samples approximate the true posterior.	71
4-8	Genetic programming. Genesis is the creation of an initial population. Offspring are produced by performing genetic operators on the population. Through natural selection, the new population is determined.	73
4-9	Learning landscape for a differentiable function, where lateral axes correspond to the parameter space (i.e. values of θ) and the height corresponds to the cost $J(\theta)$. The arrow indicates a step of stochastic gradient descent towards a lower-cost point in the space by following $-\nabla_{\theta}J(\theta)$	75
4-10	Overfitting: the orange curve overfits the data, while the blue curve is regularized.	76
4-11	(a) DSL \mathcal{D} generates programs p by sampling DSL primitives according to probabilities θ . From each program we observe a task x (program inputs/outputs). (b) Neural network $q(\cdot)$, the <i>recognition model</i> , regresses from x to the parameters of a distribution over programs, $\theta^{(x)}$. Black/red arrows correspond to the generative/recognition models.	79
4-12	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 <code>filter</code> (f_1 in List Functions)	80
4-13	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.	82

4-14	Left: syntax trees of two programs sharing common structure, highlighted in orange, from which we extract a fragment and add it to the DSL (bottom). Right: actual programs, from which we extract fragments that (top) slice from the beginning of a list or (bottom) perform character substitutions. . . .	88
4-15	Some tasks in our list function domain. See the supplement for the complete data set.	92
4-16	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.	94
4-17	Percentage of held-out test tasks solved. Solve time: averaged over solved tasks. (*: This particular experiment was done <i>without</i> Helmholtz-style sampling for training the recognition model.)	95
4-18	Learning curves for DREAMCODER both with (in orange) and without (in teal) the recognition model. Solid lines: percentage of held-out testing tasks solved; dashed lines: average solve time.	96
4-19	Schematic of the function of CONTEXTNET. The environment constitutes everything that the system observes from the “outside world.” The context, derived from the CONTEXTNET, defines all parameters for the mechanism. When the mechanism learns something new, it passes that information to the context which mutates the CONTEXTNET to incorporate that new knowledge.	102
4-20	Methods on an instance of the knowledge network.	102
4-21	The first pane shows a knowledge network, and a shaded region representing the active context. The second pane shows the memory accesses within the context leading up to the addition of a new knowledge artifact. The third pane shows the new knowledge network, with the additional knowledge artifact nondeterministically attached according to the distribution of memory accesses.	103

4-22	A knowledge network after learning programs with the EC algorithm. For demonstration purposes, each node displays the id corresponding to the phase at which the artifact was learned, and an example of a task from that phase. Actual knowledge artifacts contain learned combinators, not the content shown here. Older artifacts, towards the top, have high connectivity because they contain concepts that are useful.	106
4-23	Primitives for string transformation given to the EC algorithm, and their associated types. See Section 3.4.1 Polymorphic typed combinatory logic for an explanation of the syntax used here for types.	107
4-24	Complete list of string transformation tasks that we teach the EC algorithm to solve compositionally. Not all subdomains have more than one phasic task set. Some tasks are repeated to enable non-contextual learning to successfully complete all phasic task sets.	108
4-25	Solutions for a particular phasic task set made by both the specialized and contextual grammars, displayed in lambda calculus.	108
4-26	Tasks are automatically ordered based on performance in each metric. (a) The speed at which phases were completed, shown for each task. (b) The speed at which tasks were solved in the final iteration of the EC algorithm. (c) The log likelihood of solutions for tasks conditioned on the grammar.	110
4-27	Schematic of the function of CONTEXTNET with desired future features, adopted from Figure 4-19. Green lines indicate connections for a neural network tasked with context localization. The “mechanism” is generalized to any number of learners, indicated with blue lines. Actors which interact with the environment are indicated with orange lines.	112
5-1	Programmer as translator. A programmer is tasked with translating mental models (left) into a programming language (right), and vice-versa when interpreting foreign models.	115

5-2 Natural language and types. Both provide declarative means of descriptive communication, and they may correspond in obvious ways as illustrated here. “Sort” leverages existing concepts like quantification (“things;” polymorphism over T), “orderable” (`Ord`), “sequence” (`List`), order-independent equivalence between containers (“of those things;” `elems(i) = elems(v)`), ordered sequences (“in order;” `nondecreasing(v)`), and conjunction (implicit in natural language; `&`). 116

5-3 Causation modeled in a type system. We use orange to indicate a set of types (i.e. a set of concepts), SMALL-CAPS to indicate a type (i.e. a concept), green to indicate a type variable (i.e. a concept variable), blue to indicate an instance variable (i.e. an inhabitant of a concept), red for presumed procedures which may be innate or defined elsewhere, and purple for defined procedures. Square brackets indicate “indexing” into existentially quantified variables, the details of which are beyond the scope of this work. In words, a concept is an *event* if there is an environment and a function on that environment which determines whether the concept is observed. Event a *causes* event b (a proposition `PROP`) if there is an intermediary event c such that a causes c and c causes b , or if there is a nontrivial perturbation on the environment where a change to the distribution of event a implies a change to the distribution of event b and the *counterfactual*, that no change for a implies no change for b . This could be modeled differently, such as with metrics of comparison between distributions, but our illustration suffices. To illustrate *placeholder* capability for causation, one only needs to introduce disjunctive statements to the system. For example, “cause `SomeConcept SomeOtherConcept`” alone suffices to indicate another inhabitant of “cause” without necessitating use of its abstract definition. 117

- 5-4 A physical system. The pseudocode in (b) is mostly object-level and not type-level. These statements form *constraints* which determine the nature of `ball`, the underspecified object. When an object for `ball` is discovered by synthesis, an inhabitant for the final statement (which is not a constraint because it lacks assignment) may be synthesized. The *priors* here the implicit values that there is acceleration due to gravity on all objects and that velocity is zero unless otherwise specified. These priors are established as a function of perception, as discussed in Section 5.4 Perception, memory, and cognition. 120

- 5-5 In the work of Section 4.6 Language bootstrapping, we “compressed” common code in (a) by creating reusable helper functions as in (b), making empirically relevant concepts more accessible for future learning. We note that the arithmetic and natural numbers assumed in this example may be learned, and we remark that as discussed earlier in this chapter, an item like “two” can exist as both a type (i.e. concept) or as an object (i.e. something that inhabits a concept). 121

- 5-6 Multiplication of 78 by 3 using (a) repeated addition and (b) the lattice method. These multiplication algorithms *are* commensurable as the latter has an apparent logical construction derived from the former. 123

- 5-7 Conceptual change as *type-level refactoring* from a representation of natural numbers that is (a) incremental, as in Peano arithmetic, to (b) digital, as in the Arabic numeral system. Not shown are the corresponding implementations of addition (which is more efficient in the digital system for large numbers), number-word generation (e.g. speaking “twenty”), and other already-established numerical operations. This code is written more concretely in the Rust programming language because we think it is more illustrative and expressive for this example. 123

5-8	Repeated phyllotactic spirals according to the Fibonacci sequence, a natural phenomenon demonstrated in sunflowers, pineapples, and other plant-life. The number of petals in a flower is generally a Fibonacci number (e.g. the lily flower has three petals, often repeated twice, and the buttercup typically has five petals). When the petals repeat, they form two repeated spirals (clockwise and counter-clockwise) each of which occurs a Fibonacci number of times.	124
5-9	Construction of the RED-DOG concept in a pure type system. Note that while universal quantification is a feature of pure type systems, existential quantification is not. However, it is nonetheless constructable as proven by Geuvers [1993].	126
5-10	Construction of an interpretation of the STRANGE-LOOP concept, a thesis of Hofstadter [1979]. Here we assume the conjunction concept defined in Figure 5-9.	126
5-11	A placeholder-filling construction for the concept of Darwinian evolution. The first iteration provides a new symbol. The second iteration introduces some notion of procreation (particularly sexual reproduction). The third iteration further specifies that reproduction results in <i>mutation</i> . The fourth iteration produces a preliminary idea of evolution, and the fifth iteration introduces something external to the organisms (i.e. an environment) as a factor in that process. We use $[X]$ to denote collections of inhabitants of a particular X	127
5-12	Descriptions whose lengths determine three measures of conceptual complexity: (a) type-level, (b) object-level, and (c) runtime-level. If this examples co-occurs with a representation like that of Figure 5-7b, measurements of these complexities may demonstrate that it is an inferior representation and would be replaced. (Notably, addition is runtime-logarithmic in a digital representation whereas this incremental representation is runtime-linear.) .	131

Chapter 1

Introduction

Our abilities to make sense of our experiences come from the powerful system of the human mind, enabling us to identify causation, learn concepts, and acquire language. Reverse-engineering the mind is an old problem with computational accounts arising as early as [Boole \[1854\]](#). We are in the privileged position of being fundamentally exposed to some aspects of the mind: primarily, *thought*. Once identified, thought begets the question of *what is thought?* A common direction begins with the notion of *concepts*, the units that constitute thought. Acknowledging concepts then circularly begs the question, *what are concepts?* While the question of the intrinsic nature of concepts is daunting and controversial, there is consensus at the surface: whatever they may be, concepts (and hence, thoughts) are fundamental to several psychological processes including the production of ideas, the storing and recall of memory, decision-making, use of language, and learning.

In this thesis, we adopt the computational view of mind, accepting that there is a fundamental similarity between computation and cognition. An obvious consequence of this is that there should be some analogue between the production of thoughts and the production of computational entities. There are many ways to take this direction of pursuit, particularly those outlined by the levels of analysis from [Marr \[1982\]](#): (I) the computational level, which characterizes a problem with inputs in some environment and a method for computing the solution; (II) the algorithmic level, which describes more concretely the representations and algorithms that arrive at such a solution; and (III) the implementation level, which specifies how such representations and algorithms are presented in the hardware of a brain or machine.

We take focus on the computational and algorithmic levels of analysis. Of pique interest is the problem of *learning*. This has been explored by philosophers for millennia, but only until recently with the advent of computing technology have the doors been opened for science to regard learning as a class of computational problems and people as natural computers that have evolved to solve them. Artificial intelligence, a field born from this computational hypothesis, was popularized by the question of Turing [1950]: “Can machines think?” Turing expresses that rather than building machines that can simulate an adult human, an apparently insurmountable task, the simulation of a child would yield a machine that could learn and effectively *become* an adult-level thinker. Our efforts aim to make sense of how people — especially children — learn concepts and produce thought.

In [Chapter 2](#), we present the problems of concept representation and concept learning in a computational framing and briefly discuss theories of concepts justified by developmental science to provide some empirical insight for these problems. In [Chapter 3](#), we describe different formal representations that each support some intuition of what concepts are and what they can express. In [Chapter 4](#), we present *program induction* as a perspective on concept learning, outline different approaches taken to program induction primarily from the artificial intelligence and machine learning communities, and present our own work on program induction. Finally, in [Chapter 5](#), we present our own computational account of concepts and cognition based on (a) conceptual role semantics, a paradigm where the meaning of concepts is derived from their role relative to other concepts; (b) program induction; (c) types, a hierarchical and declarative perspective on computation; (d) Bayesian sampling and inference; and (e) information theory.

Chapter 2

Computational Framing of Representation and Learning

Many aspects of cognition can be combined into a unified computational framework. Such computational theories of cognition must account for both nativism and empiricism: there is *innate* faculty which enables the *learning* of rich structures from number concepts to natural language. To account for the development of adult cognition, there is some representational system which is capable of permitting such sophisticated concepts to be learned. Hence the focus on *representation* and *learning*, both of which must be understood in harmony such that each may “bootstrap” or support the other — i.e. learning representations and using representations to direct learning.

The language of thought (LOT) hypothesis proposes that thoughts are represented in the mind as productions of a compositional mental language, that causal sequences of such productions comprise mental processes, and that there is a correspondence between syntactic complexity and semantic complexity [Fodor, 1975]. The compositional nature of thought and the complexity of new thoughts have given the hypothesis much theoretical and empirical support, though it often requires assumptions on the approach to semantics [Pylyshyn, 1980; Fodor and Pylyshyn, 1988; Feldman, 2000; Fass and Feldman, 2003; Goodman et al., 2008; Kemp et al., 2008b; Kemp and Tenenbaum, 2009; Kemp, 2012; Ullman et al., 2012; Piantadosi et al., 2012, 2016]. Many criticisms of this direction appeal to *connectionism*, that symbolic operation is the apparent result of emergent latent structure from low-level continuous spaces of pattern recognition [Rumelhart and McClelland, 1986; Fahlman and

Hinton, 1987; Fodor and Pylyshyn, 1988; McClelland, 1995; Rogers and McClelland, 2004; McClelland et al., 2010]. In this thesis, we assume there is such a mental representation as posited by the LOT hypothesis, but we do not necessarily rule out connectionist approaches under which such representation is emergent. For example, Piantadosi [2016] has presented a theory on the computational origin of representation that is compatible with connectionism where symbolic computational mechanisms are emergent. Furthermore, there is no account for the origin of meaning in the LOT hypothesis. We adopt an approach to meaning called *conceptual role semantics* (c.f. Chapter 5 Towards Formalized Conceptual Role).

Science itself, as a sociological phenomenon, gives credence to the notion of mental representation. The historical analyses of Kuhn [1965] demonstrated that there is incommensurable distinction between concepts before and after scientific paradigm shifts. He expressed that “normal science,” the common practice which occurs most often (and between crises and paradigm shifts), exploits the existing theories of a scientific discipline. The scientific breakthroughs of Kepler, Darwin, and Maxwell were each triggered by observation of new phenomena that led these scientists to devise new theories and methods which became paradigms for the fields they brought forth. Each of these new constructions were built through placeholder structures which left details to be determined after an abstract framework was developed, a form of conceptual “bootstrapping” [Carey, 2009].

The cognitive characterization of science provides a means for understanding what cognitive processes may be present in children during cognitive development. A perspective called “theory theory” or “child as scientist” holds that children possess concepts as structured *theories* much like scientific conceptual structures, and that the construction and transformation of those theories determines conceptual development while semantics are derived from such developed theories [Carey, 1985; Murphy and Medin, 1985; Gopnik and Meltzoff, 1997; Gopnik and Schulz, 2004; Schulz et al., 2008; Schulz, 2012a; Ullman et al., 2012]. These *intuitive theories* have abstract structure that is coherent and appeal to causality. At the foundations of these theories is what developmental scientists call “core knowledge”: a set of separable innate systems, whose existence have been heavily supported by empirical evidence, which advance cognitive development along certain domains [Baillargeon, 1994; Spelke, 1998; Spelke and Kinzler, 2007; Carey, 2009].

Computational frameworks based on the “child as scientist” perspective have led to bet-

ter accounts of human thought by leveraging technical achievements of machine learning and Bayesian artificial intelligence [Tenenbaum et al., 2011]. A computational framework of human learning should account for the construction of causal models that can *explain* the world, rather than the recognition of patterns of observation as popularized in machine learning. By building intuitive theories with compositional structure, such a framework should model inductive biases that account for rapid acquisition and generalization of knowledge when confronted with new complex tasks [Lake et al., 2017]. Adopting a computational theory of cognition gives us the opportunity to concretely formulate key questions in cognitive science, such as:

- How are concepts individuated?
- Why are concepts useful?
- What is the representation of concepts?
- How does concept learning manifest?
- By what procedures are concepts learned?
- To what extent is the representation of concepts *learned*?
- To what extent are learning procedures for concepts *learned*?
- Where do concepts meet perception?

We attempt to answer each of these questions with our own framework presented in [Chapter 5 Towards Formalized Conceptual Role](#).

Chapter 3

Representations

3.1 Overview

In this chapter, we describe different substrates on which concepts are represented and upon which concepts are learned. Each section is devoted to a different substrate, or *representation*. We begin each section with a high-level description of the representation — what distinguishes it from others and examples of concepts constructed within it — before getting into the formalism of the representation. In this chapter, there will be little discussion of *learning* within these representations: that will be covered in the next chapter.

We start in [Section 3.2](#) with a representation that is very different from the others: artificial neural networks. The *connectionist* movement in cognitive science led to a growth of research on artificial neural networks, as they could behave intelligently without necessitating symbolic operation akin to conventional computers [[Rumelhart and McClelland, 1986](#); [Fahlman and Hinton, 1987](#); [Fodor and Pylyshyn, 1988](#)]. This approach does not postulate a “language of thought;” it instead relies on causal flow of information in continuous spaces in which pattern recognition and statistical learning may result in the *emergence* of latent structure from the sub-symbolic substrate [[Fodor, 1975](#); [McClelland, 1995](#)]. There is agreement in cognitive science and artificial intelligence that the ability to learn and represent complex structure is essential — something that connectionist approaches have, especially recently, been shown to achieve [[Gopnik and Meltzoff, 1997](#); [Rogers and McClelland, 2004](#); [McClelland et al., 2010](#); [Tenenbaum et al., 2011](#); [Schmidhuber, 2015](#); [LeCun et al., 2015](#); [Goodfellow et al., 2016](#)]. Because structured representation is, in effect, emergent under this paradigm, we analyze what the recent work has demonstrated in [Chapter 4 Concept](#)

Learning by Design: Program Induction (c.f. Section 4.4.9 Neural network optimization).

In subsequent sections, we look at symbolic representations starting with context-free grammars in Section 3.3. While context-free grammars are useful for describing the syntax of formal language, they are fundamentally insufficient for providing a general-purpose substrate upon which computation may occur. We therefore move attention towards representations that satisfy this computational expressive capability.

A universal computer is a machine that can compute anything which is computable¹. In Alan Turing’s seminal work “Computing Machinery and Intelligence,” he posits that a sufficiently-programmed universal computer could convince a person that it is human [Turing, 1950]. He further postulates that such a machine could be programmed with innate knowledge much like a child, including the faculty to learn, and that it could learn what people learn through developmental experience². We direct attention to such representations that correspond to universal computers, starting with two classical formulations: combinatory logic in Section 3.4, followed by lambda-calculus in Section 3.5. We follow these classical representations with term rewriting systems in Section 3.6. For the three of these representations, we also give exposition on *typed* variants. Types constrain search making learning more tractable³, and they provide declarative means of expressing computational properties. Finally, we look at powerful type systems in Section 3.7 which allow for unambiguous declarative expression of rich concepts while providing first-class means for finding procedural implementations of those concepts.

3.2 Neural networks

A modern discussion of representation is incomplete without mention of neural networks, and in particular, deep neural networks that are learned [Goodfellow et al., 2016]. Deep learning allows representations to be learned, where those representations are expressed in terms of simpler learned representations. For example, the multilayer perceptron of Figure 3-1 effectively produces a new representation after each layer: when trained, the

¹ A Turing machine is a universal computer, as is any machine which can mimic a Turing machine (c.f. Church-Turing thesis; Church [1936]; Turing [1937]).

² There is a caveat that real-world digital computers have finite limits, but it is relieved in the understanding that — particularly with the passage of time and the advancement of technology — these limits are not restrictive.

³ Constraining search does not necessarily imply more tractable learning — there is an assumption in this statement that learning is a product of search in the space of valid statements of these representations.

intermediate layers determine a latent representation that is not set by the designer of the network. Neural networks are usually designed to be differentiable functions so that they may be trained with gradient descent as we'll discuss in [Section 4.4.9 Neural network optimization](#).

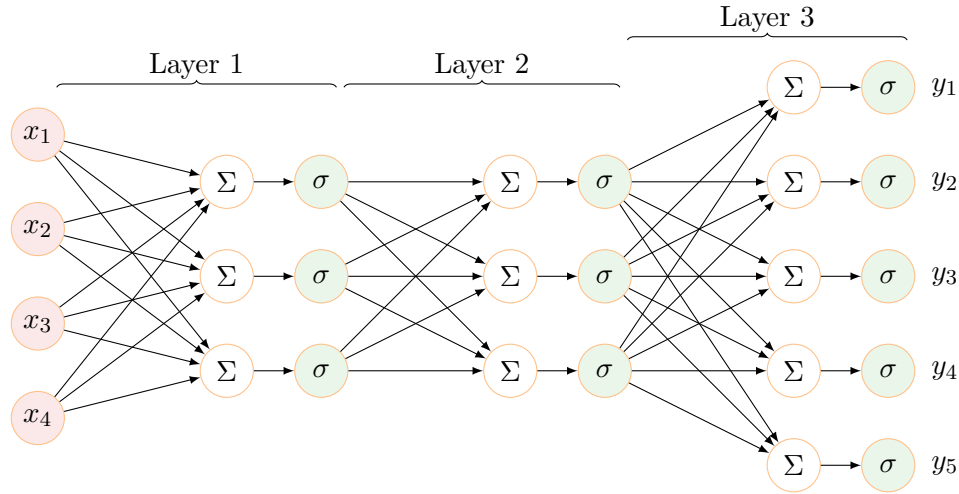


Figure 3-1: A multilayer perceptron with three layers. Weights for multiplication are assumed in edges leading into summation.

Neural networks are composed of layers of neurons, where each neuron is a pair (\mathbf{w}, σ) of weight vector \mathbf{w} and nonlinearity σ . A neuron that operates on vector \mathbf{x} computes the output $\sigma(\mathbf{w}^\top \mathbf{x})$. The nonlinearity may be thought of as a thresholding function so that a neuron's may “activate” or not depending on its input. Because a desired feature of neural networks is that they are differentiable, the nonlinearity is typically either the standard logistic sigmoid function, the hyperbolic tangent function, or the rectified linear unit (each of which is differentiable), shown in [Figure 3-2](#). A common variation of the neuron introduces a *bias* parameter, so that it computes $\sigma(\mathbf{w}^\top \mathbf{x} + b)$ ⁴.

A layer in a neural network is a set of neurons that have the same input. Typically, each neuron in a layer uses the same activation function, allowing a layer to be succinctly described by the triple $(\mathbf{W}, \mathbf{b}, \sigma)$ where \mathbf{W} is the matrix where each row is the weight vector for each neuron, \mathbf{b} is a vector comprised of the biases for each neuron, and σ is the nonlinearity. Taking $\sigma(\mathbf{v})$ to be the vector produced by elementwise transformation of \mathbf{v} by σ , a layer given input \mathbf{x} computes output $\sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$.

A feedforward neural network is a sequence of layers where each layer receives the

⁴ This is functionally equivalent to simply using a different nonlinearity. Nonetheless, it is useful for learning procedures which manipulate parameters of a network but not the nonlinearities themselves.

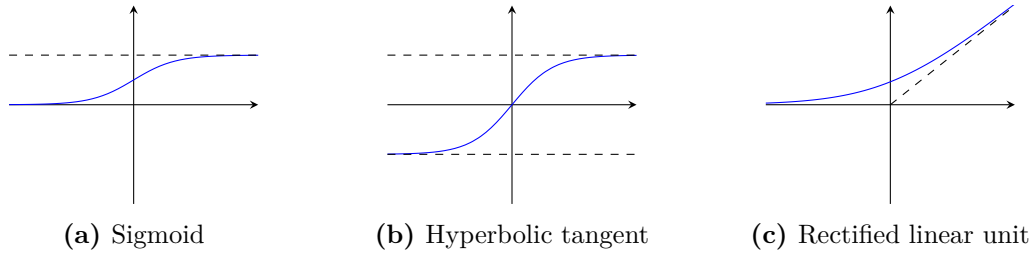


Figure 3-2: Common differentiable nonlinearities. (a) The sigmoid logistic function acts like an “on/off” switch for the neuron. (b) The hyperbolic tangent function is like a “good/bad” switch because it sends negative signal where the sigmoid gives no signal. (c) The rectified linear unit has an “off” state like the sigmoid, but provides almost-linear activation when there is sufficiently large input.

preceding layer’s output as its input. Using parenthesized superscripts to denote the layer number in sequence, a feedforward neural network’s behavior is defined by the following equations without input \mathbf{x} and output \mathbf{y} and L layers:

$$\begin{aligned} \mathbf{h}^{(0)} &= \mathbf{x} \\ \mathbf{h}^{(\ell)} &= \sigma^{(\ell)} \left(\mathbf{W}^{(\ell)} \mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)} \right) \\ \mathbf{y} &= \mathbf{h}^{(L)} \end{aligned}$$

The network’s entire operation is written $\mathbf{y} = f_{\theta}(\mathbf{x})$ where θ refers to the parameters $\theta = \{(\mathbf{W}^{(\ell)}, \mathbf{b}^{(\ell)})\}_{\ell=1}^L$.

Recurrent neural networks, such as the one shown in [Figure 3-3](#) have discrete time steps. These recurrent neural networks may often be understood by unfolding them: copying the network across time steps and make appropriate connections between time steps. If we let \mathbf{h}_t be the output of a recurrent unit g_{θ} at time step t and require some initial value \mathbf{h}_0 , then we can express the recurrent network’s computation as:

$$\mathbf{y}_t = g_{\theta}(\mathbf{x}_t, \mathbf{h}_{t-1})$$

Traditional recurrent neural networks often lead to a *vanishing gradient* during training via gradient descent, leading to rapid convergence on local minima which may be far from optimal. There are many techniques to mitigate this, most notably are the long short-term memory (LSTM) units which provide a mechanism for “forgetting” older values in a

recurrent cell [Hochreiter and Schmidhuber, 1997].

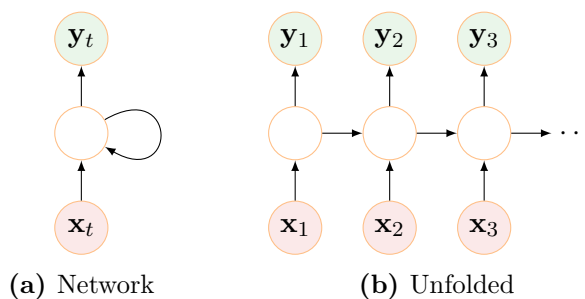


Figure 3-3: A recurrent neural network.

Deep recurrent neural networks provide a means for learning a latent representation of programs [Neelakantan et al., 2015; Reed and De Freitas, 2015; Devlin et al., 2017b; Chen et al., 2018]. By programs, we refer to more than functions from real-valued vectors to real-valued vectors: these programs may operate on symbolic and otherwise-structured data such as strings, lists, and trees.

3.3 Context-free grammars

Context-free grammars, or CFGs, are a method of describing languages based on the *production* of strings, or “sentences”, in a language [Chomsky, 1956; Sipser, 2006]. The languages that are producible by CFGs — *context-free languages* — are famously contrasted with *regular languages* which are describable with finite-state automata⁵, such as the one shown in Figure 3-4. Context-free grammars have a recursive property which permits expressive descriptions of certain languages without having to resort to an upper-bound on sentence length⁶. Context-free grammars were invented by Noam Chomsky during his study of human languages: their use aided in a formal understanding of the relationships between nouns, verbs, prepositions, and other linguistic phenomena. A context-free grammar for a

⁵ A finite-state automaton (FSA) is a machine described by a set of states, transitions between states upon input in some alphabet, an initial state, and a set of “accepting” states. A FSA associated with a regular language will yield an accepting state for some input if and only if the input is a sentence in the language.

In contrast, every CFG can be parsed (in linear time, as with regular grammars by FSAs) by a machine called a pushdown automaton (PDA). PDAs are like FSAs, but there is a persistent stack which transitions may manipulate (by push/pop) — that is, transitions depend on the current state, current input, *and* the top of the stack, yielding a new state and a replacement for the item that was read from the stack. PDAs may use the empty stack item ε (to move without reading from the stack and push a new item onto the stack, or to remove an item from the stack) as well the empty input ε which may be assumed at any time.

⁶ Every context-free language, if restricted to a particular finite limit on length, is also a regular language.

fragment of the English language is shown in [Figure 3-5](#).

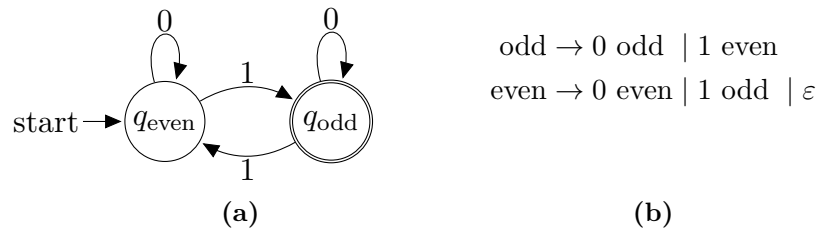
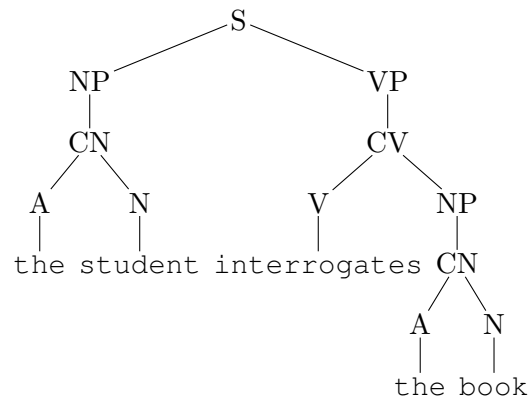


Figure 3-4: A language over all strings with an odd number of 1s. (a) A finite-state automaton which accepts strings in this language; (b) the regular grammar which produces sentences in this languages.

$S \rightarrow NP VP$
 $NP \rightarrow CN \mid CN PP$
 $VP \rightarrow CV \mid CV PP$
 $PP \rightarrow P CN$
 $CN \rightarrow A N$
 $CV \rightarrow V \mid V NP$
 $A \rightarrow a \mid the$
 $N \rightarrow student \mid robot \mid book$
 $V \rightarrow interrogates \mid likes \mid sees$
 $P \rightarrow near$



(a) (b)

Figure 3-5: (a) A fragment of the English language as a CFG, and (b) the parse tree for a production in that language. The acronyms of each nonterminal in order of rule definition are: Sentence, Noun Phrase, Verb Phrase, Prepositional Phrase, Complex Noun, Complex Verb, Article, Noun, Verb, Preposition.

CFGs are used in the study of intuitive theories — knowledge that organizes the world into causal structure [[Goodman et al., 2006](#); [Tenenbaum et al., 2007](#); [Ullman, 2015](#)]. The application of CFGs for these studies often relies on an *interpretive layer* which determines the production rules of a CFG and in which the semantics of a sentence are determined. For this reason, we believe that while CFGs provide a good representation for the study of how concepts may be constructed from some presupposed structure, they are lacking in expressive power for computation. By this we mean primarily that the *interpretive layer* should also be represented, and CFGs do not provide that.

Formally, a CFG (Σ, V, R, S) is defined by a set of terminals, or “alphabet”, Σ , a set

of nonterminals V disjoint from Σ , a set of production rules R , and a starting nonterminal $S \in V$. A production rule for a CFG has a left-hand side nonterminal and right-hand side sequence of terminals and nonterminals. We adopt the notation that a set of production rules for the same nonterminal can be written as one rule with distinct right-hand sides delimited by the alternation symbol $|$. CFGs are usually defined solely by a sequence of production rules, where nonterminals are inferred from the left-hand side of every rule, the terminals are inferred from all symbols on the right-hand side except the nonterminals, and the starting nonterminal is the left-hand side of the first defined rule. Production in a CFG is the process of iteratively applying applicable production rules to a string, initially set to S , until only terminals are left. An example of a production is shown in [Figure 3-5b](#).

3.4 Combinatory logic

Combinatory logic is based upon a number of primitive combinators that permit variable-free flow of information [[Schönfinkel, 1924](#); [Curry, 1958](#)]. Combinatory logic is Turing-complete: every computable function can be expressed in it [[Church, 1936](#); [Turing, 1937](#)]. In combinatory logic, every expression — or “combinator” — is either a primitive combinator or an application of two combinators. Combinators are higher-order functions which take one combinator and give another combinator. *Reduction* is the process by which primitive combinators are replaced using reduction rules which, in effect, define each primitive combinator. A combinator for which no reduction rules are applicable is said to be in *normal form*. A key distinction between context-free grammars and combinatory logic is that while every sentence in a context-free language can be parsed, not every combinator can be “computed” — i.e. reduced to normal form⁷. This is a consequence of combinatory logic being Turing-complete. [Section 3.1 Overview](#) explains why Turing-completeness is useful for representations of concepts. A key feature of combinatory logic is that, because it is variable-free, components of a combinator are themselves valid combinators. A notable use-case of combinatory logic is shown in [Figure 3-6](#).

Combinatory logic provides variable *routing* as an alternative to variable *binding*: inputs are routed to their destination in the combinatory structure. Because combinators can influence how subsequent arguments are used, they are typically thought of as “curried”

⁷ For example, $(S\ I\ I\ (S\ I\ I))$ in the BCIKS system of [Figure 3-7](#).

<code>if</code> \triangleq I	<code>(if true A B) = A</code>
<code>true</code> \triangleq K	<code>(if false A B) = B</code>
<code>false</code> \triangleq C K	<code>(and (not false)</code>
<code>and</code> \triangleq S C I	<code>(not (not true))) = true</code>
<code>or</code> \triangleq S I I	
<code>not</code> \triangleq C	

Figure 3-6: An example of computing with combinatory logic: Boolean logic. The symbol definitions are *not* a feature of combinatory logic: their purpose here is to illustrate a way combinators may be used to provide meaningful computation. A and B are illustrative placeholders for combinators. This is built atop the BCIKS system of [Figure 3-7](#).

functions: functions which take many arguments but are expressed such that subsequent arguments are applied in a nested fashion, with intermediate applications yielding partially-applied functions.

Syntactically, single characters are used to denote primitive combinators, and parentheses are used to denote application: f applied to x is $(f x)$. Combinators are defined inductively:

1. Every primitive combinator is a combinator.
2. If A and B are combinators, then $(A B)$ is a combinator.

Parentheses may be omitted in a left-associative manner: $((f x) y)$ is equivalently written as $f x y$. A system of combinatory logic, described by the transformations made by each primitive combinator when given a sufficient number of arguments, is depicted in [Figure 3-7](#).

B $f g x \rightarrow f (g x)$	(composition)
C $f x y \rightarrow f y x$	(swap)
I $x \rightarrow x$	(identity)
K $x y \rightarrow x$	(drop)
S $f g x \rightarrow f x (g x)$	(duplicate)

Figure 3-7: The Schönfinkel BCIKS system of combinatory logic. These rules describe how to perform reduction of a combinator.

In practice, combinatory logic is often interpreted in an environment which ascribes non-combinatory meaning to certain primitive combinators. For example $(\mathbf{S} * \mathbf{I} 2)$ reduces to $(* 2 2)$, which an interpreter may further evaluate to 4. This is theoretically unnecessary, as there exists an encoding for numbers in combinatory logic with relevant arithmetic operations such that reduction performs this evaluation.

Combinators may further be represented as binary trees where leaf is a primitive combinator and non-leaf nodes refer to function application of the node's left child with its right child. See Figure 3-8 for an example of a combinator as a binary tree. An important feature of this binary tree representation of combinators is that any subtree of a valid combinator is itself a valid combinator.

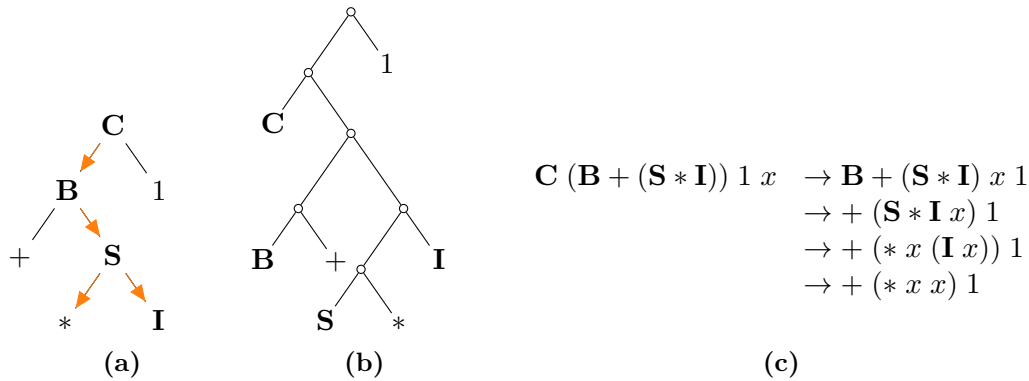


Figure 3-8: Binary tree representations for $f(x) = x^2 + 1$, corresponding to the combinatory logic expression $(\mathbf{C} (\mathbf{B} + (\mathbf{S} * \mathbf{I})) 1)$: (a) with primitive combinators as node labels and arrows to visualize the path of an input; (b) primitive combinators as terminals; (c) reduction of the combinator when supplied argument x .

3.4.1 Polymorphic typed combinatory logic

Combinatory logic may be extended with types to prevent nonsense combinators such as $(3 * \mathbf{K})$. *Types* are sets of values that may inhabit a term. For example, the type of 3 is `Int`, or “integer”⁸, and only integers can inhabit the placeholders in the combinator $(* _ _)$. We define the arrow type, which describes functions, as $(\alpha \rightarrow \beta)$ for functions that take values of type α and return values of type β . The type of $(* 3)$, then, is $(\text{Int} \rightarrow \text{Int})$. Parentheses

⁸ The type of 3 and $(*)$ can be different depending on the framing: it may be a positive number, or a natural number, or a rational number, or a real number, etc. Integer is chosen here by common computer programming convention. This ambiguity is beyond the scope of this work, but an understanding of *subtyping* should suffice as clarification.

for arrow types may be omitted in a right-associative manner: $(\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}))$, the type for a function that takes two Ints and returns an Int (e.g. multiplication $(*)$), is equivalently written as $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$. *Simple types* [Church, 1940] are defined inductively from a set of base types such as `Int` (Integer) or `Bool` (Boolean) as follows:

1. Every base type is a simple type.
2. If α and β are simple types, then $\alpha \rightarrow \beta$ is a simple type.

When adopting a type system, every valid term (or in this case, combinator) must have a type, so we will often adopt the notation $x : \tau$ for term x with type τ denoted by a Greek letter. Equipped with the type of every primitive combinator, we inductively define the types for all (valid) combinators:

1. The type of a primitive combinator is given.
2. If $(A : \alpha \rightarrow \beta)$ and $(B : \alpha)$ are combinators, then the application $(A B)$ has type β .

From this definition the utility of types is apparent: combinators can only be combined if their types match appropriately.

Types for combinatory logic get more complicated when we must deal with the primitive combinators **B**, **C**, **I**, **K**, and **S**. A first effort may be to copy these combinators for every simple type, so \mathbf{I}_α is operationally equivalent to **I** but it has the type α and is defined for every α . This would yield infinitely many copies of each combinator. We instead adopt *parametric polymorphism*: a combinator may have a simple type as discussed above, or it may have a *type schema*⁹ which permits universally quantified *type variables*. We further extend simple types to *monotypes*, which are like simple types but may include type variables that are *not* quantified¹⁰. For example, the type schema of **K** is $\forall\alpha\forall\beta. \alpha \rightarrow \beta \rightarrow \alpha$. Once **K** is used, types are filled in appropriately depending on how it is used¹¹: the type (monotype)

⁹ Combinators may be regarded as having types or as having type schemas depending on what is relevant. There is an important distinction between the *type schema* $\forall\alpha. \alpha \rightarrow \alpha$ and the *type* $\alpha \rightarrow \alpha$: the former is universally quantified to work with any type (i.e. the identity function) and the latter is an instance which, as effectively an existential quantification over type variables, works for *some* type α (e.g. unary negation over Booleans $\text{Bool} \rightarrow \text{Bool}$). In practice, primitive combinators have associated type schemas and every constructed combinator has a type.

¹⁰ It is important to recognize that valid combinators may further be constrained with the inclusion of type variables for monotypes. For example, in the monotype $\alpha \rightarrow \alpha$ the α s are identical. These type variables are kept consistent by a structure called the type context.

¹¹ This is typically done by a process called *type inference*, which we do not discuss here. See Pierce [2002] for a more thorough introduction to types and type inference for programming languages.

of $(\mathbf{K} 1)$ is $\beta \rightarrow \text{Int}$ and the \mathbf{K} used in that term has type $\text{Int} \rightarrow \beta \rightarrow \text{Int}$. As expected, the type of $(K 3 S)$ is Int . Type schemas for each primitive combinator discussed in this section are shown in [Figure 3-9](#). The naming of type variables must be unique for new polymorphic combinators that get introduced. This necessitates a *type context* to keep track of which type variable names have been introduced, to maintain assignments to type variables that have been inferred, and to provide a process of *instantiation* by which new primitive combinator may be used with new type variables where applicable.

$$\begin{aligned}
\mathbf{B} &: \forall\alpha\forall\beta\forall\gamma. (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma \\
\mathbf{C} &: \forall\alpha\forall\beta. (\alpha \rightarrow \alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \alpha \rightarrow \beta \\
\mathbf{I} &: \forall\alpha. \alpha \rightarrow \alpha \\
\mathbf{K} &: \forall\alpha\forall\beta. \alpha \rightarrow \beta \rightarrow \alpha \\
\mathbf{S} &: \forall\alpha\forall\beta\forall\gamma. (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma
\end{aligned}$$

Figure 3-9: Type schemas for the Schönfinkel BCIKS combinators.

This type system, called the Hindley-Milner type system [[Hindley, 1969](#); [Milner, 1978](#); [Damas and Milner, 1982](#)], is implemented in the many programming language, such as ML and Haskell. We provide our own implementation at <https://github.com/lucasem/polytype-rs>¹².

3.5 Lambda-calculus

Lambda-calculus (henceforth written λ -calculus) is a formalism of computation which is based on variable *binding* instead of variable *routing*, unlike combinatory logic (see [Section 3.4 Combinatory logic](#)) [[Church, 1941](#)]. Fundamental to λ -calculus is the procedural abstraction, by which procedures may generalize to take an input. For example, $2 + 3$ could be generalized to $f(x) = 2 + x$, both of which are clearly represented in λ -calculus¹³. λ -calculus has become foundational in both computer science and programming curricula for its intuitive and expressive power. A family of programming languages, LISP, is based upon it, which permits intuitive computational patterns such as the one shown in [Figure 3-10](#) [[Abelson et al., 1996](#)]. For terminology, where combinatory logic had combinators,

¹² Documentation is located at <https://docs.rs/polytype>.

¹³ Represented as $((+ 2) 3)$ and $(\lambda x((+ 2) x))$, respectively.

λ -calculus has (valid) *expressions*.

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

Figure 3-10: A procedure in Scheme, a dialect of LISP, based on the abstractions provided by λ -calculus. Here we've defined a function for the greatest common denominator (**gcd**) using (**define (gcd a b) (...)**). This corresponds to the expression in λ -calculus of $(\lambda G \langle \text{rest} \rangle)(\lambda a(\lambda b(\dots)))$ where $\langle \text{rest} \rangle$ is an expression that may use G as a procedure for **gcd**.

λ -calculus has more sophisticated syntax than combinatory logic. There are three improper symbols λ , (, and), as well as an infinite number of variables $a, b, \dots, \bar{a}, \bar{b}, \dots, \bar{\bar{a}}, \dots$. Parentheses are used to denote function application as with combinatory logic, and are additionally used directly around a λ -abstraction, which is an expression of the form $(\lambda \mathbf{x} \mathbf{M})$. Independent of the syntax of λ -calculus itself, we use bold capital letters such as \mathbf{M} , to refer to a particular expression, and bold lowercase letters such as \mathbf{x} to refer to a particular variable. Each expression associates a mutually-exclusive attribute *free* or *bound* to any variables contained within it. Expressions of λ -calculus are determined according to the following rules:

1. A variable \mathbf{x} is a valid expression that has \mathbf{x} as a free variable.
2. If \mathbf{F} and \mathbf{A} are valid expressions, then the application $(\mathbf{F} \mathbf{A})$ is a valid expression. Variables that are free in either \mathbf{F} or \mathbf{A} are also free in $(\mathbf{F} \mathbf{A})$.
3. If \mathbf{M} is a valid expression and has free variable \mathbf{x} , then $(\lambda \mathbf{x} \mathbf{M})$ is a valid expression where \mathbf{x} is bound.

We use $\mathbf{M}[\mathbf{x} := \mathbf{N}]$ to denote the resulting expression of substituting \mathbf{x} with \mathbf{N} throughout \mathbf{M} . The expression created by substitution $\mathbf{M}[\mathbf{x} := \mathbf{N}]$ is valid if \mathbf{M} and \mathbf{N} are both valid, \mathbf{x} is not bound in \mathbf{M} , and the free variables of \mathbf{N} are distinct from the bound variables of \mathbf{M} . *Conversion* is the process by which valid expressions may be transformed into other valid expressions:

1. Replace any part \mathbf{M} of an expression with $\mathbf{M}[\mathbf{x} := \mathbf{y}]$ where \mathbf{x} is not free in \mathbf{M} and \mathbf{y} does not occur in \mathbf{M} . This operation is called α -conversion, and its intension is to avoid variable naming collisions that prevent other operations from being performed.

2. Replace any part $((\lambda \mathbf{xM}) \mathbf{N})$ of an expression with $\mathbf{M}[\mathbf{x} := \mathbf{N}]$ where the free variables of \mathbf{N} are distinct from the bound variables of \mathbf{M} . This operation is called β -reduction.
3. Replace any part $\mathbf{M}[\mathbf{x} := \mathbf{N}]$ of an expression with $((\lambda \mathbf{xM}) \mathbf{N})$ where $((\lambda \mathbf{xM}) \mathbf{N})$ is a valid expression, \mathbf{x} is not bound in \mathbf{M} , and the free variables of \mathbf{N} are distinct from the bound variables of \mathbf{M} . This operation is sometimes called inverse-inlining.

A valid expression is in *normal form* if it contains no part of the form $((\lambda \mathbf{xM}) \mathbf{N})$. If such a normal form exists, then it is unique up to a number of α -conversions. *Reduction* of an expression refers to the process of applying α -conversions and β -reductions to yield an expression in normal form. Not every expression can be reduced¹⁴.

In practice, λ -calculus is often modified to permit non-variable constants which are interpreted during reduction. For example $((\lambda x(* x 2)) 2)$ reduces to $(* 2 2)$, which an interpreter may further evaluate to 4. As with combinatory logic, this is theoretically unnecessary because there exists an encoding for numbers in λ -calculus with relevant arithmetic operations such that β -reduction performs this evaluation, as shown in [Figure 3-11](#). Defining data and operators in λ -calculus is called *Church encoding*.

$$\begin{aligned}
 1 &\rightarrow (\lambda a(\lambda b(a b))) \\
 2 &\rightarrow (\lambda a(\lambda b(a (a b)))) \\
 3 &\rightarrow (\lambda a(\lambda b(a (a (a b))))) \\
 &\vdots \\
 \mathbf{M} + \mathbf{N} &\rightarrow (\lambda \mathbf{a}(\lambda \mathbf{b}((\mathbf{M} \mathbf{a}) ((\mathbf{N} \mathbf{a}) \mathbf{b})))) \\
 \mathbf{M} \times \mathbf{N} &\rightarrow (\lambda \mathbf{a}(\mathbf{M} (\mathbf{N} \mathbf{a}))) \\
 \mathbf{M}^{\mathbf{N}} &\rightarrow (\mathbf{N} \mathbf{M})
 \end{aligned}$$

Figure 3-11: Arithmetic in pure λ -calculus. Arrows represent encoding. Bold lowercase letters starting from **a** denote unused variable names.

Dealing with conflicting variable names by α -conversion is hardly convenient, so practical implementations of λ -calculus abandon the notion of α -conversion in favor of the *de Bruijn index* notation [[De Bruijn, 1972](#)]. In this notation, variables are described not by names, but by numbers indicating the location of the abstraction (λ) where a variable is bound: 1

¹⁴ For example, $((\lambda x(x x)) (\lambda x(x x)))$.

refers to the nearest surrounding abstraction, 2 refers to the nearest surrounding abstraction around the abstraction where 1 is bound, etc. For example, $(\lambda a(\lambda b(a b)))$ is written in de Bruijn index notation as $(\lambda(\lambda(2\ 1)))$, or simply $\lambda\ \lambda\ 2\ 1$ with left-associative parenthesis omission. This binding pattern is illustrated by [Figure 3-12](#). We will refrain from using this notation in this exposition of λ -calculus, but we adopt it in our practical implementations.

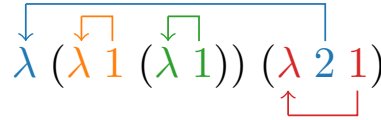


Figure 3-12: de Bruijn index notation. Colors and arrows indicate distinct variables and the abstractions which introduce them.

3.5.1 Simply typed lambda-calculus

Much like combinatory logic, λ -calculus may be extended with types to prevent nonsense operations such as $((3\ *) (\lambda a(\lambda b a)))$. However, we don't need to adopt a sophisticated type system like Hindley-Milner for λ -calculus as we did for combinatory logic: simple types suffice [[Church, 1940](#); [Curry, 1958](#)] as there were described in [Section 3.4.1 Polymorphic typed combinatory logic](#). Equipped with the type of every constant, we inductively define (valid) typed expressions:

1. Every constant is an expression with its given type.
2. A variable $(\mathbf{x} : \alpha)$ is an expression with type α that has \mathbf{x} as a free variable¹⁵.
3. If $(\mathbf{F} : \alpha \rightarrow \beta)$ and $(\mathbf{A} : \alpha)$ are expressions, then the application $(\mathbf{F}\ \mathbf{A})$ has type β . Variables that are free in either \mathbf{F} or \mathbf{A} are also free in $(\mathbf{F}\ \mathbf{A})$.
4. If $(\mathbf{M} : \beta)$ is an expression and has free variable $(\mathbf{x} : \alpha)$, then $(\lambda \mathbf{x} \mathbf{M})$ is a valid expression with type $\alpha \rightarrow \beta$ where \mathbf{x} is bound.

3.5.2 Polymorphic typed lambda-calculus

A polymorphic type system over λ -calculus has been studied as System F [[Girard, 1972](#); [Reynolds, 1974](#)]. In practice, the Hindler-Milner (HM) polymorphic type system is used. We did not cover this system in full detail in [Section 3.4.1 Polymorphic typed combinatory](#)

¹⁵ Variable must consistently be used with the same type.

logic, but we take this opportunity to fill in some relevant details. HM allows programming languages to be equipped with particular type constructors so that “lists with elements of type T ” can be represented with the type `List T`. With this faculty baked in, we can write list operations within the type system as shown in Figure 3-13. These functions implicitly take in types as arguments, which serve to replace the universal quantification with a substitution for the bound type variable. Type inference is a technique which performs this implicit type substitution [Pierce, 2002].

$$\begin{aligned}
 \text{nil} &: \forall X. \text{List } X \\
 \text{cons} &: \forall X. X \rightarrow \text{List } X \rightarrow \text{List } X \\
 \text{isnil} &: \forall X. \text{List } X \rightarrow \text{Bool} \\
 \text{head} &: \forall X. \text{List } X \rightarrow X \\
 \text{tail} &: \forall X. \text{List } X \rightarrow \text{List } X \\
 \text{filter} &: \forall X. (X \rightarrow \text{Bool}) \rightarrow \text{List } X \rightarrow \text{List } X \\
 \text{map} &: \forall X. \forall Y. (X \rightarrow Y) \rightarrow \text{List } X \rightarrow \text{List } Y \\
 \text{fold} &: \forall X. \forall Y. (Y \rightarrow X \rightarrow Y) \rightarrow Y \rightarrow \text{List } X \rightarrow Y
 \end{aligned}$$

(a)

$$\begin{aligned}
 \text{map} &= (\lambda f(\lambda \ell(((\text{fold } (\lambda y(\lambda x((\text{cons } (f x)) y)))) \text{nil}) \ell))) \\
 &= \lambda f \ell. \text{fold } (\lambda y x. \text{cons } (f x) y) \text{nil } \ell
 \end{aligned}$$

(b)

```
(define map (λ (f l) (fold (λ (y x) (cons (f x) y)) nil l)))
```

(c)

Figure 3-13: List operations in polymorphic λ -calculus. (a) Types for various list operations; (b) definition of the `map` primitive in λ -calculus, in standard syntax and with omitted parentheses, given `fold`, `cons`, and `nil`; (c) definition of `map` in the Lisp programming language.

We provide our own implementation of polymorphic λ -calculus at <https://github.com/lucasem/program-induction>¹⁶.

¹⁶ Documentation for the λ -calculus representation is located at <https://docs.rs/programinduction/~0/programinduction/lambda/index.html>.

3.6 Term rewriting

Term rewriting is yet another formalism for computation [Bezem et al., 2003]. Like λ -calculus, it relies on variable binding rather than routing. Fundamental to term rewriting is a form of pattern matching. Programming languages like Haskell may often be viewed in the frame of term rewriting, as in Figure 3-14. In this section, unless otherwise noted, we are working with *first-order* term rewriting systems¹⁷.

<pre>not True = False not False = True even 0 = True even n = odd (n-1) odd n = not (even n)</pre>	<pre>odd 3 → not (even 3) → not (odd 2) → not (not (even 2)) → not (not (odd 1)) → not (not (not (even 1))) → not (not (not (odd 0))) → not (not (not (not (even 0)))) → not (not (not (not True))) → not (not (not False)) → not (not True) → not False → True</pre>
(a)	(b)

Figure 3-14: (a) Functions in Haskell for determining whether a natural number is even or odd. These could be viewed as procedure calls, however they may instead be viewed as rewrites to a term. It is trivial to convert this Haskell code into a term rewriting system. (The fourth rule, which decrements a natural number in the right-hand side, may instead take the successor of a natural number on the left-hand side so the rewrite system does not have to implement subtraction). (b) The rewriting of a term in this system.

A term rewriting system (TRS) is a pair (Σ, R) where Σ is the *signature* that specifies objects in the system — *terms* — and R determines a set of binary relations — *rewrite rules* — over those terms. We begin with an example in Figure 3-15a and describe the TRS formalism using the example as reference. The signature Σ defines *symbols* which each have a nonnegative *arity* denoted by superscript: $A^{(2)}$ has arity two (binary), $S^{(1)}$ has arity one (unary), and $Z^{(0)}$ has arity zero (nullary). Arity is in effect the number of arguments a function, defined by a symbol, takes: a symbol with arity n corresponds to an n -ary function. We use lowercase letters x, y, \dots to denote *variables*: there is an implicit countably-infinite set Var disjoint from Σ for which the notion of arity does not apply. With Var and Σ , we can inductively define the set of (valid) terms over Σ :

¹⁷ Higher-order rewrite systems are discussed in Section 3.6.2 Higher-order rewriting

$$\begin{array}{ll}
\Sigma := A^{(2)}, M^{(2)}, S^{(1)}, Z^{(0)} & \Sigma := \circ^{(2)}, B^{(0)}, C^{(0)}, I^{(0)}, K^{(0)}, S^{(0)} \\
R := \{\rho_1, \rho_2, \rho_3, \rho_4\} & R := \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\} \\
\rho_1 : A(x, Z) \rightarrow x & \rho_1 : \circ(\circ(\circ(B, f), g), x) \rightarrow \circ(f, \circ(g, x)) \\
\rho_2 : A(x, S(y)) \rightarrow S(A(x, y)) & \rho_2 : \circ(\circ(\circ(C, f), x), y) \rightarrow \circ(\circ(f, y), x) \\
\rho_3 : M(x, Z) \rightarrow Z & \rho_3 : \circ(I, x) \rightarrow x \\
\rho_4 : M(x, S(y)) \rightarrow A(M(x, y), x) & \rho_4 : \circ(\circ(K, x), y) \rightarrow x \\
& \rho_5 : \circ(\circ(\circ(S, f), g), x) \rightarrow \circ(\circ(f, x), \circ(g, x))
\end{array}$$

(a) (b)

Figure 3-15: Term rewriting system for (a) arithmetic on natural numbers and (b) combinatory logic where \circ is the symbol for application of two combinators.

1. Every $x \in Var$ is a term.
2. If $F^{(n)} \in \Sigma$ and t_1, \dots, t_n are terms, then $F(t_1, \dots, t_n)$ is a term. Nullary symbols are written as F instead of $F()$.

We use \equiv to express syntactical identity between two terms. It is useful to distinguish terms that have variables from those that have none. Let $Var(t)$ be the variables that occur in term t . A term t is a *ground term* if $Var(t) = \emptyset$, it is *linear* if $Var(t) > 0$ and each variable in t occurs exactly once, and it is *non-linear* otherwise. The term in Figure 3-16a is a ground term, while the term in Figure 3-16b is a linear term.

Terms may be represented as labelled trees, or *term trees*. Each node in a term tree corresponds to either a symbol $F_n \in \Sigma$, where the node is labelled by F and it must have exactly n child nodes, or a variable $x \in Var$, where the node is labelled by x and it must have no children. See Figure 3-16 for examples of term trees.

Similar to terms, there are *contexts*. A context is an incomplete term which may contain empty places, or *holes*. Contexts may be thought of as terms in an extended signature $\Sigma \cup \{\square^{(0)}\}$. If C is a context with n holes and t_1, \dots, t_n are terms, then $C[t_1, \dots, t_n]$ is the replacement of holes of C from left to right by the corresponding terms t_1, \dots, t_n . Every context which has a replacement that yields the term t is called a *prefix* of t . A one-hole context C , specially denoted $C[]$, is an apparently useful construct. If a term t can be written as $C[s]$, then the term s is a *subterm* of t . With contexts, a new construct is motivated: *positions*. A position p in a term t uniquely determines a one-holed prefix $t[]_p$

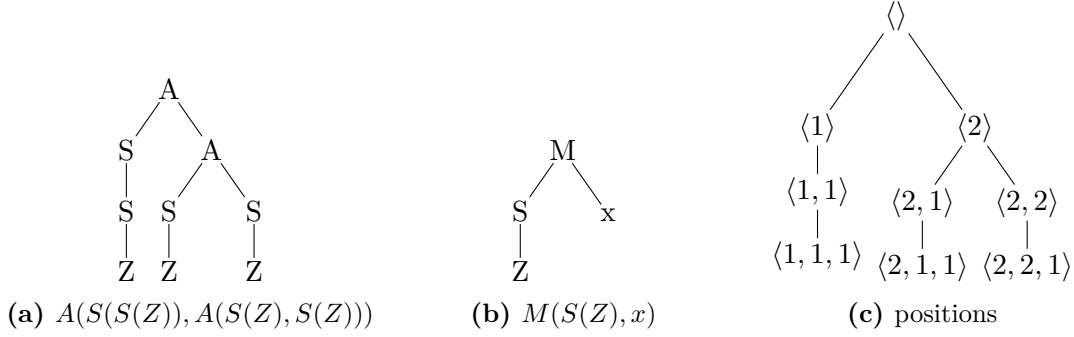


Figure 3-16: (a,b) Term trees under the arithmetic TRS of Figure 3-15a.
(c) A tree diagram showing positions associated with a term tree.

as well as a subterm s that occurs at position p , written $t|_p$. Positions are represented by finite arrays which encode movements in a term tree starting from the root and moving left-to-right starting at 1 (see Figure 3-16c for a visual depiction).

Substitution is the replacement of variables with terms. It is a map σ from terms to terms satisfying

$$\sigma(F(t_1, \dots, t_n)) \equiv F(\sigma(t_1), \dots, \sigma(t_n))$$

for all symbols $F^{(n)} \in \Sigma$. This is operationally equivalent to a recursively applied mapping of variables to terms, so we adopt the notation $\{x_1 \mapsto s_1, \dots, x_m \mapsto s_m\}$, where $x_1, \dots, x_m \in \text{Var}$ and s_1, \dots, s_m are terms, for defining a substitution. If $\sigma(s) \equiv \sigma(t)$, then the substitution σ is a *unifier* for t and s .

Until now, we have discussed *terms* in term rewriting systems. We now discuss *rewriting*. A *rewrite rule* ρ for a signature Σ is a pair of terms $\rho : l \rightarrow r$ which satisfies the restrictions $l \notin \text{Var}$ and $\text{Var}(r) \subseteq \text{Var}(l)$. A *rewrite step* according to rewrite rule $\rho : l \rightarrow r$ is a relation produced by applying a substitution σ to both terms of the rule within an arbitrary context:

$$C[\sigma(l)] \rightarrow_\rho C[\sigma(r)]$$

For a set of rewrite rules R we define the *one-step reduction* \rightarrow_R as the relation comprised of all rules $\bigcup\{\rightarrow_\rho \mid \rho \in R\}$. A *reduction* is a sequence of one-step reductions $t_1 \rightarrow_R \dots \rightarrow_R t_n$ for which we write $t_1 \rightarrow t_n$ and call t_n a *reduct* of t_1 . A reduction, described by each rewrite step, is shown in Figure 3-17.

$$\begin{aligned}
A(S(Z), \underline{M(S(S(Z)), S(Z))}) &\rightarrow_{\rho_4} A(S(Z), \underline{A(M(S(S(Z)), Z), S(S(Z)))}) \\
&\rightarrow_{\rho_2} A(S(Z), \underline{S(A(M(S(S(Z)), Z), S(Z)))}) \\
&\rightarrow_{\rho_2} A(S(Z), \underline{S(S(A(M(S(S(Z)), Z), Z))})} \\
&\rightarrow_{\rho_1} A(S(Z), \underline{S(S(M(S(S(Z)), Z))})} \\
&\rightarrow_{\rho_3} \underline{A(S(Z), S(S(Z)))} \\
&\rightarrow_{\rho_2} \underline{S(A(S(Z), S(Z)))} \\
&\rightarrow_{\rho_2} \underline{S(S(A(S(Z), Z))} \\
&\rightarrow_{\rho_1} S(S(S(Z)))
\end{aligned}$$

Figure 3-17: A reduction of $1 + 2 \times 1$ in the arithmetic TRS of Figure 3-15a. This is not the only reduction from $A(S(Z), M(S(S(Z)), S(Z)))$ to $S(S(S(Z)))$.

3.6.1 Polymorphic typed term rewriting

An extension to term rewriting systems is the association of a type for every term. The two TRSs of Figure 3-15 suffice without types, but a sufficiently complex TRS such as the one shown in Figure 3-18 requires types to prevent nonsense like $S(Cons(S(Z), Cons(Z, Nil)))$ ¹⁸. Without types, we are forced to either accept such a term and be unable to reduce it, or to introduce rules which can reduce it, such as $\rho' : S(Cons(x, y)) \rightarrow Cons(S(x), S(y))$. As with previously discussed representations, we adopt the Hindley-Milner polymorphic type system. Arity is no longer needed to describe a symbol because a type such as $Int \rightarrow Int \rightarrow Int$ inherently determines the number of arguments a symbol requires. Specifically, each symbol is associated a *type schema*, as described in Section 3.4.1 Polymorphic typed combinatory logic, and every use of a symbol in a term is instantiated under a type context to remove quantifiers¹⁹ thus yielding a type. We provide an implementation of polymorphic first-order term rewriting at <https://github.com/lucasem/program-induction>²⁰.

¹⁸ Read “the successor of the list $[1,0]$ ”. What does it mean to get the successor of a list?

¹⁹ Because every term has a *type*, not a type schema. We regret the overloaded usage of the word “type”.

²⁰ Documentation for the first-order term rewriting representation is located at <https://docs.rs/programinduction/~0/programinduction/trs/index.html>.

$\Sigma := Head^{(1)}, Tail^{(1)}, Cons^{(2)}, Nil^{(0)}, S^{(1)}, Z^{(0)}$ $R := \{\rho_1, \rho_2\}$ $\rho_1 : Head(Cons(x, y)) \rightarrow x$ $\rho_2 : Tail(Cons(x, y)) \rightarrow y$	$Head : \forall X. List\ X \rightarrow X$ $Tail : \forall X. List\ X \rightarrow List\ X$ $Cons : \forall X. X \rightarrow List\ X \rightarrow List\ X$ $Nil : \forall X. List\ X$ $S : Nat \rightarrow Nat$ $Z : Nat$
(a)	(b)

Figure 3-18: (a) A term rewriting systems for lists and natural numbers and (b) the type of each symbol in its signature.

3.6.2 Higher-order rewriting

A drawback to first-order term rewriting, as discussed above, is that it does not allow symbol binding: a higher order function such as `map` cannot be expressed as a rewrite rule directly. A higher-order rewrite system gives us this expressive power — here we describe a polymorphic typed variant of the combinatory reduction system which we name PCRS²¹ [Klop, 1980; Klop et al., 1993]. We use an example, in Figure 3-19, to motivate and guide our exposition of PCRSs.

We take Σ to be a set of symbols which each have a type $f : \tau$. Let $\text{args}(\tau)$ be the list of arguments types for type τ ²² and let $\text{yield}(\tau)$ be the return type for type τ ²³. Let Var be a countably-infinite set of variables disjoint from Σ where each variable has the universal schema $(x : \forall \alpha. \alpha) \in Var$. Let $MetaVar$ be a countably-infinite set of *meta-variables* disjoint from $\Sigma \cup Var$ where each meta-variable has the universal schema $(F : \forall \alpha. \alpha) \in MetaVar$. Variables and meta-variables, when used in the same object, have the same instantiated type²⁴. *Metaterms* for a signature Σ are defined inductively as follows:

1. Every $(x : \tau) \in Var$ is a metaterm with type τ .

²¹ Polymorphic combinatory reduction system

²² Assuming sequence concatenation by $+$:

$$\text{args}(\tau) \triangleq \begin{cases} [\alpha] + \text{args}(\beta) & \text{if } \tau = \alpha \rightarrow \beta \\ [] & \text{otherwise} \end{cases}$$

²³

$$\text{yield}(\tau) \triangleq \begin{cases} \text{yield}(\beta) & \text{if } \tau = \alpha \rightarrow \beta \\ \tau & \text{otherwise} \end{cases}$$

²⁴ Recall that instantiation is the removal of quantifiers corresponding to a particular instance. Type inference uses constraints of a term's usage to determine its concrete type

$$\begin{aligned}
\Sigma &:= \{ (\mathbf{nil} : \forall X. \text{List } X), \\
&\quad (\mathbf{cons} : \forall X. X \rightarrow \text{List } X \rightarrow \text{List } X), \\
&\quad (\mathbf{map} : \forall X. \forall Y. (X \rightarrow Y) \rightarrow \text{List } X \rightarrow \text{List } Y), \\
&\quad (+ : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}), \\
&\quad (* : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}), \\
&\quad (\mathbf{s} : \text{Nat} \rightarrow \text{Nat}), \\
&\quad (\mathbf{z} : \text{Nat}) \} \\
R &:= \{ \rho_1, \rho_2, \rho_3, \rho_4, \rho_5, \rho_6 \} \\
\rho_1 &: +(X, \mathbf{z}) \quad \rightarrow X \\
\rho_2 &: +(X, \mathbf{s}(Y)) \quad \rightarrow \mathbf{s}(+(X, Y)) \\
\rho_3 &: *(X, \mathbf{z}) \quad \rightarrow \mathbf{z} \\
\rho_4 &: *(X, \mathbf{s}(Y)) \quad \rightarrow +(*(X, Y), X) \\
\rho_5 &: \mathbf{map}(\lambda x. F(x), \mathbf{nil}) \quad \rightarrow \mathbf{nil} \\
\rho_6 &: \mathbf{map}(\lambda x. F(x), \mathbf{cons}(H, T)) \rightarrow \mathbf{cons}(F(H), \mathbf{map}(\lambda x. F(x), T))
\end{aligned}$$

Figure 3-19: A polymorphic combinatory reduction system to demonstrate the map list operation. For legibility, symbols are written in blue, meta-variables in orange, and variables in red.

2. If $(f : \tau) \in \Sigma$ and t_1, \dots, t_n are metaterms with types corresponding to $\text{args}(\tau)$, then $f(t_1, \dots, t_n)$ is a metaterm with type $\text{yield}(\tau)$. Symbols that have non-arrow type are written as F instead of $F()$.
3. If $(x : \alpha) \in \text{Var}$ and $s : \tau$ is a metaterm, then $\lambda x. s$ is a metaterm with type $\alpha \rightarrow \tau$.
4. If $(F : \tau) \in \text{MetaVar}$ and t_1, \dots, t_n are metaterms with types corresponding to $\text{args}(\tau)$, then $F(t_1, \dots, t_n)$ is a metaterm with type $\text{yield}(\tau)$.

A variable $(x : \tau) \in \text{Var}$ is said to be *bound* in a metaterm if every occurrence of it is within a scope of the form $\lambda x. t$. We define a *term* as a metaterm with no meta-variables. For example, following the system of Figure 3-19, $\mathbf{map}(\lambda x. \mathbf{s}(x), \mathbf{cons}(z, \mathbf{nil}))$ is a term. Note that meta-variables here correspond to variables in first-order TRSs, and the distinction in TRSs between ground terms and non-ground terms is taken here as the distinction between terms and metaterms. We also redefine contexts for PCRSs as they were defined earlier in Section 3.6 Term rewriting but restricted to enforce types always match. For example, if s is the term consisting only of the free variable x , and $C[\]$ is a one-hole context $f(\lambda x. \square)$, then the replacement $C[s]$ results in the capturing of s 's free variable yielding the term $f(\lambda x. x)$.

A rewrite rule for a PCRS is a pair of metaterms l, r satisfying the following restrictions:

1. Every meta-variable in l has distinct bound variables as arguments. This was not applicable for first-order TRSs because a variable always had arity 0, and that restriction is relaxed here for meta-variables.
2. Every meta-variable in r also occurs in l .
3. Every variable in l and r is bound.

Recall that for first-order TRSs, variables (that always have arity 0) in rules would be substituted with ground terms (with root symbol of any arity) during reduction. For higher-order rewrite systems, and here in particular PCRSs, meta-variables are “meta-substituted” in a manner that allows any term to take its place (provided the types match and no free variables are introduced). We will formalize this shortly, but first consider the example metaterm $\lambda u.F(u)$ and term $\lambda x.+(x, x)$: if a meta-substitution σ permitted replacing F with $\lambda x.+(x, x)$, then the substitution applied to the metaterm would yield $\lambda u.(\lambda x.+(x, x))(u)$ which simplifies to $\lambda u.+(u, u)$ ²⁵. We now formalize meta-substitution and simplification to permit this kind of operation.

A *meta-substitution* is a mapping σ from metaterms to terms, effectively defined by replacements of meta-variables with terms and an accompanying simplification method, which satisfies the following:

$$\begin{aligned} \sigma(x) &\equiv x && \text{for } x \in \text{Var} \\ \sigma(F(t_1, \dots, t_n)) &\equiv \sigma(F)(\sigma(t_1), \dots, \sigma(t_n)) && \text{for } F \in \text{MetaVar} \\ \sigma(\lambda x.s) &\equiv \lambda x.\sigma(s) \\ \sigma(\mathfrak{f}(t_1, \dots, t_n)) &\equiv \mathfrak{f}(\sigma(t_1), \dots, \sigma(t_n)) && \text{for } \mathfrak{f} \in \Sigma \end{aligned}$$

where simplification, indicated in the marked substitution clause (*), is the replacement of terms of the form $(\lambda x.t)(s)$ with $t[x \mapsto s]$ — where bound occurrences of the variable x in the term t are replaced with s .

A rewrite step using a rewrite rule $\rho : l \rightarrow r$ relates two terms t and s as $t \rightarrow_\rho s$ if there exists a one-hole context C and meta-substitution σ such that $s \equiv C[\sigma(l)]$ and $t \equiv C[\sigma(r)]$.

²⁵ This example is motivated by the PCRS of Figure 3-19: it is not obvious that the term $\text{map}(\lambda x.+(x, x), \text{cons}(s(z), \text{nil}))$ can be rewritten using the rule ρ_6 . See Figure 3-20 for the reduction of a term like this.

A reduction is a sequence of rewrite steps using a set of rules R . A reduction for the PCRS of Figure 3-19 is shown in Figure 3-20.

$$\begin{aligned}
& \underline{\text{map}(\lambda x.+(x, x), \text{cons}(s(z), \text{cons}(s(s(z)), \text{nil})))} \\
& \quad \text{with } \sigma := \{F \mapsto \lambda x.+(x, x), H \mapsto s(z), T \mapsto \text{cons}(s(s(z)), \text{nil})\} \\
\rightarrow_{\rho_6} & \underline{\text{cons}(+(s(z), s(z)), \text{map}(\lambda x.+(x, x), \text{cons}(s(s(z)), \text{nil})))} \\
& \quad \text{with } \sigma := \{X \mapsto s(z), Y \mapsto s(z)\} \\
\rightarrow_{\rho_2} & \underline{\text{cons}(s(+(s(z), z)), \text{map}(\lambda x.+(x, x), \text{cons}(s(s(z)), \text{nil})))} \\
& \quad \text{with } \sigma := \{X \mapsto s(s(z))\} \\
\rightarrow_{\rho_1} & \underline{\text{cons}(s(s(z)), \text{map}(\lambda x.+(x, x), \text{cons}(s(s(z)), \text{nil})))} \\
& \quad \text{with } \sigma := \{F \mapsto \lambda x.+(x, x), H \mapsto s(s(z)), T \mapsto \text{nil}\} \\
\rightarrow_{\rho_6} & \underline{\text{cons}(s(s(z)), \text{cons}(+(s(s(z)), s(s(z))), \text{map}(\lambda x.+(x, x), \text{nil})))} \\
& \quad \text{with } \sigma := \{X \mapsto s(s(z)), Y \mapsto s(s(z))\} \\
\rightarrow_{\rho_2} & \underline{\text{cons}(s(s(z)), \text{cons}(s(+(s(s(z)), s(z))), \text{map}(\lambda x.+(x, x), \text{nil})))} \\
& \quad \text{with } \sigma := \{X \mapsto s(s(s(z))), Y \mapsto s(z)\} \\
\rightarrow_{\rho_2} & \underline{\text{cons}(s(s(z)), \text{cons}(s(s(+(s(s(z)), z))), \text{map}(\lambda x.+(x, x), \text{nil})))} \\
& \quad \text{with } \sigma := \{X \mapsto s(s(s(s(z))))\} \\
\rightarrow_{\rho_1} & \underline{\text{cons}(s(s(z)), \text{cons}(s(s(s(s(z))))), \text{map}(\lambda x.+(x, x), \text{nil})))} \\
& \quad \text{with } \sigma := \{F \mapsto \lambda x.+(x, x)\} \\
\rightarrow_{\rho_5} & \underline{\text{cons}(s(s(z)), \text{cons}(s(s(s(s(z))))), \text{nil})}
\end{aligned}$$

Figure 3-20: A reduction of an applied map operation for doubling onto a list of numbers [1, 2] in the PCRS of Figure 3-19.

3.7 Pure type systems

In the three preceding sections, we aided general purpose computing machinery with *types*, which are effectively sets of values. We now focus our attention to a class of type systems that enable the expression of rich concepts in a declarative and relational manner, rather than through direct computational construction — we look to representation at the level of *types*. Types as they are exploited in programming languages provide the following:

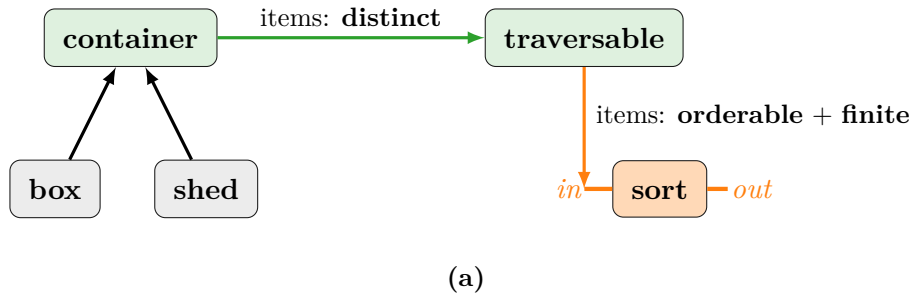
- *conceptual role semantics*, which we discuss in detail in [Chapter 5 Towards Formalized Conceptual Role](#), because types can have relations to other types where those relations are themselves types (hence meaning is derived from types and their relations);
- *no nonsense values* because illegal states are made unrepresentable by associating terms with sufficiently descriptive types; and
- *implementation is irrelevant* because illegal behavior cannot compile, where behavior is defined declaratively using a type.

Intuitionistic logic provides a framing of computation where a value (or program) is a constructive proof of some type²⁶ [[Kolmogorov, 1932](#); [Pierce, 2002](#); [Granström, 2011](#)]. With computation as consequence of types, we are justified in looking at types alone and may disregard “concrete” values or programs. We particularly look at the framework of *pure type systems* which generalizes a large class of typed λ -calculi. See [Figure 3-21](#) for examples of what is representable within a pure type system — *without writing any concrete procedure*.

To help make sense of pure type systems, we start with some type systems over which they generalize. The λ -cube diagrammed in [Figure 3-22](#) presents eight type systems upon a simply-typed λ -calculus (c.f. [Section 3.5.1 Simply typed lambda-calculus](#)). These type systems extend λ -calculus with features of *polymorphism*, *type operators*, and *dependent types*. We refrain from digging into the formal theory of these systems, instead giving a high-level exposition before discussing pure type systems: we explain what those three italicized features mean after an explanation of what they build upon:

²⁶ We mean constructive proof in the sense of mathematical logic: let a type represent some theorem, then a value (or program) of that type provides an existence proof for the theorem. This correspondence is known as the Curry-Howard isomorphism.

The law of the excluded middle — that every proposition satisfies $\neg P \vee P$ — is not assumed in intuitionistic logic because a proposition refers to computation, which is famously undecidable [[Chlipala, 2011](#); [Turing, 1937](#)].



$$\text{sort} : \text{Ord } T \Rightarrow \{i : \text{List } T\} \rightarrow \{v : \text{List } T \mid \text{elems}(i) = \text{elems}(v) \wedge \text{nondecreasing}(v)\}$$

Figure 3-21: (a) A schematic of conceptual relations that are representable in a declarative manner. It is straightforward to convert this schematic as illustrated here into types of a pure type system. (b) Sort defined with a dependent type: given a type T which is orderable (signified by the constraint $\text{Ord } T$), the procedure’s input i is a value of type $\text{List } T$, and the output type depends on the input value — it is the set of values v of type $\text{List } T$ that satisfy the conditions that the elements of v are equivalent to the elements of i and that v is a non-decreasing list.

- **value** \rightarrow **value** or *abstraction* in λ -calculus terms (c.f. [Section 3.5 Lambda-calculus](#)). This permits a representation of functions that take values and return values. Virtually every modern programming language features this. It provides procedural abstraction, which serves as a foundation of computation [[Abelson et al., 1996](#)].
- **type** \rightarrow **value** or *polymorphism* permits values that are determined by some type. For example, the identity function $(\lambda \mathbf{x} \mathbf{x})$ in a polymorphic system may have type $\forall \alpha. \alpha \rightarrow \alpha$ which effectively acts as the function which takes a type as its argument, $(\lambda \tau (\lambda (\mathbf{x} : \tau) \mathbf{x}))$ (where we adopt $\mathbf{x} : \tau$ to mean a value \mathbf{x} of type τ).
- **type** \rightarrow **type** or *type operators* permit types to be constructed using other types. For example, `Either` is a type operator that, when given two types α and β , yields a new type called “`Either α β` ” for which values can be either a value of type α or a value of type β ²⁷. With type operators comes *the kind system*, or essentially a type system for types. `Either` has kind $\text{TYPE} \rightarrow \text{TYPE} \rightarrow \text{TYPE}$ ²⁸, taking two types

²⁷ This statement is formally written as $\text{Either } \alpha \beta \triangleq \alpha \cup \beta$.

²⁸ Historically, the `TYPE` kind is written with a star `*`.

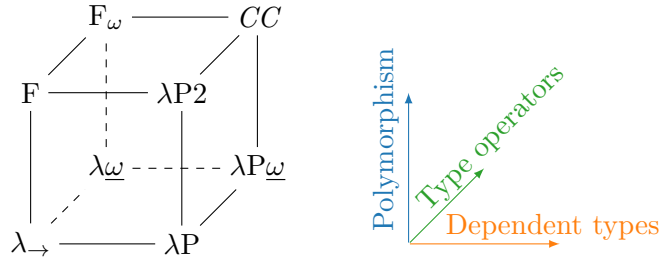


Figure 3-22: The Barendregt λ -cube with labeled axes on the right [Barendregt, 1991]. Each vertex is an enhancement from λ_{\rightarrow} (front-bottom-left), the simply-typed λ -calculus we described in Section 3.5.1 Simply typed lambda-calculus. System F (front-top-left) is polymorphic λ -calculus, a variant of which we presented in Section 3.5.2 Polymorphic typed lambda-calculus. The calculus of constructions (CC ; Coquand and Huet [1986]) has all three features indicated by the three axes of the cube: polymorphism, type operators, and dependent types.

and returning a new type. Another example of a type operator is `List`, with kind $\text{TYPE} \rightarrow \text{TYPE}$, where “`List τ` ” is a type for lists where all elements have type τ .

Type operators are often extended in practice to include *constraints*. Constraints work much like propositions in sequent calculus, where logical statements are conditioned on other logical statements [Gentzen, 1935; Kleene, 1952]. This is done in a type system by introduction of the `CONSTRAINT` kind and permitting types to not only be expressed with kind `TYPE`, but also a `TYPE` accompanied with a sequence of constraints. A constraint-returning type operator is called a *type family*. For example, `Ord` is a type family with kind $\text{TYPE} \rightarrow \text{CONSTRAINT}$, where `Ord τ` expresses the statement that τ is orderable — i.e. provides a method with type $\tau \rightarrow \tau \rightarrow \{\text{true}, \text{false}\}$ that is consistent with the mathematical notion of ordering (a binary relation that is transitive, reflexive, anti-symmetric, well-founded, and a connex). This is demonstrated in the example of `sort` in Figure 3-21b where $A \Rightarrow B$ means A is a type family and B is defined when A is satisfied. Such a conditional type, $A \Rightarrow B$, is akin to a *sequent* in sequent calculus.

- **value \rightarrow type** or *dependent types* permit types that depend on values. This typically takes form of first-order logic at the type-level where a value may be bound and quantified. There are two varieties of dependent types corresponding to universal and existential quantification over values. A *pi type* $\Pi_{(a:A)} B(a)$ is a dependent type that universally quantifies over values $a : A$ and provides the type according to B , an

operator which takes a *value* of type A and returns a type. An example of a pi type is sort in [Figure 3-21b](#), where the procedure’s input is universally quantified and the output is defined in terms of the input. The other variety is the *sigma type* $\Sigma_{(a:A)}B(a)$ which existentially quantifies over values $a : A$ and provides the type according to B in the form of a set of tuples $\{(a, B(a)) \mid a \in A\}$.

These features are all present in many proof systems and some programming languages. A language with such highly-expressive types opens the realm of *tactics*, which are methods for determining inhabitants of a type: they effectively reason from types into concrete code. This is one reason why proof systems are an apparent application of such a rich type system: you can provide a proof constructively, in terms of types, and check whether the types have inhabitants.

Every system in the λ -cube is naturally expressed in a pure type system: there is no need to introduce those features as special ad-hoc extensions to the type system. Pure type systems simplify these features by providing a generalized framework for systems with terms and types [[Barendregt, 1991, 1992](#); [Geuvers, 1993](#)].

Pure type systems derive judgements of the form

$$\Gamma \vdash M : A$$

where M and A are *pseudoterms*, which we will define shortly, and Γ is a finite sequence of declarations $x : B$ where x is a variable and B is a pseudoterm. These judgements express that M is of type A given the variables declared in Γ . A pseudoterm \mathcal{T} is defined in abstract syntax

$$\mathcal{T} \triangleq \mathcal{S} \mid \text{Var} \mid \mathcal{T}\mathcal{T} \mid \lambda \text{Var}:\mathcal{T}.\mathcal{T} \mid \Pi \text{Var}:\mathcal{T}.\mathcal{T}$$

where \mathcal{S} is the set of *sorts*, i.e. distinct universes that pseudoterms inhabit (we will see examples later in this section), and Var is the countably-infinite set of variables. For two pseudoterms M and A , MA is an *application* of two pseudoterms, and $\lambda x:A.B$ is an *abstraction* where the bound variable x is typed according to A in the body determined by B . Finally $\Pi x:A.B$ is the Π -*type* (or *dependent product*), where

$$\Pi x:A.B \triangleq \{f \mid \forall a:A. fa : B[a/x]\}$$

adopting the notation $B[N/x]$ for the substitution of free variable x in B with N . Because of the variables bindings of λ and Π , we adopt $\text{FV}(M)$ for the set of free variables of pseudoterm M . The Π -type $\Pi x : A.B$ represents a total function over domain A and range of B with the substituted variable according to the argument of the function. This means that if $x \notin \text{FV}(B)$, then $\Pi x : A.B$ is simply an arrow type $A \rightarrow B$ (c.f. [Section 3.4.1 Polymorphic typed combinatory logic](#)).

We adopt the common notation for inference rules, where premises above the line derive the conclusion below the line [[Pierce, 2002](#)]. The inference rules for typed λ -calculus, where sort s is ranged over \mathcal{S} and variable x is ranged over Var , are:

$$\frac{\Gamma, x:A \vdash M : B \quad \Gamma \vdash \Pi x:A.B : s}{\Gamma \vdash \lambda x:A.M : \Pi x:A.B} \quad (\text{abstraction})$$

$$\frac{\Gamma \vdash M : \Pi x:A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]} \quad (\text{application})$$

which provide judgements for well-typed abstraction and application. Semantics for variable declarations are also defined with inference rules:

$$\frac{\Gamma \vdash A : s}{\Gamma, x:A \vdash x : A} \quad \text{if } x \notin \Gamma \quad (\text{variable})$$

$$\frac{\Gamma \vdash B : s \quad \Gamma \vdash M : A}{\Gamma, x:B \vdash M : A} \quad \text{if } x \notin \Gamma \quad (\text{weakening})$$

It follows that through variable declarations in Γ , every variable belong to a sort. A pure type system is instantiated by defining \mathcal{S} , $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$, and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$, which provide the following inference rules:

$$\vdash s_1 : s_2 \quad \text{if } (s_1, s_2) \in \mathcal{A} \quad (\text{axiom})$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x:A \vdash B : s_2}{\Gamma \vdash \Pi x:A.B : s_3} \quad \text{if } (s_1, s_2, s_3) \in \mathcal{R} \quad (\text{product})$$

We finally have one more rule for β -conversion (defined in [Section 3.5 Lambda-calculus](#)), to indicate that β -equivalent types have the same inhabitants:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B} \quad \text{if } A =_\beta B \quad (\text{conversion})$$

A *pure type system* is defined by the triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, and provides the seven inference rules described above. It is commonplace for product rules $(s_1, s_2, s_3) \in \mathcal{R}$ to take $s_2 = s_3$ — effectively meaning that functions have the same sort as their range — so we adopt that convention here. Each corner of the λ -cube (Figure 3-22) is defined by a pure type system with sorts $\mathcal{S} = \{\text{PROP}, \text{TYPE}\}$, corresponding to the sort of all “values” (propositions) and the sort of all “types” respectively, and axiom $\mathcal{A} = \{\text{PROP} : \text{TYPE}\}$ that all values are typed, with product rules according to Figure 3-23.

System	\mathcal{R}			
λ_{\rightarrow}	(PROP, PROP)			
F	(PROP, PROP)	(TYPE, PROP)		
λP	(PROP, PROP)		(PROP, TYPE)	
$\lambda\omega$	(PROP, PROP)			(TYPE, TYPE)
$\lambda P2$	(PROP, PROP)	(TYPE, PROP)	(PROP, TYPE)	
F_{ω}	(PROP, PROP)	(TYPE, PROP)		(TYPE, TYPE)
$\lambda P\omega$	(PROP, PROP)		(PROP, TYPE)	(TYPE, TYPE)
CC	(PROP, PROP)	(TYPE, PROP)	(PROP, TYPE)	(TYPE, TYPE)

Figure 3-23: Product rules for the pure type system of each corner of the λ -cube.

The four rules each correspond to the four features described earlier:

- (PROP, PROP) provides abstraction, so terms may depend on terms;
- (TYPE, PROP) provides polymorphism, so terms may depend on types;
- (TYPE, TYPE) provides type operators, so types may depend on types;
- (PROP, TYPE) provides dependent types, so terms may depend on types.

We can further express logic with a pure type system, via the Curry-Howard isomorphism, by expressing formulas (propositions, predicates) as types. With sorts $\mathcal{S} = \{\text{PROP}, \text{PRED}\}$ and axiom $\mathcal{A} = \{\text{PROP} : \text{PRED}\}$ (effectively that predicates are sets of propositions), we have first, second, and higher-order propositional logic from the pure type systems with rules according to Figure 3-24, with correspondence:

- (PROP, PROP) provides implication ($P \supset Q$ where $P : \text{PROP}$ and $Q : \text{PROP}$);
- (PRED, PROP) provides quantification over predicates ($\forall x:A.B$ where $A : \text{PRED}$);

System	\mathcal{R}		
λPROP	(PROP, PROP)		
λPROP2	(PROP, PROP)	(PRED, PROP)	
$\lambda\text{PROP}\omega$	(PROP, PROP)	(PRED, TYPE)	(PRED, PRED)

Figure 3-24: Product rules for the propositional logics, where λPROP , λPROP2 , and $\lambda\text{PROP}\omega$ are isomorphic to first-, second-, and higher-order propositional logic, respectively.

- (PRED, PRED) provides predicates that depend on predicates.

Predicate logic can be defined with the sorts $\mathcal{S} = \{\text{PROP}, \text{PRED}, \text{SET}, \text{TYPE}^s\}$ corresponding to propositions, predicates, sets, and types of sets, and axioms $\mathcal{A} = \{\text{PROP} : \text{PRED}, \text{SET} : \text{TYPE}^s\}$ with the product rules shown in [Figure 3-25](#). Higher-order predicate logic simply introduces products that take predicates.

System	\mathcal{R}			
		(SET, SET)		
λPRED	(PROP, PROP)	(SET, PROP)		
		(SET, PRED)		
		(SET, SET)	(PRED, SET)	
$\lambda\text{PRED}\omega$	(PROP, PROP)	(SET, PROP)	(PRED, PROP)	
		(SET, PRED)	(PRED, PRED)	

Figure 3-25: Product rules for predicate logics, where λPRED and $\lambda\text{PRED}\omega$ are isomorphic to first-order and higher-order predicate logic, respectively.

As a final example of a pure type system, constructive higher order logic as in $\lambda\text{PRED}\omega$ may be succinctly expressed using the system λHOL of [Figure 3-26](#). Pure type systems are often written with the notation of \star , \square , and \triangle as sorts rather than words.

$$\begin{array}{l}
\mathcal{S} \quad \star, \square, \triangle \\
\mathcal{A} \quad \star : \square, \square : \triangle \\
\mathcal{R} \quad (\star, \star), (\square, \square), (\square, \star)
\end{array}$$

Figure 3-26: λHOL , a pure type system isomorphic to constructive higher-order logic.

Pure type systems may also be used to construct inconsistent systems, which is not necessarily problematic for our intended use: to represent concepts. For example, λHOL

with the added rules (Δ, \square) and (Δ, \star) allow for polymorphism and quantification over collections of sets/predicates/types (i.e. second-order types) — this is called *System U* or λU and it is inconsistent [Coquand, 1986]. Despite its inconsistency, λU is useful for the expression of concepts such as reflexivity²⁹:

$$\vdash \lambda A:\square. \lambda R:A \rightarrow A \rightarrow \star. \Pi x:A. Rxx \quad : \quad \Pi A:\square. \Pi R:A \rightarrow A \rightarrow \star. \star$$

with arrows as shorthand for Π -types whose bindings are unused (i.e. $A \rightarrow B \triangleq \Pi a:A. B$ if $a \notin \text{FV}(B)$). Given a type and a binary relation over that type, this expresses the proposition that the given binary relation is reflexive.

We described earlier that *tactics* are a practical means for determining inhabitants of a type, present in many programming languages with dependent types. We take particular interest in programming languages such as Idris because of a feature called “elaborator reflection,” where tactics may be expressed within the language itself, rather than as an opaque intrinsic [Brady, 2013; Christiansen and Brady, 2016]. Such a feature makes the distinction between a language and its runtime less salient, which we believe is important in a representation of concepts: it enables *learning to learn* to be expressed in an unconstrained manner at the object level, contrary to common approaches of rigid prebuilt meta-learning faculty with constrained spaces of learnable parameters.

That is why we look towards a class of languages with pure type systems as a representation for concepts. Not only do they live atop Turing-complete languages such as those we explored in the previous sections, they also (I) provide first-class objects for *declaratively* defining values using types, and similarly defining types using higher-order types, where (II) those relations aren’t limited to constructs like polymorphism, type operators, dependent types, or even higher-order logic³⁰, and they can further (III) provide the ability to represent search procedures over terms in the language, *using* terms in the language.

²⁹ This statement of reflexivity relies on the rule (Δ, \star) , which actually gives a consistent system atop λHOL . The inclusion of (Δ, \square) breaks consistency by Girard’s paradox [Hurkens, 1995]. A term that uses this product rule is the polymorphic identity on types:

$$\vdash \lambda A:\square. \lambda a:A. a \quad : \quad \Pi A:\square. A \rightarrow A$$

³⁰ This is perhaps with a loss of consistency, as in the system λU . Inconsistency is acceptable because we are not modeling the foundations of mathematics which are necessarily rigorous.

Chapter 4

Concept Learning by Design: Program Induction

4.1 Overview

In this chapter, we present *program induction* as a scheme for both problem-solving and concept learning. Program induction is, as the name suggests, the learning of a program from some incomplete information on the nature of the program. Program learning with sparse data is interesting to the cognitive science community because of its correspondence to *concept learning*, a class of learning tasks studied in cognitive psychology which humans naturally encounter and in which they perform well [Gweon et al. \[2010\]](#); [Schulz \[2012b\]](#); [Lake et al. \[2017\]](#). Unless otherwise noted, we assume a program exists in a representation like those described in [Chapter 3 Representations](#).

We start in [Section 4.2](#) by discussing a variety of notable learning tasks that are well-expressed in this scheme. In [Section 4.3](#), we acknowledge and consider resolutions to the problem of combinatorial explosion that is present in many program induction problems. In [Section 4.4](#) we discuss various learning architectures for the program induction scheme. In [Section 4.5](#) we outline *Bayesian program learning* and motivate it as a useful framework for concept learning as program induction. In [Section 4.6](#) we present our work on the problem of *language bootstrapping*, and in [Section 4.7](#) we present our work on the problem of *domain-general learning*.

We remark that the problem of concept acquisition, which we here frame as program

induction, is vital to any theory of concepts [Gopnik and Meltzoff, 1997; Carey, 2009, 2015].

4.2 Learning tasks

The kinds of problems that can be solved (or at least approached) with program induction are extensive. In this section we briefly discuss a few of these learning tasks.

<i>Input</i>	<i>Output</i>
Dr. Eran Yahav	Yahav, E.
Prof. Kathleen S. Fisher	Fisher, K.
Bill Gates, Sr.	Gates, B.
George Ciprian Necula	Necula, G.
Ken McMillan, II	McMillan, K.

aabc	→	aabd
ijkk	→	?
abc	→	abd
mrrjjj	→	?
abc	→	abd
xyz	→	?

(a)
(b)

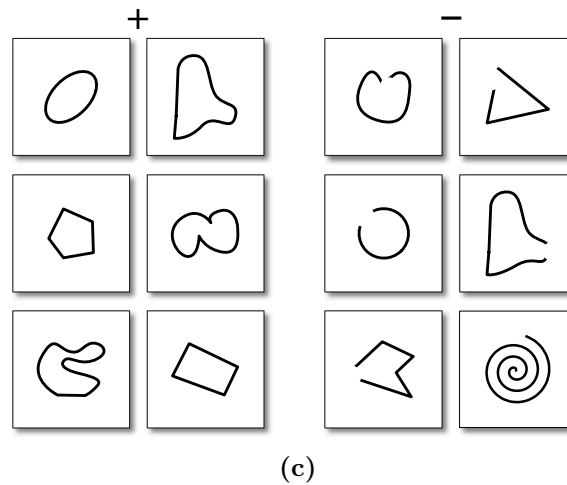


Figure 4-1: Learning from examples. (a) String transformation problems as present in spreadsheets [Gulwani, 2011]. Noisy cases of string transformation problems have also been explored [Devlin et al., 2017b]. The string transformation problems are typically presented as program *induction* problems, where one only needs to compute outputs given new inputs directly rather than synthesize a program. (b) Analogy-making, a form of one-shot learning that is based on program *induction*, as it need not explain the analogy to complete the problem [Hofstadter and Mitchell, 1995]. (c) The Bongard problems provide positive and negative examples for a program which discriminates objects in the left-hand-side from the right-hand-side [Bongard, 1970]. This is a program *synthesis* problem, as the solution is intended to be succinctly explainable.

1. **Learning from examples.** This is a dataset-oriented problem of program induction. A dataset \mathcal{X} consists of a number of examples, usually in the form of input/output pairs $(x_i, y_i) \in \mathcal{X}$ such that a program computes the corresponding output for every input. Variations of this task include the permission of noisy data where the input/output pairs may not be perfect models of the target program, or where a negative dataset $\bar{\mathcal{X}}$ of examples which are not true for the program is provided, or k -shot learning where the dataset is small $|\mathcal{X}| = k$. Two perspectives on this task are “program synthesis” and “program induction” which, though often equivalent (and hence we may use either term in settings where they are not distinguished), are different for learning procedures which do not produce symbolic programs: program *synthesis* is the generation of a symbolic program, whereas program *induction* is the computation of output from input. The distinction is made apparent in [Section 4.4.9 Neural network optimization](#). Popular cases of learning from examples are illustrated in [Figure 4-1](#).

2. **Learning from traces.** This is also dataset-oriented, but rather than the data being first-class objects (i.e. inputs and outputs) to the target program, it is instead a description of the program’s procedure itself. This can be represented with sequences of low-level instructions that are carried out when a program is executed, and the corresponding state transformations (i.e. memory effects).

3. **Learning via “blackbox” evaluation.** Under this paradigm, a likelihood model $\mathbb{P}[x|p]$ is provided which scores a task x given a particular program p . For some problems, the likelihood model can be used as a search aid to make program synthesis more tractable¹.

4. **Identifying valid inputs.** Given a program description (whether by the program itself, by examples, or by other means), this task is to provide a discriminative model which can determine whether an input is valid for the program. For example, a program that sorts a list of numbers is incompatible with an input that is a word².

¹ For example, if the target program adds five to an input, then a likelihood model for single input x of $\log \mathbb{P}[x|p] = -|p(x) - x - 5|$ may improve convergence speed of the search procedure.

² While many programming languages sort characters of a word in alphabetical or “ASCIIbetical” order, this generalization of sorting may not be outlined in the task’s description of the sorting program.

5. **Generating valid inputs.** Contrary to the discriminative model above, a generative model over inputs allows for the sampling of inputs which a discriminative model would accept. In particular, a *good* generative model for inputs should be capable of sampling inputs subject to certain conditions. For example, given the condition that a particular input is a list of length four, the generative model should be able to produce an input that satisfies the condition without necessitating the use of *rejection sampling*, where unsatisfactory samples are simply dropped until a satisfactory sample is achieved.

6. **Active learning and intentional input generation.** This is an interactive process usually based upon the “learning from examples” task. Given few examples, the machine is tasked with querying an *oracle* that can produce a new output on any input. These new examples should be specifically chosen to better determine the underlying program by demonstrate paradigmatic program behavior and resolving ambiguity. There are two major variants of this task: one where the machine knows the true program and must provide examples that efficiently aid some inductive learning process, and another where the machine does not know the true program and must use inductive machinery to both infer the program and hypothesize effective queries. From this task, *adversarial input generation* follows where inputs are generated that demonstrate flaws in an induced program.

7. **Determining restrictions on program output.** Given a program description, this task is to understand the nature of the program outputs. It may be in the form of specifying an output space of the program, or in the form of constraints on outputs depending on some inputs. For example, sorting a list of length five should always produce a list of length five.

8. **Determining conceptual dependence between programs.** Given multiple program descriptions, this task is to produce a directed graph that expresses whether a particular program is in some manner dependent on another program. For example, sorting may be conceptually dependent on taking the minimum of a list, while being conceptually independent from performing elementwise multiplication on a list.

9. **Self-optimization.** Given multiple program induction problems, this task is to improve an inductive mechanism to more efficiently solve those problems. For example, this may be realized by updating search heuristics or by changing the space in which search is preformed.

In order to approach one of these tasks, researchers often synthesize their own special-purpose datasets that are integrated with their particular models, making comparison difficult between different learning mechanisms. We provide an interactive dataset we call LIST-ROUTINES, available at <https://lucasem.github.io/list-routines>³, which can be used to assess a model’s performance on many of these program induction tasks. When we constructed LIST-ROUTINES, our intent was to have a dataset that people could perform reasonably well upon rather than a dataset that is entirely machine-directed. We crafted a large number of numerical and sequence-based concepts, engineering them to each have generative distributions over inputs and strong constraints on their behavior. By connecting different concepts whose constraints align, we compose these “subroutine” concepts into complex “routines” which are expressed with a directed graph of computation.

4.3 Combinatorial explosion

When working with a program induction mechanism, one must decide what the *priors* are: most often, this is the determination of a programming language that the machine uses to represent programs. This may be thought of as a choice of low-level assembly language versus high-level languages like Python or LISP, but independent of this underlying representation choice is the choice of *domain-specificity*. Domain-specificity is the extent to which a language is engineered to express procedures in a particular domain. Such an engineered language is called a *domain-specific language* (DSL). Having a DSL aids in the *combinatorial explosion* that is program search: the space of programs that have description length⁴ twice that of another (simpler) space is exponentially larger and hence, exponentially harder to search.

Using a DSL constrains this search space by making relevant procedures have low description length, much like how software libraries make for simpler programs targeting some

³ Source code is located at <https://github.com/lucasem/list-routines>.

⁴ The description length of a program is a measure of the bits needed to express the program in some representation.

use-case. This is called the *syntactic bias*. Rather than reinventing immensely sophisticated libraries, programmers adopt existing libraries. For many program induction approaches, a hand-engineered DSL is used to constrain the machine’s search problem. However, an apparent limitation to this is that the resulting system is only effective for that particular domain, and a poorly engineered DSL results in impoverished problem-solving ability.

The syntactic bias extends to the program representation itself. For example, combinatory logic (c.f. [Section 3.4 Combinatory logic](#)) does not introduce variable bindings, leading to a bias towards reduced flow of information across a program (every value must be explicitly passed around the syntax tree). λ -calculus, on the other hand (c.f. [Section 3.5 Lambda-calculus](#)), *does* introduce variable bindings, leading to a bias towards the computational procedures themselves.

Not only is the syntactic bias used to mitigate the combinatorial explosion of program search, but so too are other inductive biases and search heuristics. The next section describes a number of learning architectures for program induction that are for-the-most-part representation-agnostic. They instead tackle the search problem head-on, while providing approaches to some of the tasks described in the previous section.

4.4 Learning architectures

4.4.1 Overview

In this section, we discuss various approaches for inducing programs: [Enumerative search](#), [Constraint satisfaction](#), [Deductive search](#), [Inductive search](#), [Inductive logic programming](#), [Markov chain Monte Carlo](#), [Genetic programming](#), and [Neural network optimization](#). We liken program induction to *program synthesis* in cases where a symbolic program is produced by the learning mechanism.

4.4.2 Enumerative search

Enumerative search begins with a structured discrete search space and enumerates programs, often with an efficient pruning procedure. It is a straightforward approach to program synthesis using the “generate-and-test” methodology where the efficacy of a program is determined once it has been enumerated. Efficient enumerative program synthesizers have shown success in program synthesis competitions, such as `MAGICHASKELLER` and `Unagi`

[Katayama, 2005; Akiba et al., 2013]. An example of enumerative search in a context-free grammar (c.f. Section 3.3 Context-free grammars) is demonstrated in Figure 4-2.

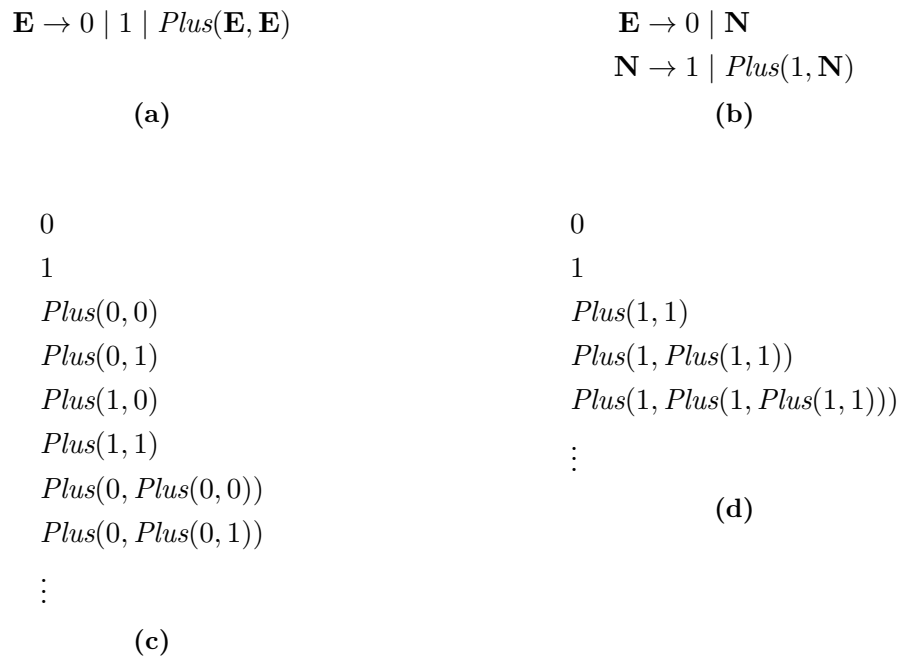


Figure 4-2: (a) A context-free grammar for addition, in which we can enumerate numbers. (b) a more sophisticated grammar over the same space, designed for more efficient enumeration by breaking symmetries: (i) a production of *Plus* cannot have 0 as one of its children and (ii) a production of *Plus* cannot have another *Plus* production as its left child (only as its right child). (c) Enumerating from the simple grammar. (d) Enumerating from the sophisticated symmetry-breaking grammar.

Strategies for making enumeration more efficient include the following:

1. **Depth limits** provide Occam’s razor for programs and prevent programs from spending much time searching in a small subset of the intended program space.
2. **Iterative-deepening with equivalence pruning** uses shorter already-generated programs to construct new programs of a given length, so that programs can be asserted non-redundant before they go into the pool to be used during the next (deeper) iteration.
3. **Symmetry-breaking** can take the form of a modified grammar, but is typically regarded as a set of forbidden rules. For example, a (sub-)program like `(if true ...)` should never be enumerated as it would be a waste of time, as with a program

like $(+ 0 \dots)$. Symmetry-breaking can also prevent the problem of a symmetric function such as *Plus* being enumerated again with the same material on its left as on its right at some later period during enumeration. Examples of symmetry-breaking are in [Figure 4-2](#).

4. **Probabilistic best-first** enumeration, where each rule in a context-free grammar is associated with a production probability and enumeration starts with a priori most probable expressions.

Although enumeration is often understood with context-free grammars, it may be used for any language on which a total ordering of programs can be constructed⁵. This becomes more complicated when introducing type systems that are not simple⁶. Type systems provide a natural constraint over the space of programs, so whenever enumerating typed programs is not especially costly⁷, it should be preferred over untyped enumeration.

Our own work provides implementation for enumerative search on probabilistic context-free grammars and on λ -calculus with polymorphic types as described in [Section 3.5.2 Polymorphic typed lambda-calculus](#), both with code available at <https://github.com/lucasem/program-induction>⁸.

4.4.3 Constraint satisfaction

Constraint programming technologies, particular SAT⁹ and SMT¹⁰ solvers, have demonstrated utility in the program synthesis setting. Their use falls into four varieties of problems [[Torlak and Bodik, 2013](#)]:

1. **Program verification** can be as simple as evaluating a program in a set of pre-specified input/output examples, or more sophisticated such as by generating novel

⁵ This may seem like a difficult assertion, but in practice it is easy to construct such an ordering for any language that can be defined formally.

⁶ Where *simple* refers to simple types — c.f. [Section 3.4.1 Polymorphic typed combinatory logic](#).

⁷ Empirically, we found that more time is spent evaluating programs than enumerating them, even with the scaffolding of polymorphic types and type inference using the Hindley-Milner type system.

⁸ Documentation is located at <https://docs.rs/programinduction>

⁹ Boolean satisfiability. In short, the SAT problem proves whether there exists an assignment to a set of Boolean values which yield TRUE for a propositional formula. For example, $(a \vee b) \wedge (\neg a \vee \neg b) \wedge c$ is satisfied by the assignment $\{a \mapsto \text{TRUE}, b \mapsto \text{FALSE}, c \mapsto \text{TRUE}\}$.

¹⁰ Satisfiability modulo theories. What SAT does for propositional logic, SMT does for first-order logic by effectively replacing Boolean values in a SAT problem with predicates over potentially non-Boolean values. For example, $(x \leq y) \wedge ((x^2 + y^2 = 25) \vee (x = 2)) \wedge (\exists z \in \mathbb{N}. z^2 - z = y)$ is satisfied by $\{x \mapsto 2, y \mapsto 6\}$.

inputs which help assert that a specification is met, or by asserting other program properties than just its functional relation.

2. **Program synthesis** uses constraints such as input/output examples and predetermined program structure to synthesize a program that meets those criteria.
3. **Bug localization** by identifying the maximal set of constraint clauses — corresponding to execution flow of the program — that satisfy every input/output example, and taking its complement [Jose and Majumdar, 2011].
4. **Efficient debugging** by determining whether an expression within a program may be replaced with a runtime value that yields a valid program, thereby constraining the space of repair programs for a synthesis task [Chandra et al., 2011].

Program synthesis by *sketching* takes a high-level program that may have implementation holes and fills in the missing implementation [Solar-Lezama et al., 2006]. The SKETCH system uses counter-example guided inductive synthesis (CEGIS) to complement an inductive synthesizer by generating inputs which exhibit bugs in a synthesized program, where both the inductive synthesizer and the CEGIS validator are optimized and translated to SAT constraints [Solar-Lezama, 2008]. An alternative constraint-based approach uses an SMT solver to synthesize a program that matches given input/output examples and attempts to synthesize *another* distinct program which also matches the input/output examples but has different output for some “distinguishing” input — the input is then passed into an *oracle* to obtain the desired program’s output before repeating the process, or if no such distinct program (and hence, distinguishing input) exists then the synthesis task terminates [Jha et al., 2010].

Constraint-based synthesis has since seen development in a probabilistic framing, as a method of inference in a generative model. One approach performs unsupervised joint inference over programs and inputs given a domain-specific language and a prior distribution over programs, by compiling both “soft” probabilistic constraints with “hard” program space constraints into a set of clauses for an SMT solver [Ellis et al., 2015]. The TERPRET probabilistic programming language provides inference algorithms based on gradient descent, integer linear programming, SMT solving, and the SKETCH system described above — the latter two constraint-based methods were found to outperform the other methods [Gaunt et al., 2016].

Another approach uses neural networks to parse noisy images into constraints for synthesizing graphics programs [Ellis et al., 2017]. Program synthesis approaches that are robust to noise are discussed in other architectures (differentiable inductive logic programming in Section 4.4.6 Inductive logic programming; neural program learning in Section 4.4.9 Neural network optimization) — dealing with noisy data in the constraint satisfaction setting is difficult and not well-explored.

4.4.4 Deductive search

Deductive search recursively reduces synthesis problems into subproblems à la divide-and-conquer by leveraging sophisticated information expressed in examples [Gulwani et al., 2017]. We look at the approaches of *inverse semantics* and *type-directed synthesis*.

Inverse semantics associates domain-specific operators with *witness functions* that transform constraints/examples for the operator into constraints/examples for synthesis subproblems. For example, given an operator F which takes two arguments a and b , and a specification (i.e. constraints and examples) ϕ , we are tasked with finding the a and b such that $F(a, b) \models \phi$. A witness function ω_F for F transforms this problem into two problems $\omega_F(\phi) \mapsto (\phi_a, \phi_b)$ which take the form of new synthesis tasks $a \models \phi_a$ and $b \models \phi_b$ ¹¹. This amounts to effectively inverting the operator F so that an input/output pair for F entails input/output pairs for a and b . In practice, witness functions are relaxed to provide more useful and efficient deductive search procedures [Polozov and Gulwani, 2015].

Type-directed synthesis is a family of deductive search approaches that expresses problems using *types* rather than examples. The synthesis problem is therefore to constructively prove that a type has an inhabitant. These approaches recursively synthesize partial programs, starting from an empty program, according to types that were either explicitly defined initially or deduced during synthesis. SYNQUID represents these types in a type system that corresponds to an intuitionistic logic per the Curry-Howard correspondence [Polikarpova et al., 2016]. Other work has adopted intuitionistic theorem-proving techniques — primarily sequent calculus — as well as logical operations on types such as disjunction, union, and negation to yield a framework in which a task can be specified using a combination of examples and type constraints [Frankle et al., 2016]. Using a sophisticated

¹¹ This is a different formulation than the one presented in Polozov and Gulwani [2015]. Nonetheless, it presents the role of witness functions.

type system with polymorphism, type operators, and dependent types, permits expressive declarative definition of procedures at the type-level. See Figure 4-3 for a type-directed synthesis task defined purely by a type rather than examples.

<pre> -- given foldr data OList α where Nil::OList α Cons::$x:\alpha \rightarrow$ OList $\{\alpha x\leq v\} \rightarrow$ OList α sort::$xs:\text{List } \alpha \rightarrow \{\text{OList } \alpha \text{elems } v = \text{elems } xs\}$ </pre>	<pre> sort = $\lambda xs .$ foldr f Nil xs where f = $\lambda t . \lambda h . \lambda acc .$ match acc with Nil \rightarrow Cons h Nil Cons z zs \rightarrow if h \leq z then Cons h (Cons z zs) else Cons z (f zs h zs) </pre>
(a)	(b)

Figure 4-3: Type-directed synthesis of `sort`. (a) The task is to find an inhabitant of the `sort` type. Terms of the form $\{\tau|P(v)\}$ refer to a *refined* type, whose inhabitants are every value v of type τ that satisfies the predicate $P(v)$. (b) The program synthesized by SYNQUID [Polikarpova et al., 2016].

We find type-directed synthesis to be a very motivating approaching to inducing programs that complements our interest in representations based on types, as described in Section 3.7 Pure type systems. Type-directed synthesis provides efficient methods for filling in implementations details. From the perspective of computer-aided programming, it enables programmers to express entire an computational workflow at the type-level, where a machine either provides the programmer with a concrete implementation of the specified computing system or it informs the programmer that the specification is invalid¹². We discuss more motivations for our interest in type-directed synthesis in Chapter 5 Towards Formalized Conceptual Role.

4.4.5 Inductive search

Inductive inference consists of generalizing from examples. Every inductive inference algorithm requires a priori assumptions, or an *inductive bias*: these inductive biases may take form in either a language representation or in a search procedure¹³. Every learning architecture we discuss has inductive biases in some form. In the section, we consider inductive

¹² Invalid here could mean many things: preferably it means the types are proven to have no inhabitants and hence the specification is *inconsistent*, but it could be the case that the types result in an *undecidable* concrete resolution, which could always happen in the setting of intuitionistic logic. If the latter is the case, the machine might be unable to inform the programmer of the specification’s invalidity because it could be caught in a non-terminating loop.

¹³ Some of the neural program synthesizers from Section 4.4.9 Neural network optimization learn a latent representation for programs, so their inductive biases don’t quite fit this statement. Those biases are usually characterized by the problem statement and the particular neural network architecture.

approaches that do not fit into the other discussed architectures.

FLASHFILL uses a domain-specific language of string transformations based on regular expressions: during inductive synthesis, examples are solved independently and those solutions are partitioned and classified in order to merge the particular solutions to a general synthesized program for all examples [Gulwani, 2011]. *Analytical* inductive programming in the IGOR2 system start with a term rewriting system (c.f. Section 3.6 Term rewriting) where the target function is expressed by a single rule with a left-hand side (lhs) that pattern matches on all examples, and an iterative induction process either splits a rule into others with more-refined lhs patterns, introduces auxiliary functions as new rules that also get induced, or introduces function calls as alterations to the right-hand side of a rule [Hofmann and Kitzelmann, 2010; Schmid and Kitzelmann, 2011]. The λ^2 synthesis tool combines many approaches we’ve discussed in this section: inductive generalization produces type-aware sketches¹⁴ from examples by leveraging hand-crafted domain knowledge, deductive procedures both refute sketches which are incompatible with examples and infer sub-examples to guide search within a sketch, and finally enumeration fills in sketches subject to the deduced sub-examples and the overall program’s examples¹⁵ [Feser et al., 2015]. Figure 4-4 shows examples and a program that λ^2 induces.

<pre> dropmins [] ↦ [] [[1]] ↦ [[]] [[1, 3, 5], [5, 3, 2]] ↦ [[3, 5], [5, 3]] [[8, 4, 7, 2], [4, 6, 2, 9], [3, 4, 1, 0]] ↦ [[8, 4, 7], [4, 6, 9], [3, 4, 1]] (a) </pre>	<pre> dropmins x = map f x where f y = filter g y where g z = foldl h False y where h t w = t (w < z) (b) </pre>
---	--

Figure 4-4: λ^2 learns the function `dropmins` from four examples [Feser et al., 2015].

4.4.6 Inductive logic programming

Inductive logic programming (ILP) involves learning a first-order logical program from examples and background knowledge [Muggleton, 1991; Raedt, 2008]. Equipped with an

¹⁴ A sketch, recall from Section 4.4.3 Constraint satisfaction, is a partial program which may implementation holes.

¹⁵ This whole process is recursive: there may be sketches within sketches.

interpretation, which maps from the space of logic programs to into some set of objects, ILP learns by finding hypotheses in the search space over logic programs which satisfy a criterion or minimize a loss function. Typically the framing is that there are a number of positive and negative examples of some target predicate, atop some background knowledge in the form of a true statements that do not directly involve the target predicate, and the goal is to find the simplest consistent hypothesis which models the positive examples and not the negative examples. An ILP problem is shown in [Figure 4-5](#).

$$\begin{aligned} \mathcal{B} &= \{\text{mother}(i, a), \text{father}(a, b), \text{father}(a, c), \text{father}(b, d), \text{father}(b, e), \\ &\quad \text{mother}(c, f), \text{mother}(c, g), \text{mother}(f, h)\} \\ \mathcal{P} &= \{\text{target}(i, b), \text{target}(i, c), \text{target}(a, d), \text{target}(a, e), \text{target}(a, f), \\ &\quad \text{target}(a, g), \text{target}(c, h)\} \\ \\ &\quad \text{target}(X, Y) \leftarrow \text{invented}(X, Z), \text{invented}(Z, Y) \\ &\quad \text{invented}(X, Y) \leftarrow \text{father}(X, Y) \\ &\quad \text{invented}(X, Y) \leftarrow \text{mother}(X, Y) \end{aligned}$$

Figure 4-5: The target concept represents the *grandfather* relation. Lowercase letters a, b, \dots represent people. \mathcal{B} is the background knowledge, \mathcal{P} is the set of positive examples, and all other ground atoms involving the target concepts are the negative examples. The clauses on the bottom are learned. The “invented” predicate may be invented to provide a more general rule for the target.

Search techniques in ILP primarily include inverse resolution, relative least-general generalization, Θ -subsumption, and inverse entailment [[Raedt, 2008](#)]. *Resolution* expands clauses C_1 using other clauses C_2 to obtain new clauses C_3 , so *inverse resolution* is the use of C_1 and C_3 to obtain C_2 — it may also be regarding is a logic program form of inverse inlining [[Muggleton and Buntine, 1988](#)]. Relative least-general generalization is an approach whose goal is solely to maximally compress positive examples. Θ -subsumption is a relationship between two clauses c_1 and c_2 where, using a substitution over variables θ , c_1 may be transformed into a statement which is implied (or “entailed”) by c_2 — it effectively provides a generality relation $g \models s$ [[Plotkin, 1970](#)]. Inverse entailment is a method for finding clauses which imply other clauses [[Muggleton, 1995](#)].

Meta-interpretive learning is a technique for ILP using higher-order existentially-quantified statements, which support learning compositional predicates, such as the solutions in

Figure 4-5 which feature the invented predicate “parent” to define the target predicate, as well as recursive definitions such as in Figure 4-6 [H. et al., 2013; Muggleton et al., 2014]. Other recent work in ILP has adopted *differentiable* approaches that enable training by gradient descent (c.f. Section 4.4.9 Neural network optimization), yielding robust predicate invention despite noisy data [Yang et al., 2017; Rocktäschel and Riedel, 2017; Evans and Grefenstette, 2018] and generalization by compressed representation learning [Campero et al., 2018].

$$\begin{aligned} \text{even}(0) &\leftarrow \\ \text{even}(A) &\leftarrow \text{invented1}(A, B), \text{invented2}(B) \\ \text{invented1}(A, B) &\leftarrow \text{succ}(B, A) \\ \text{invented2}(A) &\leftarrow \text{invented1}(A, B), \text{even}(B) \end{aligned}$$

Figure 4-6: A mutually recursive procedure for the *even* relation, learned with meta-interpretive learning. The “invented1” predicate corresponds to the *predecessor* relation, and the “invented2” predicate corresponds with the *odd* relation.

4.4.7 Markov chain Monte Carlo

Markov chain Monte Carlo (MCMC) methods are common for sampling programs in a Bayesian framing. MCMC provide randomized general-purpose means for approximate inference for sophisticated generative models, in practice typically using the Metropolis-Hastings (MH) algorithm [Metropolis et al., 1953]. These methods essentially work by stochastically generating proposals to move around in an hypothesis space towards preferable hypotheses, demonstrated in Figure 4-7. In the Bayesian inference framing with prior $\mathbb{P}(\theta)$ and likelihood $\mathbb{P}(D|\theta)$, the search for some hypothesis θ given data D involves sampling from the posterior distribution:

$$\mathbb{P}(\theta|D) \propto \mathbb{P}(D|\theta) \mathbb{P}(\theta)$$

An important feature of MCMC from a cognitive perspective is that it’s purpose is not simply to find the *best* solution, but rather to explore a *space* of potential hypotheses. A learner may not have a way to determine whether their current candidate hypothesis is

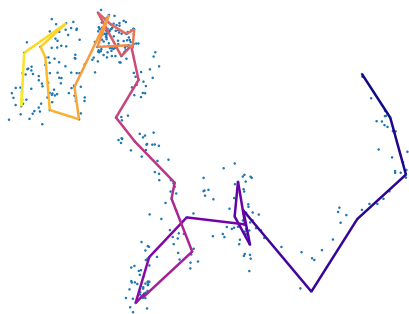


Figure 4-7: Demonstration of MCMC search with the Metropolis-Hastings algorithm, plotted in the hypothesis space. Scatter points are sampled hypotheses. The line indicates progression in the chain: purple refers to the beginning while yellow refers to the end. It is probable that after many iterations, these samples approximate the true posterior.

best, nor a way to construct better hypotheses — stochastic changes in hypotheses aids to ensure that the learner is not stuck in a local minimum and allows hypotheses to be sampled in a general way. However, MCMC performs a naïve search that is not deliberate as we see in learners. Nonetheless, stochastic search with a sufficiently powerful hierarchical generative model “provides a better account of children’s theory acquisition than alternative computational paradigms for modeling development” [Ullman, 2015].

There is typically a closed-form procedure for the prior and likelihood, which together form the *inductive bias* constituting the assumptions of a particular model, while the posterior is unknown and desired. MH provides samples from this posterior given a means to compute the prior and likelihood, where samples are determined by probabilistically accepting or rejecting proposals. In practice, particularly for the Metropolis-Hastings algorithm, a conditional sampling procedure $P(\theta'|\theta)$ is provided which generates proposals from a distribution that can be an improvement from the prior. The *rational rules* model provides a standard prior and likelihood for compositional grammar-based hypotheses: it favors simpler hypotheses while preventing noisy data from preventing effective generalization [Goodman et al., 2008].

A popular application of stochastic search via MCMC for inducing program is by establishing a probabilistic context-free grammar G (PCFG; see non-probabilistic CFGs in [Section 3.3 Context-free grammars](#)) over programs as hypotheses and inducing a production

E which explains the data D :

$$\mathbb{P}(E|D, G) \propto \mathbb{P}(D|E, G) \mathbb{P}(E|G)$$

A PCFG associates each production rule of a CFG with a probability, or *production probability* $(r, p_r) \in G$, thereby yielding a *derivation probability* for any production from the grammar (as the product of the production probability for every production rule that was used in constructing the production):

$$p_{\text{deriv}}(E|G) = \prod_{r \in \text{deriv}(E)} p_r$$

The derivation probability on its own acts as a prior $\mathbb{P}(E|G) = p_{\text{deriv}}(E|G)$ that is heavily biased on the grammar’s production probabilities. In practice, the production probabilities are induced using the inside-outside algorithm [Lafferty, 2000] or a different prior is used such as from the rational rules model which does not make strong assumptions on production probabilities [Goodman et al., 2008]. Under this popular approach, the CFG is designed to have an *interpreter*¹⁶ which evaluates a production on a datum $d \in D$ to yield a truth value: $E(d) \mapsto \{\text{TRUE}, \text{FALSE}\}$. It is trivial to construct an *all-or-nothing* likelihood measure:

$$\mathbb{P}(E|D, G) = \prod_{d \in D} \mathbb{1}_{\text{TRUE}}(E(d))$$

To account for noisy data where such a strict measure fails, the rational rules model provides exponential decay in the number of objects that yield FALSE. This PCFG- and interpreter-based approach has been used to induce Boolean expressions [Goodman et al., 2008], rich logical theories [Kemp et al., 2008a; Kemp, 2012], hierarchical concept theories as in physics and psychology [Ullman, 2015], and number concepts [Piantadosi et al., 2012]. These approaches use grammars that resemble propositional or predicate logic and are computed as such by the interpreter.

Other work has applied MCMC methods to combinatory logic (c.f. Section 3.4 Combinatory logic), assembly instructions, and first-order term rewriting (c.f. Section 3.6 Term rewriting) [Liang et al., 2010; Schkufza et al., 2013; Rule et al., 2018]. One of these approaches aims not just to induce target programs, but also to induce reusable modules that

¹⁶ As we discussed in Section 3.3 Context-free grammars.

are useful [Liang et al., 2010].

4.4.8 Genetic programming

Genetic programming (GP) uses the Darwinian principle of natural selection to evolve a population of computer programs using genetic operators [Koza, 1994]. It therefore has two primary components: a *fitness function* which is the determiner for natural selection, and *genetic operators* which produce offspring and drive evolution. The fitness function is defined according to a pre-determined task specification, and the genetic operators are defined based on the program representation. An outline of genetic programming is shown in Figure 4-8.

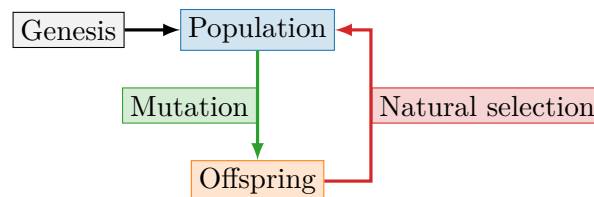


Figure 4-8: Genetic programming. Genesis is the creation of an initial population. Offspring are produced by performing genetic operators on the population. Through natural selection, the new population is determined.

The genetic operators on a per-individual basis (i.e. asexual reproduction) often include subtree mutation, point mutation, and hoist mutation¹⁷, with multi-individual operators derived from *crossover* (i.e. sexual reproduction) usually by swapping subtrees [Poli et al., 2008]. Hierarchical modular structure can be passed down across generations using automatic function definition and other “architecture-altering” operations that affect program structure to allow the emergence of hierarchy through abstraction [Koza, 1994; Olsson, 1994]. Constraints may be imposed, such as using types, to ensure that reproduction yields semantically valid programs. Or, rather than manipulating programs, genetic operators could manipulate *program development* where an individual’s efficacy (tested during natural selection with the fitness function) is based on *growing* a program from the genetic material: a notable example of this is in *neuro-evolution* where neural networks are evolved through network-generating procedures [Gruau, 1994; Stanley and Miikkulainen, 2002]. Another approach replaces genetic operators through the evolutionary process using “autoconstructive evolution” in which code-manipulating instructions make programs generate their own

¹⁷ Hoist mutation creates an offspring directly from a subtree of its parent.

offspring [Spector et al., 2005].

Fitness functions are vital to GP because without a good method of natural selection, viable offspring will not flourish. This is, however, a double-edged sword: fitness functions are often hard to define, may limit the variety of offspring that emerge after many generations, and can even prevent the objective from being achieved altogether [Lehman and Stanley, 2011]. There are two research directions in GP to counter these limitations: *co-evolution* in which different populations interact with the same environment and conditionally evolve on each others’ environmental effects, and *novelty* search which favors behaviors that other individuals in the population do not exhibit.

4.4.9 Neural network optimization

Neural network (c.f. Section 3.2 Neural networks) parameters are typically learned by performing gradient descent given a loss function for the network outputs and a regularizer for the network parameters. Each neuron is a differentiable function, and the way neurons are combined in neural networks make the entire network differentiable by propagating cost information back through the network to compute the gradient [Rumelhart and McClelland, 1986]. The resulting system allows network parameters to be learned given input/output pairs — the *training set* — and a loss function, by computing $\nabla_{\theta} J(\theta)$ where $J(\theta)$ is the loss computed using the training set¹⁸ [Goodfellow et al., 2016]. This optimization problem can be approached with *stochastic gradient descent*, illustrated in Figure 4-9. A regularizer term in the loss function is intended to prevent *overfitting*, so that the networks doesn’t effectively “memorize” the training set and failing to generalize as demonstrated in Figure 4-10.

For learning programs, there are two high-level approaches that neural network researchers have taken which are not distinguished in other architectures: *program synthesis* and *program induction*. Program synthesis is where the neural network generates a symbolic program, whereas program induction has the neural network learn a latent representation and acts as an evaluator for the latent program.

Neural program induction has been used to emulate a simple CPU, including stack-based memory or random access memory [Graves et al., 2014; Joulin and Mikolov, 2015; Kurach et al., 2015; Graves et al., 2016]. One neural program induction approach, called

¹⁸ For example, $J(\theta) = \frac{1}{|D|} \sum_{(x,y) \in D} (y - f_{\theta}(x))^2$ for network f_{θ} is based only on square loss with training set D . If f_{θ} has enough parameters, this cost function may easily result in overfitting.

the *neural programmer-interpreter*, learns how to represent and compute algorithms given program traces [Reed and De Freitas, 2015]. In neural program synthesis, Deepcoder uses a neural network to propose a distribution on which symbolic search techniques are performed [Balog et al., 2017], while the approaches of both NSPS and Robustfill provide an end-to-end neural network that directly produces programs from examples [Parisotto et al., 2016; Devlin et al., 2017b]. These program synthesis approaches require massive annotated datasets for training (e.g. hundreds of millions in Devlin et al. [2017b]), so generalization happens through numerous examples rather than few examples. However, we will demonstrate in Section 4.6 Language bootstrapping how we used neural networks to provide amortized inference in a Bayesian program learning (c.f. Section 4.5 Bayesian program learning) system despite having no annotated examples and sparse data.

Some neuro-symbolic¹⁹ approaches are able to learn from sparse data using reinforcement learning: for example, Chen et al. [2018] train a neural network to program a non-differentiable parser from hundreds of input/output pairs.

¹⁹ Neuro-symbolic means that neural networks may consume or produce symbolic structure, usually using recurrent networks. All neural program *synthesizers* are therefore neuro-symbolic because they produce symbolic programs.

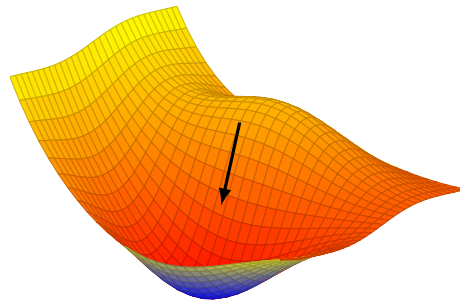


Figure 4-9: Learning landscape for a differentiable function, where lateral axes correspond to the parameter space (i.e. values of θ) and the height corresponds to the cost $J(\theta)$. The arrow indicates a step of stochastic gradient descent towards a lower-cost point in the space by following $-\nabla_{\theta}J(\theta)$.

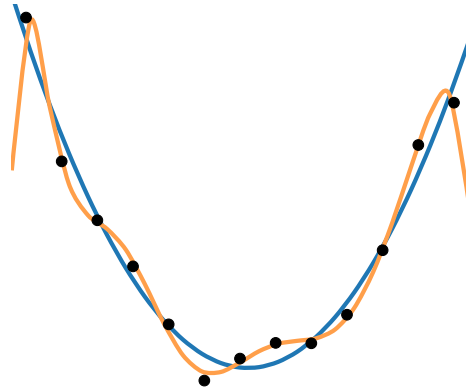


Figure 4-10: Overfitting: the orange curve overfits the data, while the blue curve is regularized.

4.5 Bayesian program learning

Bayesian program learning (BPL) is a generative model that samples programs, where programs are generative models that sample observations (or examples). That is to say, BPL is a *hierarchical Bayesian model* for programs and program behaviors. This consolidates two otherwise-independent accounts of inductive generalization: one which leverages domain-general statistical inference (e.g. pattern recognition, correlation), and another which leverages sophisticated domain-specific representations [Tenenbaum et al., 2006, 2011]. Inference in a hierarchical Bayesian model provides an approach for learning and reasoning from sparse data and inferring structured knowledge which may be domain-agnostic. Rather than describe *how* inference happens, hierarchical Bayesian models described *what* happens — i.e. it is a computation-level theory and not an algorithmic-level theory [Marr, 1982]. BPL provides an account of how people learn *intuitive theories*, abstract frameworks that guide learning within particular domains [Carey, 1985, 2009; Gopnik and Meltzoff, 1997; Ullman et al., 2012; Goodman et al., 2015]. The hierarchical structure in BPL yields rapid acquisition of abstract knowledge, which develops the inductive biases so lower-level knowledge can be learned quickly, effectively, and with fewer data: this is called the “blessing of abstraction,” manifested by *learning-to-learn* through the hierarchy [Tenenbaum et al., 2011; Goodman et al., 2011].

Bayesian accounts of learning can incorporate general biases like rationality and concept reuse [Johnson et al., 2007; Goodman et al., 2008; Liang et al., 2010; Dechter et al., 2013; O’Donnell, 2015]. Models based on BPL have shown success in cognitive studies of inductive

reasoning [Kemp and Tenenbaum, 2009], causal learning [Griffiths and Tenenbaum, 2009; Goodman et al., 2011], intuitive theories [Ullman et al., 2012], word learning [Frank et al., 2009], numerical concept learning [Piantadosi et al., 2012], and handwritten characters [Lake et al., 2015]. These models are all able to learn and generalize from few examples. Additionally, while search itself is often done stochastically by MCMC (c.f. Section 4.4.7 Markov chain Monte Carlo) because it follows easily from the Bayesian framing, other approaches may be applied to sampling for BPL such as amortized inference with neural networks (c.f. Section 4.6 Language bootstrapping) and constraint satisfaction [Ellis et al., 2016].

Bayesian program learning as a model for cognition acts as a *probabilistic language of thought* (c.f. LOT in Chapter 2 Computational Framing of Representation and Learning). It unites statistical and highly-structured symbolic frameworks, each of which saw developments for cognitive modelling and cognitive architecture despite heated debate between the two [Fodor and Pylyshyn, 1988; McClelland and Patterson, 2002; Pinker and Ullman, 2002]. Another historical debate in cognitive science has been between *empiricism* and *nativism*: is knowledge derived generally from experience, or is it constructed through hard-wired special-purpose faculty? BPL provides a computational perspective where both nature and nurture are necessary and explicitly outlined to the system. Using BPL to model a cognitive process therefore necessitates a decision of what to build in: “*what are the priors?*” as a Bayesian scientist often asks. An appeal to developmental science, which provides empirical evidence for *core knowledge* to answer these precise concerns, should be made when designing experiments that intend to model child behavior [Spelke, 1998; Spelke and Kinzler, 2007; Carey, 2009]. Furthermore, adopting Turing-complete representations (c.f. Chapter 3 Representations) permits the expression of novel complex concepts and, most critically, BPL provides an account for *learning* in such representations and to do so in an unbounded manner, thereby making BPL the only system which can in principle learn arbitrarily complex representations such as those adults may possess [Piantadosi and Jacobs, 2016]. Finally, BPL is more than a system for modeling abstract concepts: by expressing knowledge at many levels of abstraction, it can be integrated across amodal and multi-sensory representations to yield cross-modal transfer of knowledge [Yildirim and Jacobs, 2015].

4.6 Language bootstrapping

4.6.1 Introduction

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. In this section, we contribute a program induction algorithm called DREAMCODER that learns a DSL and trains a neural network to efficiently search for programs in the learned DSL [Ellis et al., 2018]. We use our model to solve symbolic regression problems, edit strings, and synthesize functions on lists, in each case showing the model learns a domain-specific vocabulary for expressing solutions to problems in the domain.

Automatically inducing programs from examples is a long-standing goal of artificial intelligence. Recent work has successfully used symbolic search techniques (e.g., Metagol: [Muggleton et al., 2015], FlashFill: [Gulwani, 2011]), neural networks trained from a corpus of examples (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., 2017]) to synthesize programs for task domains such as string transformations, list processing, and robot navigation and planning (these approaches are discussed in more detail above, in [Section 4.4 Learning architectures](#)). However, all these approaches – symbolic, neural and neural-symbolic – rely upon a hand-engineered DSL. The DSL is an inventory of restricted programming primitives, encoding domain-specific knowledge about the space of programs. In practice we often have only a few input/output examples for each program to be induced, and thus success often hinges on having a good DSL that provides a crucial inductive bias for what would otherwise be an unconstrained search through the space of all computable functions. Here we ask, to what extent can we dispense with such highly hand-engineered domain-specific languages?

We propose *learning* the DSL. We consider the setting where we have a collection of related programming tasks, each specified by a set of input/output examples. We do not assume that the tasks are annotated with ground-truth programs. We typically will not try to learn a DSL completely from scratch, but rather to start from a weaker or more general set of primitives and construct a richer, more powerful, and better-tuned DSL.

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 distills commonalities across programs that solve tasks, helping the agent solve related program induction problems. The neural network ensures that searching for programs remains tractable even as the DSL (and hence the search space for programs) expands.

Viewed as a probabilistic inference problem per Bayesian program learning (c.f. [Section 4.5 Bayesian program learning](#)), our goal is to jointly infer a single DSL (written \mathcal{D}) and many programs (written p), one for each set of inputs/outputs (*tasks*, written x , which comprise the task set X). A DSL \mathcal{D} is a collection of domain-specific subroutines. We equip \mathcal{D} with a real-valued weight vector θ , and together (\mathcal{D}, θ) define a distribution over programs. [Figure 4-11a](#) diagrams this inference problem as a hierarchical Bayesian model. Inference in this model is difficult because the programs are unobserved, and so we must solve a hard search problem to recover them. To make search tractable we learn a bottom-up *recognition model* (written $q(\cdot)$) illustrated in [Figure 4-11b](#)). The recognition model $q(\cdot)$ is a neural network that regresses from input/output pairs to a distribution over programs likely to explain the input/outputs. We can also view $q(\cdot)$ as implementing an amortized inference scheme [[Le et al., 2017](#)]. The neural recognition model and the generative model embodied in the DSL jointly train each other, as they iteratively learn to solve more programming tasks.

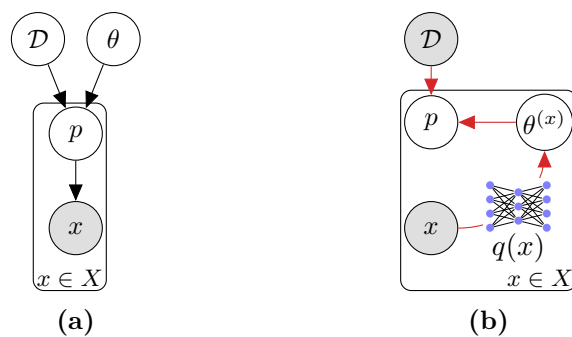


Figure 4-11: (a) DSL \mathcal{D} generates programs p by sampling DSL primitives according to probabilities θ . From each program we observe a task x (program inputs/outputs). (b) Neural network $q(\cdot)$, the *recognition model*, regresses from x to the parameters of a distribution over programs, $\theta(x)$. Black/red arrows correspond to the generative/recognition models.

We apply DREAMCODER to three domains: symbolic regression; FlashFill-style [[Gul-](#)

wani, 2011] string processing problems; and Lisp-style functions on lists. For each of these we deliberately provide an impoverished set of programming primitives, and show that our algorithm discovers its own domain-specific vocabulary for expressing solutions in the domain, highlighted in Figure 4-12.



	List Functions	Text Editing	Symbolic Regression
Programs & Tasks	<pre>[7 2 3]→[7 3] [1 2 3 4]→[3 4] [4 3 2 1]→[4 3] f(ℓ) = (f1 ℓ (λ (x) (> x 2))) [7 3]→False [3]→False [9 0 0]→True [0]→True [0 7 3]→True [2 7 8 1]→8 f(ℓ) = (f3 ℓ 0) [3 19 14]→19 f(ℓ) = (f2 ℓ)</pre>	<pre>+106 769-438→106.769.438 +83 973-831→83.973.831 f(s) = (f0 "." "-" (f0 "." "-" (cdr s)))</pre> <p>Lara Gregori→LG Temple Anna H→TAH f(s) = (f2 s)</p>	 <p>$f(x) = (f_1 x)$ $f(x) = (f_6 x)$</p>  <p>$f(x) = (f_4 x)$ $f(x) = (f_3 x)$</p>
	DSL	<pre>f0(ℓ,r) = (foldr r ℓ cons) (f0: Append lists r and ℓ) f1(ℓ,p) = (foldr ℓ nil (λ (x a) (if (p x) (cons x a) a))) (f1: Higher-order filter function) f2(ℓ) = (foldr ℓ 0 (λ (x a) (if (> a x) a x))) (f2: Maximum element in list ℓ) f3(ℓ,k) = (foldr ℓ (is-nil ℓ) (λ (x a) (if a a (= k x)))) (f2: Whether ℓ contains k)</pre>	<pre>f0(s,a,b) = (map (λ (x) (if (= x a) b x)) s) (f0: Performs character substitution) f1(s,c) = (foldr s s (λ (x a) (cdr (if (= c x) s a)))) (f1: Drop characters from s until c reached) f2(s) = (unfold s is-nil car (λ (z) (f1 z " "))) (f2: Abbreviates a sequence of words) f3(a,b) = (foldr a b cons) (f3: Concatenate strings a and b)</pre>

Figure 4-12: 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)

4.6.2 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 [\[Menon et al., 2013; Balog et al., 2017; Devlin et al., 2017b,a\]](#). 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 learn new, reusable subroutines that are shared in a multitask 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], which permits learning routines that are more complex than subexpressions of programs; 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 Section 4.5 Bayesian program learning). Previous BPL systems have largely assumed a fixed DSL (but see Liang et al. [2010]; Dechter et al. [2013]), and our contribution here is a general way of doing BPL with less hand-engineering of the DSL.

4.6.3 DREAMCODER

Our goal is to induce a DSL while finding programs solving each of the tasks. We take inspiration from the Wake/Sleep algorithm [Hinton et al., 1995], as well as the exploration-compression algorithm (EC) for bootstrap learning [Dechter et al., 2013]. Wake/Sleep alternates between training a generative model on samples from a recognition model (our q), and training a recognition model on samples from the generative model (our \mathcal{D}, θ). EC alternates between exploring the space of solutions to a set of tasks, and compressing those solutions to suggest new search primitives for the next exploration stage. We combine these ideas into an inference strategy that iterates through three steps: a **Wake** cycle uses the current DSL and recognition model to search for programs that solve the tasks. The **Sleep-G** and **Sleep-R** cycles update the DSL and the recognition model, respectively. Crucially, these steps bootstrap off each other, diagrammed in Figure 4-13.

Wake: Searching for programs. Our program search is informed by both the DSL and the recognition model. When these improve, we find more programs solving the tasks.

Sleep-G: Improving the DSL. We induce the DSL from the programs found in the wake

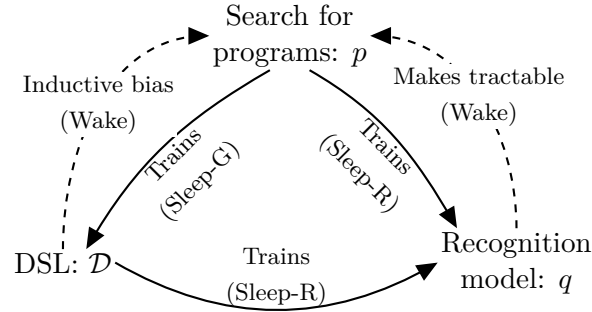


Figure 4-13: DREAMCODER solves for programs, the DSL, and a recognition model. Each of these steps bootstrap off of the others in a Helmholtz-machine inspired wake/sleep inference algorithm.

phase, aiming to maximally compress (or, raise the prior probability of) these programs. As we solve more tasks, we hone in on richer DSLs that more closely match the domain.²⁰

Sleep-R: Learning a neural recognition model. We update the recognition model by training on two data sources: samples from the DSL (as in the Helmholtz Machine’s “sleep” phase), and programs found by the search procedure during waking. As the DSL improves and as search finds more programs, the recognition model gets both more data to train on, and better data.²¹

[Section 4.6.3 Probabilistic framing](#) frames this 3-step procedure as a means of maximizing a lower bound on the posterior probability of the DSL given the tasks. [Section 4.6.3 Wake phase: searching for programs](#) explains how we search for programs that solve the tasks; [Section 4.6.3 Sleep-R: Learning a recognition model](#) explains how we train a neural network to search for programs; and [Section 4.6.3 Sleep-G: Learning a generative model \(a DSL\)](#) explains how we induce a DSL from programs.

Probabilistic framing

DREAMCODER takes as input a set of *tasks*, written X , each of which is a program induction problem. It has at its disposal a *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 \mathcal{D} .

We frame this problem as maximum a posteriori (MAP) inference in the generative

²⁰ This is loosely biologically inspired by the formation of abstractions during sleep memory consolidation [Dudai et al. \[2015\]](#).

²¹ These two sources are also loosely biologically inspired by the distinct episodic replay and hallucination components of dream sleep [Fosse et al. \[2003\]](#).

model diagrammed by [Figure 4-11a](#). Writing J for the joint probability of (\mathcal{D}, θ) and X , we want the \mathcal{D}^* and θ^* solving:

$$\begin{aligned}
 J(\mathcal{D}, \theta) &\triangleq \mathbb{P}[\mathcal{D}, \theta] \prod_{x \in X} \sum_p \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta] \\
 \mathcal{D}^* &= \arg \max_{\mathcal{D}} \int J(\mathcal{D}, \theta) d\theta \\
 \theta^* &= \arg \max_{\theta} J(\mathcal{D}^*, \theta)
 \end{aligned}
 \tag{4.1}$$

The above equations summarize the problem from the point of view of an ideal Bayesian learner. However, [Equation 4.1](#) is intractable because calculating J involves taking a sum over every possible program. We therefore define the *frontier* of a task x , written \mathcal{F}_x , to be a finite set of programs where $\mathbb{P}[x|p] > 0$ for all $p \in \mathcal{F}_x$ and establish an intuitive lower bound \mathcal{L} :

$$J \geq \mathcal{L} \triangleq \mathbb{P}[\mathcal{D}, \theta] \prod_{x \in X} \sum_{p \in \mathcal{F}_x} \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta]$$

DREAMCODER does approximate MAP inference in the generative model of [Figure 4-11a](#) by maximizing this lower bound on the joint probability, alternating maximization w.r.t. the frontiers (Wake) and the DSL (Sleep-G):

Program search: maximizing \mathcal{L} w.r.t. the frontiers. Here the DSL and distribution (\mathcal{D}, θ) are fixed and we want to find new programs to add to the frontiers so that \mathcal{L} increases the most, which are programs where $\mathbb{P}[x, p|\mathcal{D}, \theta]$ is large.

DSL induction: maximizing $\int \mathcal{L} d\theta$ w.r.t. the DSL. Here the frontier $\{\mathcal{F}_x\}_{x \in X}$ is held fixed, so we can evaluate \mathcal{L} . Now the problem is that of searching the discrete space of DSLs and finding one maximizing $\int \mathcal{L} d\theta$. Once we have a DSL \mathcal{D} we can update θ to $\arg \max_{\theta} \mathcal{L}(\mathcal{D}, \theta, \{\mathcal{F}_x\})$.

Searching for programs is hard because of the large combinatorial search space. We ease this difficulty by training a neural recognition model during the Sleep-R phase:

Recognition model: making program search *tractable*. Here we train a neural network, q , to predict a distribution over programs conditioned on a task. The objective of q is to assign high probability to programs p where $\mathbb{P}[x, p|\mathcal{D}, \theta]$ is large, because including those programs in the frontiers will most increase \mathcal{L} . Rather than directly predicting a distribution over p conditioned on x , the recognition model predicts a distribution $\theta^{(x)}$ over

components of the DSL. This taps into the intuition that programming is primarily a top-down activity: as human programmers, we often first decide what kind of programming constructs we might need to use, and then we figure out how to assemble them in to the desired program.

Wake phase: searching for programs

Now our goal is to search for programs that solve the tasks. In this work we use the simple search strategy of enumerating programs from the DSL in decreasing order of their probability, and then checking if an enumerated program p assigns positive probability to a task ($\mathbb{P}[x|p] > 0$); if so, we incorporate p into the frontier \mathcal{F}_x .

To make this concrete we need to define what programs actually are, what form $\mathbb{P}[p|\mathcal{D},\theta]$ takes, and how $\mathbb{P}[x|p]$ is evaluated. We represent programs as λ -calculus expressions in a Hindler-Milner polymorphic type system as described in [Section 3.5.2 Polymorphic typed lambda-calculus](#), and we adopt the notion described there. We say a type α *unifies* with τ if every expression $p : \alpha$ also satisfies $p : \tau$. Furthermore, the act of *unifying* a type α with τ is to introduce constraints on the type variables of α to ensure that α unifies with τ . With the representation of λ -calculus, we associate each task x with a set of input/output examples $(i,o) \in x$ and define the “all-or-nothing” likelihood model asserting that the program satisfies for each example:

$$\mathbb{P}[x|p] \triangleq \prod_{(i,o) \in x} \mathbb{1}(p(i) = o)$$

A DSL \mathcal{D} is a set of typed λ -calculus expressions. A weight vector θ for a DSL \mathcal{D} is a vector of $|\mathcal{D}| + 1$ probabilities: one for each DSL primitive $e \in \mathcal{D}$, written θ_e , and a weight controlling the probability of a variable occurring in a program, θ_{var} . [Algorithm 1](#) is a procedure for drawing samples from the generative model (\mathcal{D},θ) . In practice, we enumerate programs rather than sampling them. Enumeration proceeds by a depth-first search over the random choices made by [Algorithm 1](#); we wrap the depth-first search in iterative deepening to build λ -calculus expressions in order of their probability.

Why enumerate, when the program synthesis community has invented many sophisticated algorithms that search for programs²²? We have two reasons: (1) A key point of

²² Many of these sophisticated algorithms were discussed earlier in [Section 4.4 Learning architectures](#). For

Algorithm 1 Generative model over programs

function sample($\mathcal{D}, \theta, \mathcal{E} = \emptyset, \tau$):
Input: DSL \mathcal{D} , weight vector θ , environment \mathcal{E} , type τ
Output: a program whose type unifies with τ
if $\tau = \alpha \rightarrow \beta$ **then**
 var \leftarrow an unused variable name
 body \sim sample ($\mathcal{D}, \theta, \{\text{var} : \alpha\} \cup \mathcal{E}, \beta$)
 return (lambda (var) body)
end if
primitives $\leftarrow \{p \mid p : \tau' \in \mathcal{D} \cup \mathcal{E}$
 if τ can unify with yield(τ') $\}$
Draw $e \sim$ primitives, w.p. $\propto \theta_e$ if $e \in \mathcal{D}$
 w.p. $\propto \frac{\theta_{var}}{|\text{variables}|}$ if $e \in \mathcal{E}$
Unify τ with yield(τ').
 $\{\alpha_k\}_{k=1}^K \leftarrow$ args(τ')
for $k = 1$ **to** K **do**
 $a_k \sim$ sample($\mathcal{D}, \theta, \mathcal{E}, \alpha_k$)
end for
return ($e a_1 a_2 \cdots a_K$)
where:
 yield(τ) = $\begin{cases} \text{yield}(\beta) & \text{if } \tau = \alpha \rightarrow \beta \\ \tau & \text{otherwise.} \end{cases}$
 args(τ) = $\begin{cases} [\alpha] + \text{args}(\beta) & \text{if } \tau = \alpha \rightarrow \beta \\ [] & \text{otherwise.} \end{cases}$

our work is that learning the DSL along with a neural recognition model can make program induction tractable, even if the search algorithm is very simple. (2) Enumeration is a general approach that can be applied to *any* program induction problem — many of the state-of-the-art approaches for search require special conditions or hand-engineered features in the space of programs that enumeration does not require.

A drawback of using an enumerative search algorithm is that we have no efficient means of solving for arbitrary constants that might occur in the program. In [Section 4.6.4 Symbolic regression](#), we will show how to find programs with real-valued constants by automatically differentiating through the program and setting the constants using gradient descent.

the paradigm we take, the works of Solar-Lezama [2008]; Schkufza et al. [2013]; Feser et al. [2015]; Osera and Zdancewic [2015]; Polozov and Gulwani [2015] are particularly relevant.

Sleep-R: Learning a recognition model

The purpose of the recognition model is to accelerate the search for programs. It does this by learning to predict programs which are probable under (\mathcal{D}, θ) while also assigning high likelihood for a task according to $\mathbb{P}[x|p]$.

The recognition model q is a neural network that predicts, for each task $x \in X$, a weight vector $q(x) = \theta^{(x)} \in [0, 1]^{|\mathcal{D}|+1}$. After normalizing to satisfy total probability, this with the DSL together define a distribution over programs: $\mathbb{P}[p|\mathcal{D}, \theta = q(x)]$. We abbreviate this distribution as $q(p|x)$. The crucial aspect of this framing is that the neural network leverages the structure of the learned DSL, so it is *not* responsible for generating programs wholesale. We share this aspect with DeepCoder [Balog et al., 2017].

We want a recognition model that closely approximates the true posteriors over programs. We formulate this as minimizing the expected KL-divergence,

$$\min_{x \sim X} \mathbb{E} [\text{KL}(\mathbb{P}[p|x, \mathcal{D}, \theta] \parallel q(p|x))] \iff \max_{x \sim X} \mathbb{E} \left[\sum_p \mathbb{P}[p|x, \mathcal{D}, \theta] \log q(p|x) \right]$$

One could take this expectation over the observed empirical distribution of tasks, like how an autoencoder is trained [Hinton and Salakhutdinov, 2006], or one could take this expectation over samples from the generative model, like how a Helmholtz machine is trained [Dayan et al., 1995]. We found it useful to maximize both an autoencoder-style objective \mathcal{L}_{AE} and a Helmholtz-style objective \mathcal{L}_{HM} , giving the objective for a recognition model, \mathcal{L}_{RM} :

$$\mathcal{L}_{\text{RM}} = \mathcal{L}_{\text{AE}} + \mathcal{L}_{\text{HM}} \tag{4.2}$$

$$\mathcal{L}_{\text{HM}} = \mathbb{E}_{p \sim (\mathcal{D}, \theta)} [\log q(p|x)], \text{ } p \text{ evaluates to } x$$

$$\mathcal{L}_{\text{AE}} = \mathbb{E}_{x \sim X} \left[\sum_{p \in \mathcal{F}_x} \frac{\mathbb{P}[x, p|\mathcal{D}, \theta]}{\sum_{p' \in \mathcal{F}_x} \mathbb{P}[x, p'|\mathcal{D}, \theta]} \log q(p|x) \right]$$

The \mathcal{L}_{HM} objective is essential for data efficiency: all of our experiments train DREAM-CODER on only a few hundred tasks, which is too few for a high-capacity neural network q . Once we bootstrap a (\mathcal{D}, θ) , we can draw unlimited samples from (\mathcal{D}, θ) and train q on those samples. Evaluating \mathcal{L}_{HM} involves sampling programs from the current DSL, 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 architecture of q depends upon the domain. It regresses from an observation x , as a set of input/output pairs $(i, o) \in x$, to a $|\mathcal{D}| + 1$ dimensional vector. Each input/output pair is processed by an identical encoder network; the outputs of the encoders are averaged and passed to an MLP with 1 hidden layer, 32 hidden units, and a ReLU activation, before finally being normalized for total probability:

$$q(x) = \text{Normalize} \left(\text{MLP} \left(\text{Average} \left(\{\text{Encode}(i, o)\}_{(i, o) \in x} \right) \right) \right)$$

For the string editing and list function domains, the inputs and outputs are sequences. Our encoder for these domains is a bidirectional GRU (c.f. [Cho et al., 2014]) with 64 hidden units that reads each input/output pair where we concatenate the input and output with a special delimiter symbol between them. We MaxPool the final hidden unit activations in the GRU along both passes of the bidirectional GRU. For symbolic regression, the input/output pairs are densely sampled points along the curve of the function. We rendered these points to a graph, and pass the image of the graph to a convolutional network, which acts as the encoder.

Sleep-G: Learning a generative model (a DSL)

The purpose of the DSL is to offer a set of abstractions that allow an agent to easily express solutions to the tasks at hand. In the DREAMCODER algorithm we infer the DSL from a collection of frontiers. Intuitively, we want the algorithm to look at the frontiers and generalize beyond them, both so the DSL can better express the current solutions, and also so that the DSL might expose new abstractions which will later be used to discover more programs.

Recall from the [Probabilistic framing](#) that we want the DSL maximizing $\int \mathcal{L} d\theta$. We replace this with an AIC approximation, giving the following objective for DSL induction:

$$\log \mathbb{P}[\mathcal{D}] + \arg \max_{\theta} \sum_{x \in X} \log \sum_{p \in \mathcal{F}_x} \mathbb{P}[x|p] \mathbb{P}[p | \mathcal{D}, \theta] + \log \mathbb{P}[\theta | \mathcal{D}] - \|\theta\|_0 \quad (4.3)$$

We induce a DSL by searching locally through the space of DSLs, proposing small local changes to \mathcal{D} until [Equation 4.3](#) fails to increase. The search moves work by introducing

new λ -expressions into the DSL. We propose these new expressions by extracting fragments of programs already in the frontiers (Figure 4-14). An important point here is that we are *not* simply adding subexpressions of programs to \mathcal{D} , as done in the EC algorithm [Dechter et al., 2013] and other prior work [Lin et al., 2014]. Instead, we are extracting fragments that unify with programs in the frontiers. This idea of storing and reusing fragments of expressions comes from Fragment Grammars [O’Donnell, 2015] and Tree-Substitution Grammars [Cohn et al., 2010], and is closely related to the idea of antiunification Henderson [2013].

We define a prior distribution over DSLs which penalizes the sizes of the λ -calculus expressions in the DSL, and put a Dirichlet prior over the weight vector:

$$\mathbb{P}[\mathcal{D}] \propto \exp\left(-\lambda \sum_{p \in \mathcal{D}} \text{size}(p)\right) \quad (4.4)$$

$$\mathbb{P}[\theta | \mathcal{D}] = \text{Dir}(\theta | \alpha) \quad (4.5)$$

where $\text{size}(p)$ measures the size of the syntax tree of program p , λ is a hyperparameter that acts as a regularizer on the size of the DSL²³, and α is a concentration parameter controlling the smoothness of the prior over θ ²⁴. Algorithm 2 specifies the DSL induction algorithm.

To appropriately score each proposed \mathcal{D} we must reestimate the weight vector θ . Al-

²³ Implementation detail: we assigned $\lambda = 1$.

²⁴ Implementation detail: we assigned $\alpha = 10$.

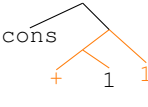


	Example programs in frontiers	Proposed λ -expression
program 	$(\lambda(\ell) (\text{map } (\lambda(x) (\text{index } x\ell))$ $(\text{range } (- (\text{length}\ell) 1))))$	$(\text{map } (\lambda(x) (\text{index } x\ell))$ $(\text{range}\alpha))$
program 	$(\lambda(\ell) (\text{map } (\lambda(x) (\text{index } x\ell))$ $(\text{range } (+ 1 1))))$	$(\lambda(s) (\text{map } (\lambda(x)$ $(\text{if } (= x \text{'.'}) \text{'-' } x))) s)$
fragment 	$(\lambda(s) (\text{map } (\lambda(x)$ $(\text{if } (= x \text{'-'}) \text{'.' } x))) s)$	$(\lambda(s) (\text{map } (\lambda(x)$ $(\text{if } (= x\alpha) \beta x))) s)$

Figure 4-14: **Left:** syntax trees of two programs sharing common structure, highlighted in orange, from which we extract a fragment and add it to the DSL (bottom). **Right:** actual programs, from which we extract fragments that (top) slice from the beginning of a list or (bottom) perform character substitutions.

Algorithm 2 DSL Induction Algorithm

Input: Set of frontiers $\{\mathcal{F}_x\}$
Hyperparameters: Pseudocounts α , regularization parameter λ
Output: DSL \mathcal{D} , weight vector θ
Define $L(\mathcal{D}, \theta) = \prod_x \sum_{p \in \mathcal{F}_x} \mathbb{P}[p | \mathcal{D}, \theta]$
Define $\theta^*(\mathcal{D}) = \arg \max_{\theta} \text{Dir}(\theta | \alpha) L(\mathcal{D}, \theta)$
Define $\text{score}(\mathcal{D}) = \log \mathbb{P}[\mathcal{D}] + L(\mathcal{D}, \theta^*) - \|\theta\|_0$
 $\mathcal{D} \leftarrow$ every primitive in $\{\mathcal{F}_x\}$
while true **do**
 $N \leftarrow \{\mathcal{D} \cup \{s\} | x \in X, p \in \mathcal{F}_x, s \text{ a fragment of } p\}$
 $\mathcal{D}' \leftarrow \arg \max_{\mathcal{D}' \in N} \text{score}(\mathcal{D}')$
 if $\text{score}(\mathcal{D}') < \text{score}(\mathcal{D})$ **return** $\mathcal{D}, \theta^*(\mathcal{D})$
 $\mathcal{D} \leftarrow \mathcal{D}'$
end while

though this problem may seem very similar to estimating the parameters of a probabilistic context free grammar (PCFG), for which we have effective approaches like the Inside/Outside algorithm [Lafferty, 2000], a DSL in DREAMCODER may be context-sensitive due to the presence of variables in the programs and also due to the polymorphic typing system. We derive a tractable MAP estimator for θ :

Estimating θ

We use an expectation-maximization (EM) algorithm to estimate the continuous parameters of the DSL, e.g. θ . Suppressing dependencies on \mathcal{D} , the EM updates are

$$\theta = \arg \max_{\theta} \log \mathbb{P}(\theta) + \sum_x \mathbb{E}_{Q_x} [\log \mathbb{P}[p | \theta]] \quad (4.6)$$

$$Q_x(p) \propto \mathbb{P}[x | p] \mathbb{P}[p | \theta] \quad (4.7)$$

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

We write $c(e, p)$ to mean the number of times that primitive e was used in program p ; $c(p) = \sum_{e \in \mathcal{D}} c(e, p)$ to mean the total number of primitives used in program p ; $R(p)$ to mean the sequence of types input to sample in Algorithm 1. Jensen’s inequality gives a

lower bound on the likelihood:

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

where we have defined

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

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

$$\sum_e (C(e) + \alpha) \log \theta_e - \beta \log \sum_{\tau} \frac{R(\tau)}{\beta} \sum_{\substack{e: \tau' \in \mathcal{D} \\ \text{unify}(\tau, \tau')}} \theta_e \quad (4.8)$$

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

$$\frac{C(e) + \alpha}{\theta_e} \propto \sum_{\tau'} \mathbb{1}[\text{unify}(\tau, \tau')] R(\tau') \quad (4.9)$$

$$\theta_e \propto \frac{C(e) + \alpha}{\sum_{\tau'} \mathbb{1}[\text{unify}(\tau, \tau')] R(\tau')} \quad (4.10)$$

The above is our estimator for θ_e . Despite the convoluted derivation, the above estimator

has an intuitive interpretation. The quantity $C(e)$ is the expected number of times that we used e . The quantity $\sum_{\tau'} \mathbb{1}[\text{unify}(\tau, \tau')] R(\tau')$ 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. This lower bound is tight whenever all of the types of the expressions in the DSL are not polymorphic, in which case our DSL is equivalent to a PCFG and this estimator is equivalent to the inside/outside algorithm. Polymorphism introduces context-sensitivity to the DSL, and exactly maximizing the likelihood with respect to θ becomes intractable, so for domains with polymorphic types we use this estimator.

Implementing DREAMCODER

[Algorithm 3](#) describes how we combine program search, recognition model training, and DSL induction. We note the following implementation details:

1. We perform a wake cycle before each of the sleep cycles.
2. On the first iteration, we do *not* train the recognition model on samples from the generative model because a generative model has not yet been learned – we instead train the network to only maximize \mathcal{L}_{AE} .
3. Because the frontiers can grow very large, we only keep around the top 10^4 programs p in each frontier \mathcal{F}_x with the highest likelihood $\mathbb{P}[x, p | \mathcal{D}, \theta]$.
4. During both DSL induction and neural net training, we calculate [Equation 4.3](#) and \mathcal{L}_{RM} by only summing over the top K programs in \mathcal{F}_x as measured by $\mathbb{P}[x, p | \mathcal{D}, \theta]$ – we found that $K = 2$ sufficed.
5. For added robustness, we enumerate programs from both the generative model and the recognition model.

Algorithm 3 The DREAMCODER Algorithm

Input: Initial DSL \mathcal{D} , set of tasks X , iterations I
Hyperparameters: Maximum frontier size F
Output: DSL \mathcal{D} , weight vector θ , recognition model $q(\cdot)$
Initialize $\theta \leftarrow$ uniform
for $i = 1$ **to** I **do**
 $\mathcal{F}_x^\theta \leftarrow \{p \mid p \in \text{enum}(\mathcal{D}, \theta, F) \text{ if } \mathbb{P}[x|p] > 0\}$ (Wake)
 $q \leftarrow$ train recognition model, maximizing \mathcal{L}_{RM} (Sleep-R)
 $\mathcal{F}_x^q \leftarrow \{p \mid p \in \text{enum}(\mathcal{D}, q(x), F) \text{ if } \mathbb{P}[x|p] > 0\}$ (Wake)
 $\mathcal{D}, \theta \leftarrow \text{induceDSL}(\{\mathcal{F}_x^\theta \cup \mathcal{F}_x^q\}_{x \in X})$ (Sleep-G)
end for
return \mathcal{D}, θ, q

4.6.4 Experiments with DREAMCODER

We apply DREAMCODER to two sequence-processing domains: [List functions](#) and [Text editing](#). For both of these domains we initially provide the system with a generic set of sequence processing primitives: `foldr`, `unfold`, `if`, `map`, `length`, `index`, `=`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`.

We also apply DREAMCODER to [Symbolic regression](#) problems based on visual input. For this domain, we initially provide the system with three primitives: `+`, `*`, `÷`, and `real`.

List functions

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.

We created 236 human-interpretable list manipulation tasks, each with 15 input/output

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

Figure 4-15: Some tasks in our list function domain. See the supplement for the complete data set.

examples, some of which are shown in [Figure 4-15](#). 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 [Figure 4-12](#), right of [Figure 4-14](#)). See the supplement for a complete list of DSL primitives discovered by DREAMCODER.

Text editing

Synthesizing programs that edit text is a classic problem in the programming languages and AI literatures [[Lau, 2001](#); [Menon et al., 2013](#)], 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 $(f_3 \text{string } s)$ — where f_3 is the learned appending primitive defined in [Figure 4-12](#) — and then define $\mathbb{P}[x|p]$ by approximately marginalizing out the string parameters via a simple dynamic program. When we analyze [Symbolic regression](#), 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 [Figure 4-12](#)) that let it solve all of the training tasks. But, how well does the learned DSL generalize to real text-editing scenarios? We tested, but did not train, on the 108 text editing problems from the SyGuS program synthesis competition

[Alur et al., 2016]. 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*.²⁵ Here we learned a single DSL that applied generically to all of the tasks, and perform comparably to the best prior work.

Symbolic regression

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 graph (Figure 4-16) — 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.

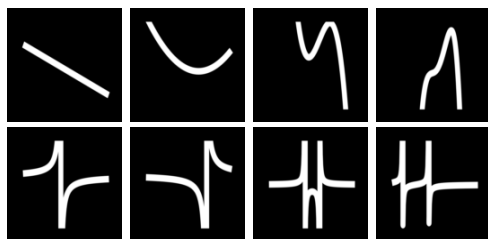


Figure 4-16: Recognition model input for symbolic regression. DSL learns subroutines for polynomials (top row) and rational functions (bottom row) while the recognition model jointly learns to look at a graph of the function (above) and predict which of those subroutines best explains the observation.

²⁵ SyGuS text editing problems also pre-specify the set of allowed string constants for each task. For these experiments, our system did not use this assistance.

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 (from each use of the `real` primitive), and represents quartic functions using the invented DSL primitive f_4 in the rightmost column of Figure 4-12 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.

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.

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

Figure 4-17: Percentage of held-out test tasks solved. Solve time: averaged over solved tasks. (*: This particular experiment was done *without* Helmhotlz-style sampling for training the recognition model.)

We compare against baselines in Figure 4-17 with the following ablations:

No NN lesions the recognition model.

SE lesions the recognition model and restricts the DSL learning algorithm to only add

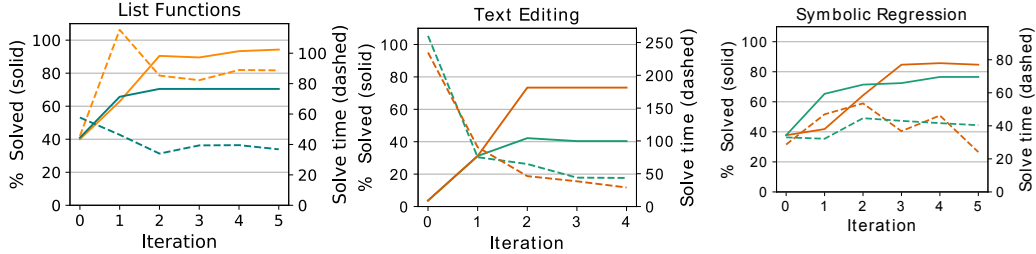


Figure 4-18: Learning curves for DREAMCODER both with (in orange) and without (in teal) the recognition model. Solid lines: percentage of held-out testing tasks solved; dashed lines: average solve time.

SubExpressions²⁶ 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]. NPS learns DSL induction, 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 DeepCoder [Balog et al., 2017] is trained.

PCFG, which lesions the recognition model *and* DSL induction, but instead learns the parameters θ of the DSL without learning any DSL structure.

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

For each domain, we are interested both in how many tasks the agent can solve and how quickly it can find those solutions. Figure 4-17 compares our model against these alternatives. We consistently improve on the baselines, and find that lesioning the recognition model slows down the convergence of the algorithm, taking more iterations to reach a given number of tasks solved as shown in Figure 4-18. This supports a view of the recognition model as a way of amortizing the cost of search.

4.6.5 DREAMCODER: conclusion and future work

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

²⁶ Our approach does not simply extract subexpressions, it extracts fragments which can introduce new variable bindings as illustrated in Figure 4-14.

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 learning more sophisticated generative models than the programming domains explored 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?

4.7 Domain-general learning

4.7.1 Introduction

Machine learning typically becomes less effective as a subject domain is broadened. Humans, on the other hand, are good at learning in arbitrarily broad domains, and exhibit the ability to specialize in many domains without facing critical consequences from working at such large scales. We hypothesize that this ability emerges, in part, from automatically structured knowledge, where local relations between knowledge artifacts correspond to contextual relevance. We introduce the construction and utilization of this knowledge structure as *contextual learning*, and consider the problem of implementing a contextual learner. In this section, we propose a solution to this problem with a framework, CONTEXTNET, for automatic learning at various scales of generality inspired by patterns found in social wealth, natural languages, protein interactions, and many other natural phenomena. This framework implements a knowledge abstraction, so many existing machine learning algorithms can be augmented to interface with it allowing knowledge to be automatically structured behind-the-scenes. We experimentally verified our hypothesis by applying our framework to a program induction algorithm tasked with string transformation problems, and by demonstrating the enhanced ability for the program inductor to learn and solve complex problems in a broad domain when augmented to utilize our framework.

Consider a mathematician attempting to prove a theorem. In order to come up with the proof, the mathematician needs to utilize and build on top of theorems that already exist. However, good mathematicians don't consider using every single theorem that they know — they instead focus their efforts on a handful of theorems they expect to be relevant. This *context* makes searching for a proof tractable. Despite knowing thousands of theorems, a mathematician might focus on only a dozen that seem relevant to the task at hand. Upon successfully proving the theorem, the mathematician will add it to her tool belt, utilizing it whenever a relevant circumstance arises. This example illustrates *contextual learning* — learning that is conditioned on a relevant subset of knowledge called the context. Learning mechanisms that are compositional — which both consume and construct knowledge, allowing for rapid acquisition and generalization of knowledge when confronted with new complex tasks [Lake et al., 2017] — may be adapted to become contextual. We accomplish this by imposing a hierarchical model over the learner, which learns a latent distribution

over domains so that, given a domain which is motivated by the environment, relevant knowledge artifacts to be readily available while setting others further apart.

Our approach is motivated by the observation that many compositional learners scale poorly to broad domains. We propose an answer to this challenge of enabling compositional learning mechanisms to become scalable contextual learners by designing a knowledge abstraction that provides an interface for learning mechanisms while automatically structuring knowledge behind-the-scenes. Our solution is general enough to act as a unified framework on which distinct mechanisms can share knowledge and learn in tandem, so we refer to it as a component of cognitive architecture.

To contextualize knowledge in an abstract manner, we develop a component of cognitive architecture called the CONTEXTNET framework. Knowledge is represented as distinct artifacts of information and their relations in the structure of a connected network. The network is constructed by preferential “least effort” attachment [Barabási and Albert, 1999; i Cancho and Solé, 2003], where a new knowledge artifact joins the network with relations to the contextual knowledge from which it was learned. This network is scale-free — it conforms to a mathematical pattern similar to that of Zipf’s Law and Pareto distributions [Zipf, 1949; Barabási et al., 2016], where the degrees of connectivity for nodes in the network follow a class of power law distributions — a design which inherently encodes a metric of utility in its structure and is prevalent in many natural systems. The resulting system enables automatic contextualization of any learned knowledge, provides effective compositionality for learning mechanisms which rely on knowledge artifacts, and supports multi-mechanism learning with a unified context.

We apply this system to a compositional program inductor tasked with learning string transformations. Our results show that our approach to automatically structuring knowledge allows us to achieve rapid learning of rich models without sacrificing ability to scale to broad domains.

4.7.2 Related work

In the discipline of artificial intelligence, much effort is placed on designing accurate learning mechanisms. These learning mechanisms are either designed for a particular domain, or designed to specialize given consistent domain-specific input. Special-purpose learning mechanisms include visual object recognition systems based on the human visual cortex

[Serre et al., 2007], handwritten character identification and generation [Lake et al., 2015], visual feature modification of images [Kulkarni et al., 2015], sound texture perception and synthesis [McDermott and Simoncelli, 2011], and countless others. General-purpose learning mechanisms include hierarchical Bayesian methods and every mechanism discussed in [Section 4.4 Learning architectures](#).

Each of these mechanisms are remarkable, but lack the generality that is apparent in human cognition. Even the general-purpose mechanisms, while capable of being applied to many domains, are only practical when restricted to a single domain, as a specialized instantiation of the mechanism. The need for specialization makes general-purpose mechanisms unappealing in practice, because a special-purpose mechanism could outperform it. This stems, in part, from a problem of *knowledge*, for which learning mechanisms have some internal representation (e.g. enumeration internally represents primitives and a search procedure). Those representations are mechanism-specific, and are generally used without hierarchy. Making all knowledge flat in this manner yields systems that only perform well when confined to a single domain and flawed as a model of cognition due to the reduced capability of both handling very complex problems and specializing in multiple domains.

Theories for cognitive architecture have pursued solving these problems. ACT* [Anderson, 1983] is a system which utilizes memory according to a degree of activation, where activation spreads to favor information most related to the immediate context. ACT* relates items of memory with a matrix pairing the strength of connection between any two items, which restricts memory capacity with a flat representation of knowledge and is therefore lacking in the same vein as the mechanisms mentioned above. Soar [Newell, 1994] is another system of cognitive architecture which traverses memory by pattern-matching stored production rules. Soar has unstructured sets of memory and learns new production rules with implicit generalization about the context it informs. Soar does not have implicit relationships between items of knowledge like ACT* — it has limited explicit relationships about context as a hierarchical data structure of subgoals, problem spaces, states, and operators. Soar relies on generating new production rules efficiently to reduce the complexity of problem solving. Hierarchical models such as those discussed in [Section 4.5 Bayesian program learning](#) provide a key insight which we leverage towards solving these problems, but it must be noted that they are not typically used in this way: their application has found itself in domain-specific contexts, where there is a particular class of learning problems that

are modeled within a single inferred distribution. We extend this traditional approach by having the single inferred distribution arise from a sophisticated *mixture model* determined by context.

Preferential “least-effort” attachment and resulting power-law distributions have been reported heavily over the last century, from urn models and Yule processes describing the statistics of the evolution of simple mathematical systems, to Zipf’s Law describing patterns in social wealth and natural language, to the Matthew effect describing analogous patterns in psychosocial processes, to the emergence of scale-free networks and its presence in protein interactions, linguistics, citations, and many other systems [Yule et al., 1925; Zipf, 1949; Merton, 1968; Barabási and Albert, 1999; Jeong et al., 2001; i Cancho and Solé, 2001; Price, 1965; Clauset et al., 2009].

We model non-flat collections of knowledge artifacts using a scale-free network based on intuitive traits of cognition:

- *Growth*, the learning and discovery of new knowledge artifacts expanding a knowledge-base.
- *Least-effort attachment*, the construction of relations between a new knowledge artifact and its antecedents. These relations are not random, but instead associate according to relevance.

Presence of these two traits in systems have been shown effective in yielding scale-free networks. While the flat representations described earlier in this section yield an average graph distance of $\langle d \rangle = 1$ (corresponding to a fully-connected graph), scale-free networks yield an average distance of $\langle d \rangle \sim \lg \lg N$ for network size N . Even with massive network sizes it’s still easy to travel from one point of the network to another. The lack of “scale” of the network refers to the variety of generally-useful (high degree) and highly-specialized (low degree) items within the same network, a feature which inherently encodes a metric of utility in the network structure.

The goal of CONTEXTNET is to augment learning mechanisms around *contextual* knowledge in a manner that reduces the intractability of hard problems by implicitly relating knowledge in a non-flat structure, enabling compositionality at arbitrary scale. In terms of the Marr-Poggio levels of analysis [Marr, 1982], CONTEXTNET serves as a tool on the algorithmic level of abstraction, not as a description of hardware-level structure — analogous

to the World Wide Web operating at a higher level over the Internet.

4.7.3 CONTEXTNET

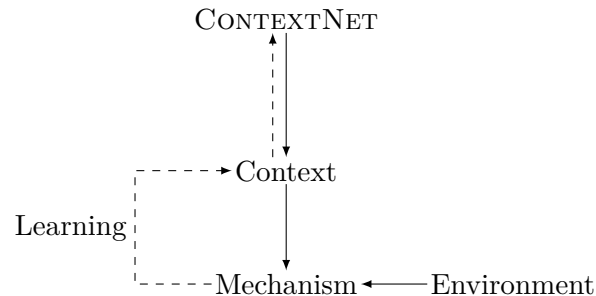


Figure 4-19: Schematic of the function of CONTEXTNET. The environment constitutes everything that the system observes from the “outside world.” The context, derived from the CONTEXTNET, defines all parameters for the mechanism. When the mechanism learns something new, it passes that information to the context which mutates the CONTEXTNET to incorporate that new knowledge.

The CONTEXTNET framework illustrated in Figure 4-19 implements a knowledge abstraction, on par with typical non-hierarchical sets or addressed memory. We refer to a small motivated subnet of the knowledge network as the *context*. We provide a simple computer interface for interacting with knowledge where all interactions either retrieve information related to the current context or adjust the context within the knowledge network, outlined in Figure 4-20.

GET() $\{\mathcal{I}\}$	Yields the set of items in the active context
EXPLORE() $\{\mathcal{I}\}$	Yields the set of items within one edge of the active context
ORIENT(\mathcal{I})	Shifts context to focus on the given item
ADD(\mathcal{I})	Adds an item to the knowledge network

Figure 4-20: Methods on an instance of the knowledge network.

A knowledge artifact \mathcal{I} is a structure containing a unique identifier, a symbolic tag denoting which mechanism can understand the item’s content, and the arbitrary content itself. Only the ADD(\mathcal{I}) and ORIENT(\mathcal{I}) methods have the potential to augment the active context. The ADD(\mathcal{I}) method automatically adds a new item of knowledge to the network, and adjusts the context to include the new item, illustrated in Figure 4-21.

The context itself is sized according to a pre-determined minimum size K_{\min} and the

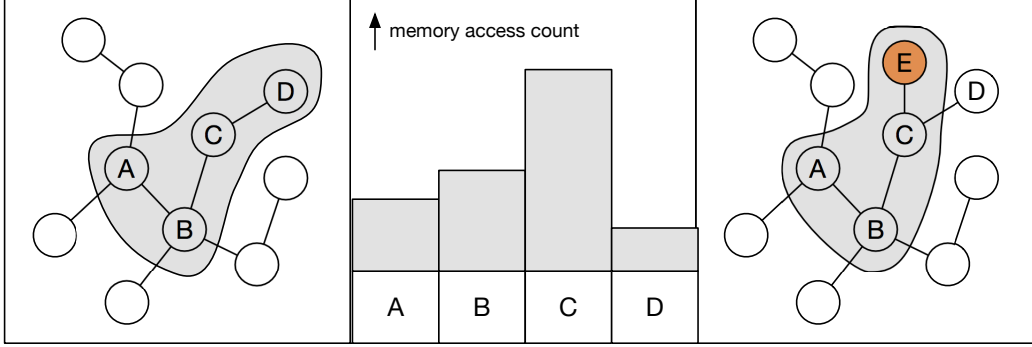


Figure 4-21: The first pane shows a knowledge network, and a shaded region representing the active context. The second pane shows the memory accesses within the context leading up to the addition of a new knowledge artifact. The third pane shows the new knowledge network, with the additional knowledge artifact nondeterministically attached according to the distribution of memory accesses.

maximum degree of a node in a scale-free network based on network size N , with degree exponent γ satisfying $2 < \gamma < 3$:

$$k_{\max} \sim N^{\frac{1}{\gamma-1}}$$

$$K = \max(K_{\min}, k_{\max})$$

We equate the context size K to k_{\max} wherever possible in order to support the scale-free property. When we generate a new context after calls to either $\text{ORIENT}(\mathcal{I})$ or $\text{ADD}(\mathcal{I})$, we use a growth procedure as follows: we first uniformly sample γ to probabilistically determine a context size K and initialize the context to the selected node. Then we add the selected node’s neighbors to the context in order preferring the higher-degree nodes, select the neighbor with the highest recent access count, and repeat until we reach size K . This process adds relevant nodes while preferring to generalize.

In order to enable the emergence of the scale-free property, the automatic attachment via $\text{ADD}(\mathcal{I})$ follows a least-effort procedure where the new node’s connections are determined using popularity-based probabilistic subset selection. The selection technique we implement is equivalent to a non-initial iteration of the Indian Buffet Process (IBP) with parameter $\alpha = 0$ [Griffiths and Ghahramani, 2011]. For every node $\mathcal{I}_j \in \{\mathcal{I}_i\}_{i=1}^k$ in the context, we have m_j , a count of accesses to \mathcal{I}_j since the last context-switch. Additionally, we define $M = \sum_{i=1}^k m_i$ as the total count of knowledge artifact accesses since the last context-switch. The new node attaches to node \mathcal{I}_j with probability proportional to its popularity, according

to $\text{BERNOULLI}\left(\frac{m_j}{M}\right)$. IBP doesn't guarantee non-empty set, so if no connections are made for the new node, a single connection is forced using an iteration of the Chinese Restaurant Process (CRP) with parameter $\alpha = 0$ and an initial configuration of k tables associated with nodes in the context where the occupancy of each table is the count of memory accesses to its corresponding node. This yields the same probability of selecting a node as the IBP, but guarantees exactly one connection.

While this attachment procedure follows a least-effort scheme, emergent properties of the network (such as the degree exponent γ) are ultimately determined by how a learning mechanism interacts with knowledge, and no theoretical guarantee of the scale-free property can be made. A guarantee of the scale-free property typically comes from attachments being made to nodes according to their degree of connectivity. This is not anticipated to be problematic for learning mechanisms which interact with knowledge artifacts according to their utility, because the tasks that a mechanism faces should tend to utilize certain concepts more than others, naturally resulting in memory accesses that are proportional to their high utility and thus correlated with high degree of connectivity in the network.

It follows from this specification that the requirements of a learning mechanism in order to utilize `CONTEXTNET` effectively are as follows:

1. The mechanism must be able to **learn** a number of knowledge artifacts, which consist of information that is technically independent of other artifacts. Technical independence here refers to the mechanism's capability of utilizing an artifact without needing any other particular artifact.
2. The mechanism must have the capacity to **consume** knowledge artifacts, leveraging them to learn new richer models.
3. The mechanism must have a quantifiable notion of **usage** of any particular knowledge artifact in its context.

The first two requirements describe a variant of compositional capability. There are reasons to break away from these requirements without changing the specification of `CONTEXTNET`, which are discussed in [Section 4.7.5 `CONTEXTNET`: conclusion and future work](#).

Our implementation of `CONTEXTNET` is available at <https://github.com/lucasem/context>.

4.7.4 Experiment: CONTEXTNET atop EC

To experiment with CONTEXTNET, consider the class of problems for which this system would excel: complex domains where modular knowledge improves the efficiency of completing relevant tasks. There are many learning mechanisms which are suitable for experimentation, such as inductive logic programming (ILP) systems or algorithms which perform search on a grammar. Distinct knowledge artifacts for ILP systems could be sets of predicates, and for grammar search systems they could be language artifacts which construct a mixture model on which search is performed. In this section, we apply CONTEXTNET to a problem of learning effective grammars. The grammar learner consumes an initial grammar and solves tasks to construct a more effective grammar, yielding a learning mechanism which fits the requirements we enumerated above. We augment the grammar learner to utilize CONTEXTNET and task it with solving string transformation tasks. We demonstrate that CONTEXTNET improves the effectiveness of the baseline grammar learner.

Materials

Learning mechanism

We augment the EC algorithm²⁷ [Dechter et al., 2013], a grammar learner build atop combinatory logic (c.f. Section 3.4 Combinatory logic). The augmentation defines the grammar based on learned language artifacts stored *in the context*, rather than artifacts stored in a small bounded set constructed dynamically on every task set²⁸. We use CONTEXTNET iteratively, referring to each iteration as a *phase* and the complete ordered set of phases as the *curriculum*. Within each phase lies a set of tasks used with a number of iterations of the EC algorithm. The augmented algorithm interacts with the knowledge network at the beginning and end of each phase.

The initial grammar employed at the beginning of each phase is defined using the set of combinators in the current context:

$$C := \bigcup \text{GET}() = \bigcup_{i=1}^k \mathcal{I}_i$$

²⁷ The details of the EC algorithm are not important here. For more information on the EC algorithm, consult the original publication [Dechter et al., 2013] or our extension of it that features amortized inference, a different program representation, and more sophisticated compression that is detailed in Section 4.6 Language bootstrapping.

²⁸ Without augmentation, the EC algorithm statically maintains a set of primitives and dynamically introduces new ones as a function of the tasks at hand.

At the end of each phase, we must determine whether the context should be adjusted and whether to add a new knowledge artifact. We refer to the new set of combinators constructed by the grammar learner as C^* .

Consider the set of combinators beyond the context:

$$C' := \bigcup \text{EXPLORE}()$$

If the most probable combinator c_i in C^* is also in C' (in other words, the grammar learner's preferred primitive is defined within one edge of the current context), we adjust the context according to the knowledge artifact which contains c_i :

$$\exists \mathcal{I}. \mathcal{I} \in \text{EXPLORE}() \wedge c_i \in \mathcal{I} \implies \text{ORIENT}(\mathcal{I})$$

We subsequently recompute C' with the new state of the knowledge network. We check the P most probable combinators in C^* , referred to as $C^{(P)} \subseteq C^*$, for presence in C' , and add a new knowledge artifact with every combinator not in C' :

$$(\mathcal{I} := C^{(P)} \setminus C') \wedge |\mathcal{I}| > 0 \implies \text{ADD}(\mathcal{I})$$

The resulting system, parameterized by P , is such that the current context may appropriately be identified as domain-specific knowledge, where a knowledge artifact is a set of combinators which are used as primitives for a grammar defined on the context. [Figure 4-22](#) illustrates a knowledge network that arises from this system.

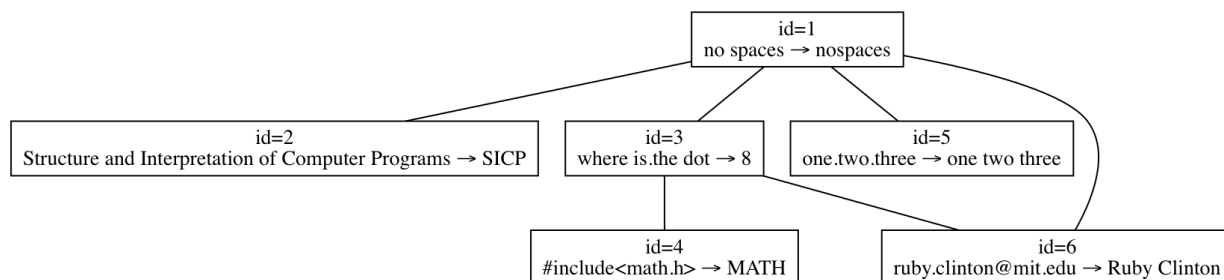


Figure 4-22: A knowledge network after learning programs with the EC algorithm. For demonstration purposes, each node displays the id corresponding to the phase at which the artifact was learned, and an example of a task from that phase. Actual knowledge artifacts contain learned combinators, not the content shown here. Older artifacts, towards the top, have high connectivity because they contain concepts that are useful.

Task domain

We test the augmented EC algorithm in the domain of string transformation [Gulwani, 2011; Lin et al., 2014], chosen because it has potential for common underlying structure between complex tasks. Microsoft Excel’s FlashFill implements program induction for string transformations, but has not been demonstrated to utilize composition over previously learned programs. To handle this task domain, we provide our augmented EC algorithm with the primitives shown in Figure 4-23. Examples of string transformation tasks are shown in Figure 4-24.

<code><ascii chars>...</code>	<code>char</code>
<code>zero</code>	<code>int</code>
<code>empty</code>	<code>str</code>
<code>string_of_char</code>	<code>char → str</code>
<code>string_of_int</code>	<code>int → str</code>
<code>upper</code>	<code>str → str</code>
<code>lower</code>	<code>str → str</code>
<code>capitalize</code>	<code>str → str</code>
<code>concat_on_spaces</code>	<code>str → str</code>
<code>replace_substr_first</code>	<code>str → str → str</code>
<code>replace_substr_all</code>	<code>str → str → str</code>
<code>incr</code>	<code>int → int</code>
<code>decr</code>	<code>int → int</code>
<code>word_count</code>	<code>str → int</code>
<code>char_count</code>	<code>str → int</code>
<code>find_char</code>	<code>char → str → int</code>
<code>substr</code>	<code>int → int → str → str</code>
<code>replace</code>	<code>str → int → int → str → str</code>
<code>nth</code>	<code>int → str → str</code>
<code>fnth</code>	<code>(str → str) → int → str → str</code>
<code>feach</code>	<code>(str → str) → str → str</code>
<code>is</code>	<code>str → str → bool</code>
<code>filter_words</code>	<code>(str → bool) → str → str</code>

Figure 4-23: Primitives for string transformation given to the EC algorithm, and their associated types. See Section 3.4.1 Polymorphic typed combinatory logic for an explanation of the syntax used here for types.

The string transformation tasks were first separated into three distinct subdomains, each of which contained tasks of varying complexity that respectively culminated in tasks 5, 15, and 20 shown in Figure 4-24. We then separated the tasks within each subdomain into smaller sets of tasks which would be supplied to the augmented EC algorithm in a single phase. We refer to these as *phasic task sets*. The structure of these phasic task sets is much like a school curriculum — they start simple and get more complex as foundational concepts are learned. See Figure 4-25 for an example of a phasic task set and solutions to its tasks.

composition. We perform two analyses with specialized grammars: *per-phase* and *full-domain*. For the per-phase specialized grammar, we independently perform many iterations of the EC algorithm on each phasic task set. This will tailor the learned grammar to the tasks at hand, while not relying on learned grammars from any other phasic task set. For the full-domain specialized grammar, we pool together every task into one massive task set, and run many iterations of the EC algorithm on the that task set. This will demonstrate that the task domain is broad and yields ineffective learning when attempted at full scale. For the contextual grammar, we first learn according to the curriculum, then reiterate the curriculum and use results from the reiteration. We reiterate because we are interested in the contextual grammar’s results after it has already learned the material and automatically formed a CONTEXTNET. In other words, we are interested in the *recall* of contextual knowledge. This will yield a large — but not comprehensive — grammar corresponding to whatever artifacts are in the CONTEXTNET’s context during the particular reiterated phase.

The total clock-time measurement is simply the time spent on each phase. The EC-iteration clock-time measurement is the time it took the final iteration of the EC algorithm to enumerate particular solutions for each task, which demonstrates the expressiveness of the grammar that the EC algorithm ultimately utilized. The likelihood measurement effectively represents the probability of an expression given the grammar in the final iteration of the EC algorithm. All results use the same parameters for the EC algorithm except for the number of iterations: The primitive grammar uses only one iteration, the specialized grammars use five iterations, and the contextual grammar uses five iterations when initially learning from the curriculum and only one iteration during recall when we take measurements.

Results

To demonstrate that CONTEXTNET provides automatic compositionality, we anticipate the contextual grammar to out-perform all other grammars in terms of total speed and to perform at least as well as all other grammars in the solve speeds measured during the final iteration of the EC algorithm, and we anticipate the solution likelihoods with the contextual grammar to be about the same as with per-phase specialized grammar. The faster total speed is expected because the contextual grammar is running only one iteration of the EC algorithm — we can get away with this because the contextual grammar should start

each phase with an expressive grammar already prepared, whereas the specialized grammars must spend more iterations to learn such a grammar. The at-least-as-fast speeds in the final iteration are expected because the contextual grammar would have, during a previous phase, already performed the EC algorithm on the same tasks for the same number of iterations as the per-phase specialized grammar. This does not imply that such measurements are meaningless, because the contextual grammar must rely on an expressive contexts that do not contain too many irrelevant combinators which would increase the enumeration time for EC. The similar solution likelihood is expected because the contextual grammar should find the same solutions as the per-phase specialized grammar. Additionally, we anticipate the full-domain specialized grammar to perform poorly in all measurements because the domain is sufficiently broad to deem a single unified grammar poorly expressive for many tasks. Our results are consistent with these expectations.

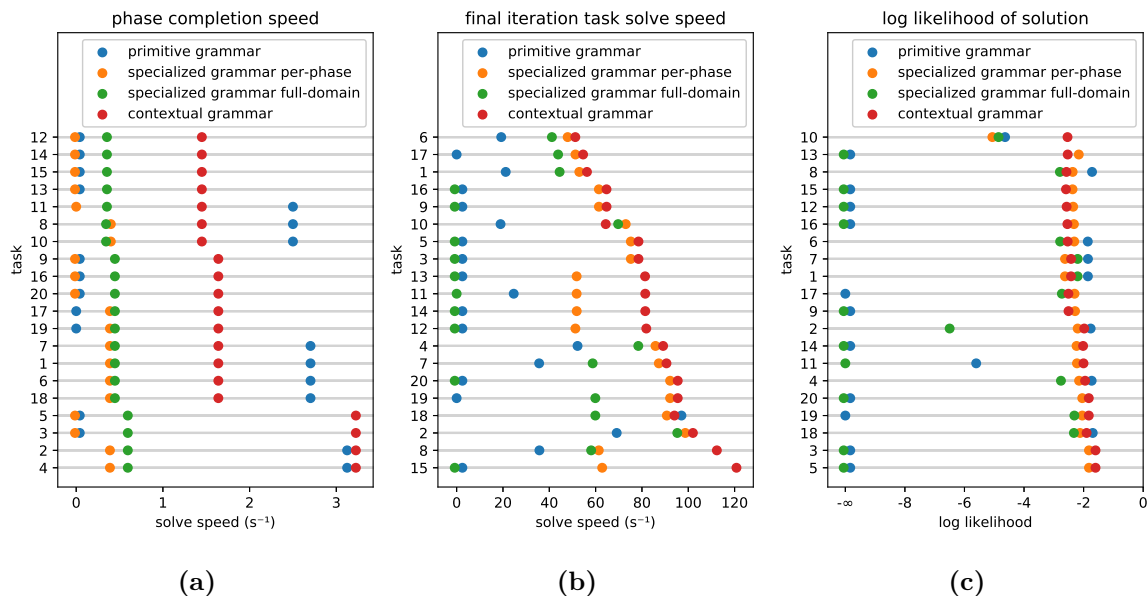


Figure 4-26: Tasks are automatically ordered based on performance in each metric. (a) The speed at which phases were completed, shown for each task. (b) The speed at which tasks were solved in the final iteration of the EC algorithm. (c) The log likelihood of solutions for tasks conditioned on the grammar.

The comparison of phase completion speeds is shown in Figure 4-26a. Because the primitive and contextual grammars used only one iteration of the EC algorithm, they are much faster than either of the specialized grammars.

The comparison of solution speeds in the final iteration of the EC algorithm is shown in [Figure 4-26b](#). The performance of the contextual grammar is almost always superior to every other grammar. The shortcomings of the primitive grammar are due to its lack of compositional learning. Though it can solve every task we provided, the per-phase specialized grammar occasionally lacks the performance of the contextual grammar because it does not also utilize learned grammars from previous phases that are contextually relevant. The full-domain specialized grammar is unable to solve every task due to the large number of tasks it faced, especially given the constraints on the search depth and the number of iterations of the EC algorithm.

The comparison of solution likelihoods is shown in [Figure 4-26c](#). The per-phase specialized and contextual grammars perform very similarly due to the similarities in their respective learned combinators. The full-domain specialized grammar generally performed worse than either the per-phase specialized or contextual grammars, though it is superior to the primitive grammar with few exceptions.

4.7.5 CONTEXTNET: conclusion and future work

We introduce the problem of contextual learning, and observe the costly oversight of the lack of attention to knowledge structure in many existing learning mechanisms. We propose a solution that enables compositional learners to become scalable contextual learners. Our solution features a novel method of automatically structuring knowledge inspired by patterns found in natural processes, and is designed to interface well with existing learning mechanisms. This makes it an ideal candidate for utilization with any mechanisms that satisfy the requirements described in [Section 4.7.3 CONTEXTNET](#), whether to help scale a mechanism to broad domains or to use as a unified learning framework for many mechanisms. We applied CONTEXTNET to one such mechanism, the EC algorithm, and demonstrated that CONTEXTNET is effective at enhancing learning in broader domains.

The CONTEXTNET system as experimented on in this paper is missing an important element of *context localization*. We believe a major step in improving CONTEXTNET’s effectiveness is to introduce an artificial neural network with an online learning algorithm which maps perceptual input to items in the context. Such an approach must iteratively expand its memory as new knowledge artifacts are learned by other mechanisms, and make calls to $\text{ORIENT}(\mathcal{I})$ whenever deemed effective. This would introduce the green lines illustrated in

the updated CONTEXTNET schematic of Figure 4-27.

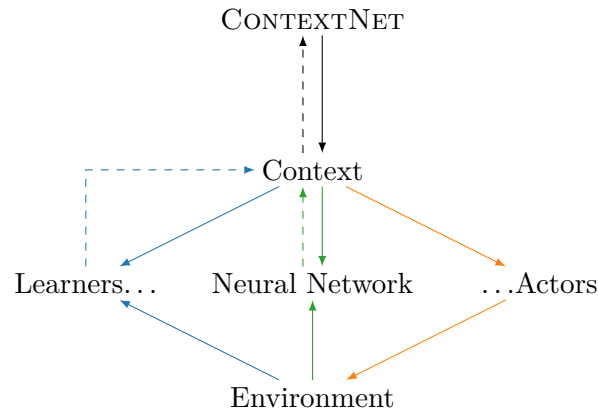


Figure 4-27: Schematic of the function of CONTEXTNET with desired future features, adopted from Figure 4-19. Green lines indicate connections for a neural network tasked with context localization. The “mechanism” is generalized to any number of learners, indicated with blue lines. Actors which interact with the environment are indicated with orange lines.

The CONTEXTNET system can be utilized in many different ways. By connecting many learning mechanisms to the same knowledge network, interoperability between distinct learning mechanisms such as for vision and sound can result in proximal — and perhaps even shared — knowledge artifacts representing the same *multi-modal* symbol being perceived by different mechanisms (see the blue lines in Figure 4-27). Because learning mechanisms may treat knowledge artifacts as first-class objects that can be observed or created, the mechanisms do not necessarily have to be restricted to *learners* — the introduction of *actors* as mechanisms would yield a production system conditioned on contextual knowledge and immediate perceptual information (see the orange lines in Figure 4-27). Supplemented with support for direct communication between mechanisms, collections of mechanisms can be structured and would be more fittingly referred to as *agents* [Minsky, 1988]. The CONTEXTNET system, in conjunction with these agents, should be tested as a model of cognitive architecture.

Chapter 5

Towards Formalized Conceptual Role

5.1 Introduction

Conceptual role semantics (CRS) is a philosophical framework that aims to answer many of the questions posed in [Chapter 2 Computational Framing of Representation and Learning](#) [Field, 1977; Harman et al., 1982; Block, 1986]. The approach of CRS accepts that representations attain meaning through their relationships to other representations. For example, from Piantadosi [2016]:

There is nothing inherently disjunctive about your mental representation of the mental operation OR. What distinguishes it from AND is that the two interact differently with other mental tokens, in particular TRUE and FALSE.

This is evident in how people define terms (e.g. the reader may attempt to define “computation”): by using other terms that may themselves need to be defined. The *role*, or functional relevance, of a concept is effectively what *defines* the concept and what gives it meaning. CRS and the theory theory of concepts (c.f. [Chapter 2 Computational Framing of Representation and Learning](#)) provide the same accounts for conceptual structure [Brigandt, 2004]: loosely, the *inferential role* of concepts in CRS accords to the *explanatory* capacity of intuitive theories while also providing grounds for cognitive development. The concepts described by CRS are inherently abstract, much like intuitive theories — however, we discuss the “grounding” of these cognitive phenomena in [Section 5.4 Perception, mem-](#)

ory, and cognition. As mandated by theory theory, locally connected concepts¹ according to role may be coherent simply by construction of concept (and their roles). Ensuring causal structure is a more challenging problem when adopting the perspective of CRS. This may be mitigated within our proposal by supplanting strong immutable priors on a set of concepts which collectively determine causation. In this chapter, we invoke CRS in a language of thought to construct — speculatively — a computational formalism for conceptual role. In doing so, we provide answers to the following questions posited in [Chapter 2 Computational Framing of Representation and Learning](#):

- I. How are concepts individuated?
- II. Why are concepts useful?
- III. What is the representation of concepts?
- IV. How does concept learning manifest?
- V. By what procedures are concepts learned?
- VI. To what extent is the representation of concepts *learned*?
- VII. To what extent are learning procedures for concepts *learned*?
- VIII. Where do concepts meet perception?

5.2 Representation and role

A computational formalism of conceptual role necessitates a concrete definition of *role* that may be modeled computationally, as well as a declaration of what mental representations should be present. We adopt the direction of [Piantadosi \[2016\]](#) that models of the world — from environmental and physical phenomena to agents and their actions to abstract concepts such as *number* — are effectively emulated in the mind. This emulation is carried out by mental operators whose behavior produces an *isomorphism* to a real world system. With this framing, adopting a Turing-complete computational representation for mental operators necessarily provides this capability. In this section we discuss a high-level representation

¹ Connectivity in the setting refers to the relations between concepts within some conceptual structure or intuitive theory. Hence, local connection corresponds to some sort of domain-specific or contextual co-relevance.

that can be grounded in structured computation which is implementable by connectionist networks².

We propose that a suitable representation is a pure type system, and we remind the reader that *types* are sets of values³. As we described in [Section 3.7 Pure type systems](#), adopting a type system permits us to declaratively express concepts without necessitating concrete code that grounds out in some primitive language. In other words, we can express **role through type declarations**, which define types or relations on types by means of other types. [Figure 3-21a](#) illustrates conceptual role via types: we can model classes of object like *container*, with instances like *box* or *shed*; if a container has distinct objects and not something liquid, then the container is *traversable*; if something is traversable and its items are orderable, then we can *sort* those items. While we used English words here to aid the reader’s understanding of our example, the words themselves carry no meaning in this context; meaning is derived from the relationships between concepts/types. Furthermore, by working within a type system, **coherence is guaranteed** because nonsense values and inconsistent behavior are not representable⁴.

We believe this representation may best be understood by regarding programmers as translators ([Figure 5-1](#)). A computer programmer maintains a mental model for some

² See Section 5 of [Piantadosi \[2016\]](#) for an exposition of this feat.

³ In a pure type system, there may be “higher order” types which are themselves sets of types.

⁴ We mean inconsistent behavior in accordance to an ascribed type. If a type itself is effectively nonsense, but it is valid within the system, then it may be constructable. This is not problematic as CRS does not posit that there necessarily is an objective notion of meaning. I.e., people are not harbingers of truth and may have mental representations for concepts which have no apparent analogue in the mental representations of other people.

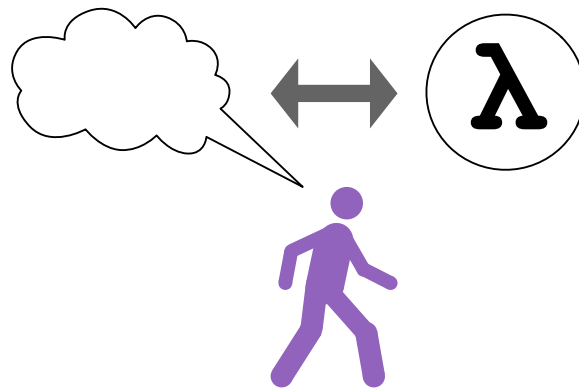


Figure 5-1: Programmer as translator. A programmer is tasked with translating mental models (left) into a programming language (right), and vice-versa when interpreting foreign models.

“Sort” takes a sequence of things which are orderable,
and gives a sequence of those things in order.

(a)

$$\text{sort} : \text{Ord } T \Rightarrow \{i : \text{List } T\} \rightarrow \{v : \text{List } T \mid \text{elems}(i) = \text{elems}(v) \\ \wedge \text{nondecreasing}(v)\}$$

(b)

Figure 5-2: Natural language and types. Both provide declarative means of descriptive communication, and they may correspond in obvious ways as illustrated here. “Sort” leverages existing concepts like quantification (“things;” polymorphism over T), “orderable” (Ord), “sequence” (List), order-independent equivalence between containers (“of those things;” $\text{elems}(i) = \text{elems}(v)$), ordered sequences (“in order;” $\text{nondecreasing}(v)$), and conjunction (implicit in natural language; \wedge).

computational workflow and must translate this model into a programming language. Many high-level programming languages prioritize ergonomics to make this translation process easier. For the programmer, an entire computational workflow can be modeled using only type declarations — without having to write any concrete code. We ask, “what is the best programming language?” The answer is, for the purposes of cognitive science, whatever language the programmer uses to construct mental models. The key intuition here is that type systems serve as a framework in which programmers represent concepts. However, these rich type systems are merely the current cutting-edge of what programming languages provide. Perhaps in the future, natural language may be translated into formal structure via declarative types as demonstrated in [Figure 5-2](#).

At the beginning of this chapter, we mentioned the importance and non-triviality of causal structure within CRS. An approach to resolve this is to introduce types that carry causal meaning. While nature has had the opportunity to construct causal models through natural evolution, we must reverse-engineer whatever *causality* is in order to supplant the notion within our framework. The recently-developed “interventionist” account of causation takes inspiration from causal Bayes nets: A causes B if, other things equal, an intervention on the probability distribution associated with A yields a change in the probability distribution associated with B [[Woodward, 2003](#); [Gopnik and Wellman, 2012](#)]. In tune with our acceptance of CRS, this supports role-based understanding: the mechanistic details of a causal process are unnecessary for the understanding that causation is present. While the details of the instantiation of such a causal framework in the mind are unknown to us, we

can nonetheless adopt it in a type system as demonstrated in Figure 5-3.

```

Distribution = {D | ...}
Event = {
  E |  $\exists \text{Env}. \exists f:\text{Env} \rightarrow \text{Distribution}.$ 
       $\forall \text{env}:\text{Env}. (\text{observe}(\text{env}) \in E) \sim f(\text{env})$ 
}
cause : Event  $\rightarrow$  Event  $\rightarrow$  PROP
cause a b =  $\exists b:\text{Event}.$  cause a c  $\wedge$  cause c b
 $\wedge$  let JointEnv = a[Env]  $\cup$  b[Env] in
   $\exists \text{perturb}:\text{JointEnv} \rightarrow \text{JointEnv}.$ 
    not_identity(perturb)
 $\wedge \forall \text{env}:\text{JointEnv}.$ 
      a[f](perturb(env))  $\neq$  a[f](env)  $\Rightarrow$  b[f](perturb(env))  $\neq$  b[f](env)
       $\wedge$  a[f](perturb(env)) = a[f](env)  $\Rightarrow$  b[f](perturb(env)) = b[f](env)

```

Figure 5-3: Causation modeled in a type system. We use orange to indicate a set of types (i.e. a set of concepts), SMALL-CAPS to indicate a type (i.e. a concept), green to indicate a type variable (i.e. a concept variable), blue to indicate an instance variable (i.e. an inhabitant of a concept), red for presumed procedures which may be innate or defined elsewhere, and purple for defined procedures. Square brackets indicate “indexing” into existentially quantified variables, the details of which are beyond the scope of this work. In words, a concept is an *event* if there is an environment and a function on that environment which determines whether the concept is observed. Event *a* causes event *b* (a proposition PROP) if there is an intermediary event *c* such that *a* causes *c* and *c* causes *b*, or if there is a nontrivial perturbation on the environment where a change to the distribution of event *a* implies a change to the distribution of event *b* and the *counterfactual*, that no change for *a* implies no change for *b*. This could be modeled differently, such as with metrics of comparison between distributions, but our illustration suffices. To illustrate *placeholder* capability for causation, one only needs to introduce disjunctive statements to the system. For example, “cause SomeConcept SomeOtherConcept” alone suffices to indicate another inhabitant of “cause” without necessitating use of its abstract definition.

A seemingly profound and controversial statement is made in our computational model of causation: that concepts may be events rather than taking an event to be a particular object or real-world entity. We do not mean “object” in the physical sense; we instead adopt that an object is anything which may satisfy the abstract determination of a concept. The statement of our model is actually not such a significant claim because of an important

feature of our representation: *singleton types*. A singleton type is a type with exactly one inhabitant, or under our interpretation, a concept that refers to a particular object or real-world entity. With singleton types, a particular entity may occupy the conceptual status of event by representing that entity as a concept.

A *collapsed type hierarchy* allows for mathematical inconsistency that seems necessary for a model of cognition: e.g. Russell’s paradox⁵ should be representable as well as the paradoxical phrase, “there is no absolute truth.” These paradoxes, despite being so, carry meaning as concepts that people have expressed, and can further provide useful tools for critical thinking by challenging overlooked assumptions or by promoting alternative perspectives. These paradoxes both rely on self-reference, which can be achieved in a type system whether by procedural recursion at the object-level, inductive recursion at the type-level, or by self-inhabitation between types and higher-order types. Self-inhabitation is granted by the collapsed type hierarchy, while procedural and inductive recursion are granted without necessitating inconsistency. Furthermore, dual factor theory [Block, 1986; Carey, 2009] is trivially supplied by a collapsed type hierarchy — we will not get into the details of the theory for this reason, but it is of note for cognitive scientists who question the compatibility of the theory with our approach⁶.

We therefore have answers for questions I-III listed at the beginning of this chapter: (I) we individuate concepts by their distinct computational functions⁷; (II) concepts are useful by giving meaning to other concepts through conceptual role; (III) we represent concepts by their role, as a type.

5.3 Learning and conceptual change

In this section, we discuss what constitutes *learning* in a type system. We appeal to program induction (c.f. [Chapter 4 Concept Learning by Design: Program Induction](#)) for its approaches to tackling many of the hard learning problems under our interpretation. As

⁵ In short, Russell’s paradox is that if we define S to be the set of all sets that don’t contain themselves, i.e. $S = \{x | x \notin x\}$, then it follows that $S \in S$ iff $S \notin S$. In formal mathematics this is typically resolved by the axiom of specification. In fact, the origin of types themselves was from Bertrand Russell’s efforts to mitigate this paradox [Russell, 1903; Whitehead and Russell, 1912].

⁶ We believe questions regarding this compatibility and “trivial” satisfaction of dual factor theory are clarified and answered in [Section 3.7 Pure type systems](#) and this chapter.

⁷ If two types, though defined differently, have the same inhabitants, then they are equivalent. In logic and computer science terminology this is *extensional* equivalence, contrasted with *intensional* equivalence where the definitions themselves are the basis of equivalence.

such, our approach is compatible with theories of learning based on Bayesian models by integration with BPL (c.f. [Section 4.5 Bayesian program learning](#)). By the end of this section, we’ll have answered questions IV-VII listed at the beginning of this chapter: each subsection provides some account for questions IV and V, while [Section 5.3.2 Consolidation and abstraction](#) and [Section 5.3.3 Conceptual change with discontinuity](#) both tackle question VI and [Section 5.3.5 Inducing search procedures and probabilistic models](#) particularly tackles question VII.

5.3.1 Synthesis

To mentally formulate entities that bear a concept, program synthesis may be used to find inhabitants of a type. E.g., this is to go from the abstract concept of “dog” to a particular instance of a dog, from an entrée like “eggplant and zucchini parmesan” to a recipe for its construction, or from the abstract concept of “sort” to a sorting procedure. The approach of *type-directed synthesis* we described in [Section 4.4.4 Deductive search](#) is particularly relevant in this setting, as the example from [Figure 4-3](#) fits exactly with the example of sorting described here. We further note that types need not be expressed with such intentional rigor as in “sort;” types can be expressed using operations on sets such as intersection and union⁸ or by explicitly denoting inhabitants of a type, thereby permitting synthesis problems to be formulated *by example*. For example, using this scheme for type specification, a task for repeating a sequence may have type $\{f:[T] \rightarrow [T] \mid f([]) \mapsto [] \wedge f([1]) \mapsto [1, 1]\}$ before a more abstract description is determined⁹.

Synthesis is a generative process, whether by constrained search-based methods of type-directed program synthesis or by sampling from a probabilistic generative model. We hereafter refer to the latter as sampling from a *probabilistic program*, where a probabilistic program is a probabilistic model defined by random choices in some program. Causal structure learning has been widely observed in development and Bayesian models of that learning have provided new insights [[Griffiths and Tenenbaum, 2009](#); [Schulz, 2012b](#); [Gopnik and Wellman, 2012](#)]. Because of peoples’ ability to learn and mentally manipulate causal structure, we believe it is important that natively probabilistic computation is accounted

⁸ Depending on the choice of priors, the conceptual procedures of intersection and union may be considered either innate or learned — we need not presume of one of these cases as they are effectively created equal within the type system.

⁹ The determination of a more abstract description is discussed in [Section 5.3.2 Consolidation and abstraction](#) and [Section 5.3.4 Construction of concepts](#).

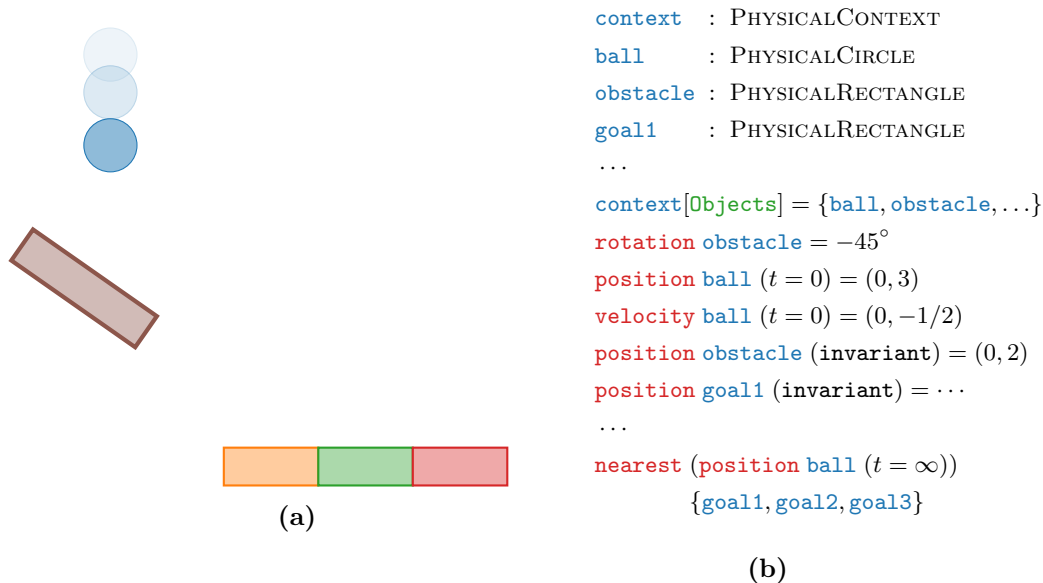


Figure 5-4: A physical system. The pseudocode in (b) is mostly object-level and not type-level. These statements form *constraints* which determine the nature of `ball`, the underspecified object. When an object for `ball` is discovered by synthesis, an inhabitant for the final statement (which is not a constraint because it lacks assignment) may be synthesized. The *priors* here are the implicit values that there is acceleration due to gravity on all objects and that velocity is zero unless otherwise specified. These priors are established as a function of perception, as discussed in [Section 5.4 Perception, memory, and cognition](#).

for. Sampling procedures for probabilistic programs are viable synthesis mechanisms in our computational framing. Later in [Section 5.3.5 Inducing search procedures and probabilistic models](#) we will show that these probabilistic models are expressible and learnable in our framing.

Synthesis is useful for more than just the kinds of problem-solving that obviously translate to finding an inhabitant of a type (as with the examples of a recipe or an algorithm). It supplies a means for *simulation*, which involves finding the inhabitant of a type under various other type constraints. A keen example is in the simulation of the dynamics of a physical system, illustrated in [Figure 5-4](#). The constraints here are hand written, but we will discuss in [Section 5.4 Perception, memory, and cognition](#) how they may be constructed.

Similarly, synthesis is also useful for *hallucination*. Hallucination comes for free with synthesis-based simulation for reasons we discuss in [Section 5.4 Perception, memory, and cognition](#).

5.3.2 Consolidation and abstraction

There is evidence in neuroscience for the consolidation of memory and creation of abstractions during sleep [Dudai et al., 2015]. Within our framing, existing information can be reorganized to be more efficient: program synthesis for language bootstrapping provides “refactored” implementations which keep the same types and behavior. This kind of refactoring reduces software bloat¹⁰ and makes solutions to probable tasks more accessible. We refer the reader to [Section 4.6 Language bootstrapping](#) for a deeper dive into how this consolidation process may be computed.

Under our representation, consolidation in this form can manifest in many ways. One variety of *concept construction* (which we will discuss further in [Section 5.3.4 Construction of concepts](#)) arises from consolidation through abstraction. Such as the transformation from concepts to an abstract consolidation illustrated in [Figure 5-5](#).

<code>add2 ℓ = map (λx. (plus x two)) ℓ</code>	<code>add2 = addk two</code>
<code>add3 ℓ = map (λx. (plus x three)) ℓ</code>	<code>add3 = addk three</code>
	<code>addk k ℓ = map (λx. (plus x k)) ℓ</code>
(a)	(b)

Figure 5-5: In the work of [Section 4.6 Language bootstrapping](#), we “compressed” common code in (a) by creating reusable helper functions as in (b), making empirically relevant concepts more accessible for future learning. We note that the arithmetic and natural numbers assumed in this example may be learned, and we remark that as discussed earlier in this chapter, an item like “two” can exist as both a type (i.e. concept) or as an object (i.e. something that inhabits a concept).

Consolidation also provides a means for continuous conceptual change, where different abstractions may be used to represent the same concept. These abstractions still express the same concrete treatment of the concept because they are induced through the same procedure (i.e. efficient abstraction) from the same episodic experiences (i.e. program segments, whether at object-level or type-level). Hence we refer to transformations between use of these abstractions as *continuous* conceptual change. The discontinuous variety is discussed in [Section 5.3.3 Conceptual change with discontinuity](#). The decided use of one representation (i.e. set of abstractions) over another is be discussed in [Section 5.4 Perception, memory,](#)

¹⁰ Software bloat is when many parts of a system are not useful, yielding unnecessary sophistication and reduced performance.

and cognition.

Search procedures and probabilistic models are consequently learnable with this approach, as will be made more clear in [Section 5.3.5 Inducing search procedures and probabilistic models](#). In particular, specific inductive or deductive search procedures may be abstracted into general procedures which are broadly useful, or probabilistic models may be consolidated by modeling them as conditional distributions of some more general probabilistic program.

5.3.3 Conceptual change with discontinuity

A computational theory of concepts must account for the discontinuous representation transformations that people encounter, both in development for concepts like “number” and “matter/weight/density” and in adulthood as the history of science introduced concepts like “temperature/heat” and “magnetism” [[Carey, 2009](#)]. These conceptual discontinuities are of at least two varieties: increase in expressive power, and *incommensurabilities* [[Kuhn, 1965](#); [Carey, 2015](#)]. The former comes with an apparent parent/child relation between two conceptual structures, where the parent is less efficient than the child, but both are comparable and derived from the same foundations. For example, in a digital numerical system where addition is well-understood, two numbers may be multiplied by repeated addition as many times as one of the multiplicands, or by the “lattice method” involving adding the digits that arise from smaller multiplications — see illustration in [Figure 5-6](#). Incommensurability arises when conceptual structures are essentially incompatible: there is no effective means of comparison and the new concept does not logically follow from the old concept. These two varieties of discontinuous change are related to material we save for [Section 5.3.4 Construction of concepts](#), in particular corresponding to *consolidation* and *intensional learning* for commensurable change and *placeholder filling* for incommensurable change.

In our computational approach, this has an analogue to keeping some consistent behavior, but changing the types — i.e. type-level refactoring (contrasted with implementation-level refactoring in [Section 5.3.2 Consolidation and abstraction](#)). By “keeping some consistent behavior,” we mean that certain roles of each conceptual structure correspond. For example, the notion of “number” may be associated with some particular conceptual structure, and during discontinuous conceptual change *that same notion* of “number” still exists

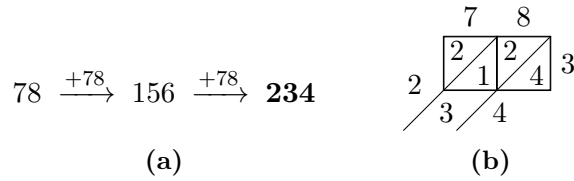


Figure 5-6: Multiplication of 78 by 3 using (a) repeated addition and (b) the lattice method. These multiplication algorithms *are* commensurable as the latter has an apparent logical construction derived from the former.

and has role similarly tied to the new conceptual structure. This effectively *changes the way the system think about things* but knowledge may still transfer over to the new conceptual structure — as will be established in [Section 5.4 Perception, memory, and cognition](#). An apparent example of discontinuous conceptual change in a type system is between incremental numbers and digital numbers, as illustrated in [Figure 5-7](#).

```

enum NAT {
    Zero,
    Succ(NAT),
}
let twenty : NAT = Succ(Succ(Succ(Succ(
    Succ(Succ(Succ(Succ(
    Succ(Succ(Succ(Succ(
    Succ(Succ(Succ(Succ(
    Zero))))))))))))));

```

(a)

```

enum DIGIT {
    Zero,
    One,
    ...,
    Nine,
}
type NAT = [DIGIT];
let twenty : NAT = [Two, Zero];

```

(b)

Figure 5-7: Conceptual change as *type-level refactoring* from a representation of natural numbers that is (a) incremental, as in Peano arithmetic, to (b) digital, as in the Arabic numeral system. Not shown are the corresponding implementations of addition (which is more efficient in the digital system for large numbers), number-word generation (e.g. speaking “twenty”), and other already-established numerical operations. This code is written more concretely in the Rust programming language because we think it is more illustrative and expressive for this example.

Another related feature is that of thinking at different levels of abstraction. For example, consider arrangements of flower petals, such as the one illustrated in [Figure 5-8](#). One might explain this natural occurrence by reasoning about optimal sunlight exposure for each petal

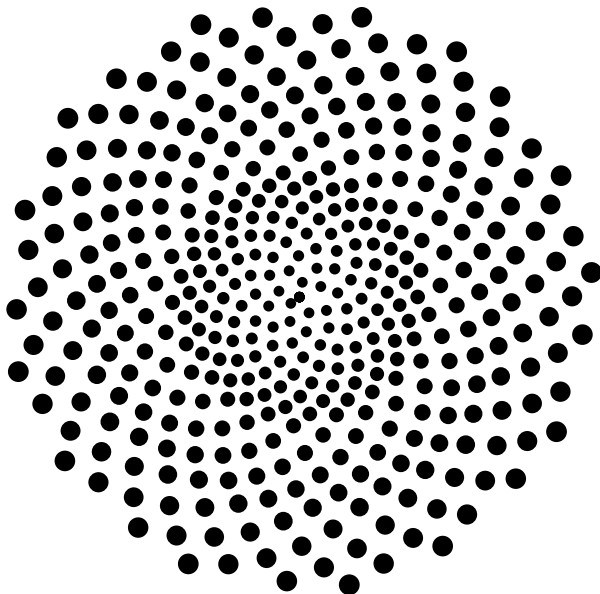


Figure 5-8: Repeated phyllotactic spirals according to the Fibonacci sequence, a natural phenomenon demonstrated in sunflowers, pineapples, and other plant-life. The number of petals in a flower is generally a Fibonacci number (e.g. the lily flower has three petals, often repeated twice, and the buttercup typically has five petals). When the petals repeat, they form two repeated spirals (clockwise and counter-clockwise) each of which occurs a Fibonacci number of times.

that is tolerant to over-growth in the number of petals: by spacing them evenly apart according to an irrational number, repeated layers won't significantly limit the sunlight exposed to previous layers¹¹. Alternatively, one might think of this as the result of a generative process by which seeds travel during growth, where the travel is a function of rotation and distance from the stem. Yet another way to think about this is by the inter-cellular mechanics from which this travel emerges. Finally, one may think about this as a byproduct of evolution: sure there is something useful for survival and proliferation given by obeying such a pattern, one might imagine that these plants grew faster and procreated faster than plants without this property, hence consuming more land and preventing the less-biologically-competitive alternative from surviving many generations.

We have the capacity to entertain many layers of abstraction in our thought, despite them all being of the same concept (e.g. phyllotaxis as a discriminative property, as a generative process, as emergent from low-level biological mechanism, or as a mutation that

¹¹ With a rational number, such as 2 or 1/2, leaves alternate on opposite sides of the stem thereby limiting sunlight exposure for lower layers. If the leaves are grown far-enough apart, however, this exposure is unaffected, which is also observed in nature.

yielded evolutionary advantage). This is achieved in our system by *maintaining transformations* between conceptual structures where many of those structures remain useful despite the other’s existence. A transformation may look like a function which goes from the NAT of Figure 5-7a to the NAT of Figure 5-7b. We will look into the “usefulness” notion in Section 5.4 Perception, memory, and cognition, as it distinguishes whether conceptual change occurs in a backwards-compatible way (i.e. conceptual structure is maintained) versus an incommensurable and reconstructed way (i.e. conceptual structure is replaced).

5.3.4 Construction of concepts

Where does knowledge come from? How do people imagine new ideas, whether consistent with observation or nonsense? In this section, we provide a computational account for constructivism under our role-based representation in attempt to aid the resolution of a core problem in cognitive science [Piaget, 1937; Schulz, 2012a]. This process accords to *type generation* in our framing, and it comes in many forms.

Consolidation, as discussed in Section 5.3.2 Consolidation and abstraction, accounts for a variety of concept construction. By creating abstractions over existing concepts, certain generalizations may be made. For example, our DREAMCODER system of Section 4.6 Language bootstrapping can discover the higher-order list function “`filter`” by generalizing from solutions of particular list filtering problems.

Intentional learning is the result of combining existing concepts by association via other concepts. For example, the concept RED-DOG can be constructed from the existing concepts of RED, DOG, and **conjunction** as shown in Figure 5-9. While this is an example of constructive *specification*, where there are fewer inhabitants in the constructed concept, we can also perform constructive *generalization* through means other than those of consolidation. In particular, consolidation can only generalize from concepts which are represented similarly. Such an alternative means of constructive generalization is demonstrated in Figure 5-10. Where consolidation uses abstraction to “zoom out,” here we can “zoom in” by combining ideas in new ways and introducing new and more particular types. This can amount to one-off search or sampling of concepts that fit together (in the type-theoretic way; e.g. you cannot perform procedural application with an object that is not a procedure).

```

RED = { ... }
DOG = { ... }
conjunction : Concept → Concept → Concept
conjunction A B = { a : A | ∃b:B. a = b }
RED-DOG = conjunction RED DOG

```

Figure 5-9: Construction of the RED-DOG concept in a pure type system. Note that while universal quantification is a feature of pure type systems, existential quantification is not. However, it is nonetheless constructable as proven by Geuvers [1993].

```

GÖDEL-INCOMPLETENESS = { ... }
ESCHER-TESELATION = { ... }
BACH-FUGUE = { ... }
disjunction : Concept → Concept → Concept
disjunction A B = { a | ∃C:{A,B}. ∃c:C. c = a }
STRANGE-LOOP = conjunction Self
                (disjunction GÖDEL-INCOMPLETENESS
                 (disjunction ESCHER-TESELATION
                  BACH-FUGUE))

```

Figure 5-10: Construction of an interpretation of the STRANGE-LOOP concept, a thesis of Hofstadter [1979]. Here we assume the conjunction concept defined in Figure 5-9.

Placeholder filling constructs new types by gradually specifying meaning over time. This is analogous to “Quinian bootstrapping” of Carey [2009], which provides flexible ways of introducing partial forms of knowledge. This may amount to intentional learning over time, where search/sampling of underspecified concepts is more common and where mutation is permitted for fragments of those concepts. These types do not need to carry much meaning until they have been constructed sufficiently. I.e., triviality or nonsense may be generated during this process. An example for the beginnings of an idea of Darwinian evolution is illustrated in Figure 5-11. This provides for *novelty* by constructing new concepts and trying to find inhabitants or otherwise make sense of them.

`Organism = ∅`

`Organism = {0 | ∃procreate:0 → 0 → 0. true}`

`Organism = {0 | ∃procreate:0 → 0 → 0. ∀a, b ∈ 0. procreate a b ∉ {a, b}}`

`Organism = {0 | ∃procreate:0 → 0 → 0. ∀a, b ∈ 0. procreate a b ∉ {a, b}}`
`evolution : ∀0:Organism. [0] → [0]`

`Organism = {0 | ∃procreate:0 → 0 → 0. ∀a, b ∈ 0. procreate a b ∉ {a, b}}`
`Environment = {ISLAND}`
`evolution : ∀E:Environment. ∀0:Organism. (E, [0]) → (E, [0])`

Figure 5-11: A placeholder-filling construction for the concept of Darwinian evolution. The first iteration provides a new symbol. The second iteration introduces some notion of procreation (particularly sexual reproduction). The third iteration further specifies that reproduction results in *mutation*. The fourth iteration produces a preliminary idea of evolution, and the fifth iteration introduces something external to the organisms (i.e. an environment) as a factor in that process. We use $[X]$ to denote collections of inhabitants of a particular X .

5.3.5 Inducing search procedures and probabilistic models

We mentioned in [Section 3.7 Pure type systems](#) a programming language feature called “elaborator reflection.” This feature allows certain type-directed synthesis procedures to be implemented *within* the language itself, thereby enabling a form of *learning to learn*. In our framework, this means a synthesis procedure (c.f. [Section 5.3.1 Synthesis](#)) is itself an inhabitant of a concept and therefore subject to all of the learning capacities described earlier in this section.

Similarly, probabilistic models are expressible both as objects and as types within our approach, as are various modeling and inference techniques. These probabilistic models may lie in the role of concepts which are distributional such as the probability that a coin comes up heads. However, probabilistic models may also provide inference schemes in the Bayesian interpretation — probability as a measure of belief — by interacting with information available in the form of objects and types. Probabilistic programs together

with inference techniques provide a means for generalizing functions as distributions and evaluation as as sampling [Mansinghka, 2009]. This is compatible with our system as long as the underlying computational substrate supports nondeterminism.

5.4 Perception, memory, and cognition

In this section, we answer question VIII proposed at the beginning of this chapter: where do concepts meet perception? We start with high-level descriptions of these three factors *as they relate to our approach*. **Perception** is uncontrollable¹² and determines the space in which thought occurs. **Memory** is the integration of those spaces over percepts¹³, including everything that has been both learned and maintained (but neither everything that is learnable nor whatever has been learned and unmaintained/forgotten). **Cognition** is the production of concepts and their inhabitants and the source of “intention.” It occurs within the space that resulted from perception and has the capacity to update memory particularly through the means described in [Section 5.3 Learning and conceptual change](#).

All of the types and constraints assumed in the previous section clearly lead to extreme combinatorial explosion for almost every learning task we discussed. Through perception this space becomes limited, reducing from *all memory* to whatever is known *currently*, so we refer to this space as a “scope.” This scope, while described as a function of perception, may be manipulated to a limited extent at the expense of cognitive effort¹⁴. The role of perception is precisely to limit the scope in which cognition occurs so that it is useful — to find balance between the combinatorial explosion of the space of everything that is thinkable to the narrow but highly-expressive space of whatever is apparently relevant. There is connection here to our work in [Section 4.7.3 CONTEXTNET](#).

Perceptual input analyzers [Carey, 2009] effectively map real-world entities, such as those which are perceived via a determination process atop sensory organs, to symbols which refer to them. In our approach, a symbol may be any item of the type hierarchy: objects (i.e. inhabitants of a concept, which also provide constraints for concepts), types

¹² By uncontrollable, we mean that at the lowest level, observation starts at sensory experience which cannot be influenced by cognition. However, cognition may effect that which is perceived at a conceptual level, which we discuss in this section.

¹³ This integration over perceptual space is shallow — it does not assume any propagation of the tools of [Section 5.3 Learning and conceptual change](#).

¹⁴ An alternative formulation of this is that thoughts are perceived, and hence cognition doesn’t change the scope directly but only through perception.

(i.e. a concept), higher-order types (i.e. sets of concepts). As such, search procedures and probabilistic models are also subject to perceptual localization, per [Section 5.3.5](#). By constructing a set of objects, input analyzers determine a set of concepts that they inhabit. These objects and concepts comprise the scope of constraints and types used in cognition. How do these input analyzers come to be? Developmental science suggests some of them may be innate [[Spelke, 1998](#); [Carey, 2009](#)]. We adopt this notion, and further add that there are constructions that can be made from them. Deductive procedures, most notably conjunction (AND) and disjunction (OR) provide a means for composing perceptual input analyzers without adopting sophisticated machinery reserved for cognition. There is a correspondence between perceptual input analyzers and connectionist models like artificial neural networks which gives insight and motivation for the introduction of such deductive procedures. Such deductive procedures accord to an added layer (increased depth) of neurons in a neural network. An increased in depth yields *exponentially* increased expressivity in a deep neural network, thereby permitting a constructive and connectionist perceptual mapping to range over many concepts not were not effectively expressible in a shallower network [[Poole et al., 2016](#)]. It is for this practical reason that we suggest the capacity for primitive compositions of perceptual input analyzers.

The perception of these phenomena in our framework follows from memory, as manipulated by cognition, and by the synthesis of concept inhabitants (c.f. [Section 5.3.1 Synthesis](#)). Hallucination may occur when conceptual inhabitants are realized that are of a perceptive nature despite not being the result of perception¹⁵. Along these lines, *simulation* is provided. When cognition determines inhabitants of concepts, new constraints may be introduced which further determine constraints on inhabitants of other concepts in scope. The repetition of this process yields simulation. Hence, physical simulation may be scoped in real or hallucinated settings, depending on the determination of perception¹⁶. The presence of probabilistic programs as procedures for determining concept inhabitants permits its use as a simulation mechanism, which conditions on objects in scope to provide sampling and inference.

¹⁵ By “of a perceptive nature” we mean concept inhabitants that are may be determined by the output of some perceptual input analyzer. Otherwise we simply refer to the generated inhabitants as exactly that — not of a “hallucinatory” nature.

¹⁶ The distinction between “real” and “hallucinated” becomes more vague, as both are perceived. As a result, the two need not be distinguished. However, other concepts in scope may associate those objects with a notion of implausibility, thereby creating a determiner that decides whether objects are real or hallucinated. This implausibility distinction would aid in counterfactual reasoning.

Cognition itself involves every procedure of [Section 5.3 Learning and conceptual change](#). The restriction of cognition to *scope* makes this learning tractable by dramatically reducing both the number of constraints and the size of the search space. We note that “learning” is not restricted to concept learning, as our methods can be used to find inhabitants of concepts by propagating type-based constraints. This habitation yields objects which may be perceived as hallucination, as explained in the paragraph above. Habitation may manifest as decision-making, identification of particular causal properties, modifying search procedures and probabilistic models, or the fine-tuning of object-level statements that arose from perception. All of these processes that cognition accounts for demonstrates that conceptual change effectively changes the way the system “thinks.”

Foundational to many models of this chapter has been a notion of *use*. This provides an account for choice of representation — whether conceptual structure is worthy of maintaining — and thereby explains conceptual change. We call this the *reconciliation between co-occurring conceptual structures*: whether to replace-and-transfer from one conceptual structure to another, to keep conceptual structures and maintain transformations between them, or to disregard (i.e. forget) a conceptual structure entirely. We believe that a combination of Kolmogorov complexity and runtime analyses can determine whether a proposed type-level restructuring is worthy of one of these cases [[Solomonoff, 1964](#); [Kolmogorov, 1968](#); [Chaitin, 1969](#)]. By measuring which structure is “simpler” (in terms of information entropy) and “more efficient” (in terms of running time), a suitable judgement can be made for choice of representation. We refer to this as “conceptual complexity.” More concretely, we leverage the objects and types determined by the present scope. We measure the total description length of the types relative to the declarative type-level logic of a pure type system (the representation we adopt, c.f. [Section 3.7 Pure type systems](#) and [Section 5.2 Representation and role](#)). We further measure the description length of the objects that are represented via those types relative to the computational substrate (whatever the pure type system exists atop of, e.g. lambda calculus as in [Section 3.5](#) or a term rewriting system as in [Section 3.6](#)). Running time is another factor of our measure which yields something like a description length of objects but specifically in terms of other objects (e.g. the length of object “plus 18 13” given the procedure object plus)¹⁷. See [Figure 5-12](#) for an illustra-

¹⁷ The reader may note that the halting problem makes this in calculable. However, this can be avoided by establishing a finite limit on computational cost associated with program execution, a common practice when systems compute on untrusted programs. If this finite limit is surpassed, the length may be ruled as

tion of these three complexity measures. The objects in scope may be provided by both episodic replay (the recall of real experience) and hallucination (objects that are the product of cognition), both of which are notoriously present during dreaming [Fosse et al., 2003]. These complexity measures for conceptual structures permits useful comparison between co-occurring conceptual structures — where “co-occurring” means that they exist in the same scope (such that the same objects constrain each conceptual structure). We hypothesize a function which, given these three complexity measures for each of a number of conceptual structures, determines what items should be written in terms of a particular structure.

$$\begin{aligned}
 \text{NAT} &= \{\text{zero}\} \cup \{\text{succ } n \mid n \in \text{NAT}\} \\
 \text{plus} &: \{ \text{plus} : \text{NAT} \rightarrow \text{NAT} \rightarrow \text{NAT} \\
 & \mid \forall a, b, c \in \text{NAT}. \text{plus } a (\text{plus } b c) = \text{plus } (\text{plus } a b) c \\
 & \wedge \forall a, b \in \text{NAT}. \text{plus } a b = \text{plus } b a \\
 & \wedge \forall a \in \text{NAT}. \text{plus } \text{zero } a = a \} \\
 & \text{(a)}
 \end{aligned}$$

$$\begin{aligned}
 \text{plus } \text{zero } a &= a \\
 \text{plus } (\text{succ } a) b &= \text{succ } (\text{plus } a b) \\
 & \text{(b)}
 \end{aligned}$$

$$\begin{aligned}
 [(3 + 4)] & \quad \text{plus } (\text{succ } (\text{succ } (\text{succ } \text{zero}))) (\text{succ } (\text{succ } (\text{succ } (\text{succ } \text{zero})))) \\
 [1 + (2 + 4)] & \rightarrow \text{succ } (\text{plus } (\text{succ } (\text{succ } \text{zero})) (\text{succ } (\text{succ } (\text{succ } (\text{succ } \text{zero})))))) \\
 [1 + 1 + (1 + 4)] & \rightarrow \text{succ } (\text{succ } (\text{plus } (\text{succ } \text{zero}) (\text{succ } (\text{succ } (\text{succ } (\text{succ } \text{zero})))))) \\
 [1 + 1 + 1 + (0 + 4)] & \rightarrow \text{succ } (\text{succ } (\text{succ } (\text{plus } \text{zero } (\text{succ } (\text{succ } (\text{succ } (\text{succ } \text{zero}))))))) \\
 [1 + 1 + 1 + 4] & \quad \rightarrow \text{succ } (\text{succ } (\text{succ } (\text{succ } (\text{succ } (\text{succ } (\text{succ } \text{zero})))))) \\
 & \text{(c)}
 \end{aligned}$$

Figure 5-12: Descriptions whose lengths determine three measures of conceptual complexity: (a) type-level, (b) object-level, and (c) runtime-level. If this examples co-occurs with a representation like that of Figure 5-7b, measurements of these complexities may demonstrate that it is an inferior representation and would be replaced. (Notably, addition is runtime-logarithmic in a digital representation whereas this incremental representation is runtime-linear.)

infinite thereby making the representation in question unworthy.

5.5 Discussion and Future Work

We present a computational formalism for conceptual role. However, there are ways that our exposition does not satisfy the essence of a scientific formalism. The approach presented in this chapter is in dire need of thorough theory and experimentation to justify the approach. The framework must be implemented at least in some significant part in order to evaluate any aspect of the approach. Most of the motivation of our approach comes from intuition and loosely-related empirical studies while our reliance on research particularly focused on computational models of cognition was somewhat lacking. We believe there is plenty that needs to be done for computational models of cognition.

An apparent step is to implement a language with a rich type system which has some of the procedures of [Section 5.3 Learning and conceptual change](#) built-in. However, many aspects of our approach can be modeled without this foundation. We believe that the best way to assess our approach is by breaking it down into smaller systems that can be modeled and experimented upon in isolation. Only once emergent properties of those underlying subsystems is determined can they be put together meaningfully, assuming they are shown to be fruitful.

Therefore the first thing that should be done is to break apart our theory into many independent models and to perform empirical computational and psychological studies with them. The consolidation for probabilistic programs may yield a promising project on its own — where some probabilistic models may be abstracted into conditional distributions induced by a deeper probabilistic model if such consolidation would be useful for storage and computation regarding relevant concepts. Similar may be said for our discussion in [Section 5.3.5 Inducing search procedures and probabilistic models](#): many of the necessary tools already exist, and the application of such techniques could extend to a variety of program induction tasks and machine learning problems. Especially relevant to cognitive science is the computational account of creating new ideas we discussed in [Section 5.3.4 Construction of concepts](#), which in conjunction with a computational account of conceptual change itself (particularly the conceptual complexity measure described in [Section 5.4 Perception, memory, and cognition](#)) provide a promising research path that should be explored.

Bibliography

- Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. Justin Kelly, 1996.
- Takuya Akiba, Kentaro Imajo, Hiroaki Iwami, Yoichi Iwata, Toshiki Kataoka, Naohiro Takahashi, Michał Moskal, and Nikhil Swamy. Calibrating research in program synthesis using 72,000 hours of programmer time. *Microsoft Research, Redmond, WA, USA, Technical Report*, 2013.
- Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.
- John R. Anderson. *The architecture of cognition*. Psychology Press, 1983.
- René Baillargeon. How do infants learn about the physical world? *Current Directions in Psychological Science*, 3(5):133–140, 1994.
- Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *CoRR*, 2017.
- Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.
- Albert-László Barabási et al. *Network science*. Cambridge university press, 2016.
- Henk Barendregt. Introduction to generalized type systems. *Journal of functional programming*, 1(2):125–154, 1991.
- Henk P Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.
- Marc Bezem, Jan Willem Klop, and Roel de Vrijer. *Term rewriting systems*, volume 55. Cambridge University Press, 2003.
- Ned Block. Advertisement for a semantics for psychology. 10:615–678, 1986.
- Mikhail M. Bongard. *Pattern Recognition*. Spartan Books, 1970.
- George Boole. *An investigation of the laws of thought: on which are founded the mathematical theories of logic and probabilities*. Dover Publications, 1854.
- Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

- Ingo Brigandt. Conceptual role semantics, the theory theory, and conceptual change. 2004.
- Andres Campero, Aldo Pareja, Tim Klinger, Joshua B. Tenenbaum, and Sebastian Riedel. Theory learning and logical rule induction with neural theorem proving. 2018.
- Susan Carey. Conceptual change in childhood. 1985.
- Susan Carey. *The Origin of Concepts*. Oxford University Press, 2009.
- Susan Carey. Why theories of concepts should not ignore the problem of acquisition. In Morgolis and Lawrence, editors, *The Conceptual Mind: New Directions in the Study of Concepts*. MIT Press, 2015.
- Gregory J. Chaitin. On the simplicity and speed of programs for computing infinite sets of natural numbers. *J. ACM*, 16(3):407–422, Jul 1969. ISSN 0004-5411.
- Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 121–130. ACM, 2011.
- Xinyun Chen, Chang Liu, and Dawn Song. Towards synthesizing complex programs from input-output examples. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.
- Adam Chlipala. Certified programming with dependent types, 2011.
- 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.
- Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.
- David Christiansen and Edwin Brady. Elaborator reflection: extending idris in idris. *ACM SIGPLAN Notices*, 51(9):284–297, 2016.
- Alonzo Church. A note on the entscheidungsproblem. *The journal of symbolic logic*, 1(1):40–41, 1936.
- Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.
- Alonzo Church. *The calculi of lambda-conversion*. Number 6 in Annals of Mathematical Studies. Princeton University Press, 1941.
- Aaron Clauset, Cosma Rohilla Shalizi, and Mark E. J. Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.
- Trevor Cohn, Phil Blunsom, and Sharon Goldwater. Inducing tree-substitution grammars. *JMLR*, 11:3053–3096, 2010.
- Thierry Coquand. *An analysis of Girard’s paradox*. PhD thesis, INRIA, 1986.

- Thierry Coquand and Gérard Huet. *The calculus of constructions*. PhD thesis, INRIA, 1986.
- Haskell B. Curry. *Combinatory logic*. 1958.
- 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.
- Peter Dayan, Geoffrey E Hinton, Radford M. Neal, and Richard S. Zemel. The helmholtz machine. *Neural computation*, 7(5):889–904, 1995.
- Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.
- Eyal Dechter, Jonathan Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. Bootstrap learning via modular concept discovery. In *IJCAI*, 2013.
- Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural program meta-induction. In *NIPS*, 2017a.
- Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdelrahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *ICML*, 2017b.
- Yadin Dudai, Avi Karni, and Jan Born. The consolidation and transformation of memory. *Neuron*, 88(1):20 – 32, 2015. ISSN 0896-6273. doi: <https://doi.org/10.1016/j.neuron.2015.09.004>. URL <http://www.sciencedirect.com/science/article/pii/S0896627315007618>.
- Kevin M. Ellis, Armando Solar-Lezama, and Joshua B. Tenenbaum. Unsupervised learning by program synthesis. In *Advances in neural information processing systems*, pages 973–981, 2015.
- Kevin M. Ellis, Armando Solar-Lezama, and Joshua B. Tenenbaum. Sampling for bayesian program learning. In *Advances in Neural Information Processing Systems*, pages 1297–1305, 2016.
- Kevin M. Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B. Tenenbaum. Learning to infer graphics programs from hand-drawn images. *arXiv preprint arXiv:1707.09627*, 2017.
- Kevin M. Ellis, Lucas E. Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Joshua B. Tenenbaum. Learning libraries of subroutines for neurally-guided bayesian program learning. 2018. In submission.
- Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61:1–64, 2018.
- Scott E. Fahlman and Geoffrey E. Hinton. Connectionist architectures for artificial intelligence. *Computer;(United States)*, 20(1), 1987.

- David Fass and Jacob Feldman. Categorization under complexity: A unified mdl account of human learning of regular and irregular categories. In *Advances in neural information processing systems*, pages 35–42, 2003.
- Jacob Feldman. Minimization of boolean complexity in human concept learning. *Nature*, 407(6804):630, 2000.
- John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, volume 50, pages 229–239. ACM, 2015.
- Hartry H. Field. Logic, meaning, and conceptual role. *The Journal of Philosophy*, 74(7): 379–409, 1977.
- Jerry A. Fodor. *The language of thought*, volume 5. Harvard University Press, 1975.
- Jerry A. Fodor and Zenon W. Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1-2):3–71, 1988.
- Magdalena J Fosse, Roar Fosse, J Allan Hobson, and Robert J Stickgold. Dreaming and episodic memory: a functional dissociation? *Journal of cognitive neuroscience*, 15(1): 1–9, 2003.
- Michael C. Frank, Noah D. Goodman, and Joshua B. Tenenbaum. Using speakers’ referential intentions to model early cross-situational word learning. *Psychological science*, 20(5): 578–585, 2009.
- Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. *ACM SIGPLAN Notices*, 51(1):802–815, 2016.
- Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016.
- Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1):176–210, 1935.
- Jan Herman Geuvers. *Logics and type systems*. PhD thesis, Radboud University Nijmegen, 1993.
- Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de larithmétique dordre supérieur*. PhD thesis, Université Paris VII, 1972.
- Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.
- Noah D. Goodman, Chris L. Baker, Elizabeth Baraff Bonawitz, Vikash K. Mansinghka, Alison Gopnik, Henry Wellman, Laura Schulz, and Joshua B. Tenenbaum. Intuitive theories of mind: A rational approach to false belief. In *Proceedings of the twenty-eighth annual conference of the cognitive science society*, volume 6. Cognitive Science Society, 2006.

- Noah D. Goodman, Joshua B. Tenenbaum, Jacob Feldman, and Thomas L. Griffiths. A rational analysis of rule-based concept learning. *Cognitive science*, 32(1):108–154, 2008.
- Noah D. Goodman, Tomer D. Ullman, and Joshua B. Tenenbaum. Learning a theory of causality. *Psychological review*, 118(1):110, 2011.
- Noah D. Goodman, Joshua B. Tenenbaum, and T. Gerstenberg. Concepts in a probabilistic language of thought. In Morgolis and Lawrence, editors, *The Conceptual Mind: New Directions in the Study of Concepts*. MIT Press, 2015.
- Alison Gopnik and Andrew N Meltzoff. *Words, Thoughts, and Theories*. MIT Press, Cambridge, MA, 1997.
- Alison Gopnik and Laura Schulz. Mechanisms of theory formation in young children. *Trends in cognitive sciences*, 8(8):371–377, 2004.
- Alison Gopnik and Henry M. Wellman. Reconstructing constructivism: Causal models, bayesian learning mechanisms, and the theory theory. *Psychological bulletin*, 138(6):1085, 2012.
- Johan Georg Granström. *Treatise on intuitionistic type theory*, volume 22. Springer Science & Business Media, 2011.
- Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.
- 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.
- Thomas L. Griffiths and Zoubin Ghahramani. The indian buffet process: An introduction and review. *Journal of Machine Learning Research*, 12(Apr):1185–1224, 2011.
- Thomas L. Griffiths and Joshua B. Tenenbaum. Theory-based causal induction. *Psychological review*, 116(4):661, 2009.
- Frederic Gruau. Neural network synthesis using cellular encoding and the genetic algorithm. 1994.
- Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.
- Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.
- Hyowon Gweon, Joshua B. Tenenbaum, and Laura E. Schulz. Infants consider both the sample and the sampling process in inductive generalization. *Proceedings of the National Academy of Sciences*, 107(20):9066–9071, 2010.
- Stephen Muggleton H., Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Machine Learning*, 100:49–73, 2013.

- Gilbert Harman et al. Conceptual role semantics. *Notre Dame Journal of Formal Logic*, 23 (2):242–256, 1982.
- Robert John Henderson. *Cumulative learning in the lambda calculus*. PhD thesis, Imperial College London, 2013.
- Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.
- Geoffrey E. Hinton and Ruslan R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 2006.
- 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.
- Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- Martin Hofmann and Emanuel Kitzelmann. I/o guided detection of list catamorphisms: towards problem specific use of program templates in ip. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 93–100. ACM, 2010.
- Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 1979.
- Douglas R. Hofstadter and Melanie Mitchell. Fluid concepts and creative analogies. chapter The copycat project: A model of mental fluidity and analogy-making, pages 205–267. Basic Books, 1995.
- Antonius J.C. Hurkens. A simplification of girard’s paradox. In *International Conference on Typed Lambda Calculi and Applications*, pages 266–278. Springer, 1995.
- Ramon Ferrer i Cancho and Ricard V. Solé. Least effort and the origins of scaling in human language. *Proceedings of the National Academy of Sciences*, 100(3):788–791, 2003.
- Ramon Ferrer i Cancho and Richard V. Solé. The small world of human language. *Proceedings of the Royal Society of London B: Biological Sciences*, 268(1482):2261–2265, 2001.
- Hawoong Jeong, Sean P. Mason, Albert-László Barabási, and Zoltan N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41, 2001.
- Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 215–224. ACM, 2010.
- Mark Johnson, Thomas L. Griffiths, and Sharon Goldwater. Adaptor grammars: A framework for specifying compositional nonparametric bayesian models. In *Advances in neural information processing systems*, pages 641–648, 2007.
- Manu Jose and Rupak Majumdar. Bug-assist: assisting fault localization in ansi-c programs. In *International conference on computer aided verification*, pages 504–509. Springer, 2011.

- Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in neural information processing systems*, pages 190–198, 2015.
- 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.
- Susumu Katayama. Systematic search for lambda expressions. *Trends in functional programming*, 6:111–126, 2005.
- Charles Kemp. Exploring the conceptual universe. *Psychological review*, 119 4:685–722, 2012.
- Charles Kemp and Joshua B. Tenenbaum. Structured statistical models of inductive reasoning. *Psychological review*, 116(1):20, 2009.
- Charles Kemp, Noah D. Goodman, and Joshua B. Tenenbaum. Learning and using relational theories. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 753–760. Curran Associates, Inc., 2008a.
- Charles Kemp, Noah D. Goodman, and Joshua B. Tenenbaum. Theory acquisition and the language of thought. Cognitive Science Society, 2008b.
- Stephen Cole Kleene. *Introduction to metamathematics*. Ishi Press International, 1952. Reprinted 2009.
- Jan Willem Klop. Combinatory reduction systems. 1980.
- Jan Willem Klop, Vincent Van Oostrom, and Femke Van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical computer science*, 121(1-2):279–308, 1993.
- Andrei Nikolaevich Kolmogorov. Three approaches to the quantitative definition of information. *International journal of computer mathematics*, 2(1-4):157–168, 1968.
- Andrej N. Kolmogorov. Zur deutung der intuitionistischen logik. *Mathematische Zeitschrift*, 35(1):58–65, 1932.
- John R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.
- Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago press, 1965.
- Tejas D. Kulkarni, William F. Whitney, Pushmeet Kohli, and Joshua B. Tenenbaum. Deep convolutional inverse graphics network. In *Advances in neural information processing systems*, pages 2539–2547, 2015.
- Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

- J.D. Lafferty. *A Derivation of the Inside-outside Algorithm from the EM Algorithm*. Research report. IBM T.J. Watson Research Center, 2000.
- Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.
- 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*, 2017.
- Tessa Lau. *Programming by demonstration: a machine learning approach*. PhD thesis, University of Washington, 2001.
- Tuan Anh Le, Atim Güne Baydin, and Frank Wood. Inference Compilation and Universal Probabilistic Programming. In *AISTATS*, 2017.
- Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.
- Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19:189–223, 2011.
- Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *ICML*, pages 639–646, 2010.
- Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenenbaum, and Stephen H. Muggleton. Bias reformulation for one-shot function induction. 2014.
- Vikash Kumar Mansinghka. *Natively probabilistic computation*. PhD thesis, Massachusetts Institute of Technology, Department of Brain and Cognitive Sciences, 2009.
- David Marr. *Vision*. W. H. Freeman, San Francisco, CA, 1982.
- James L. McClelland. A connectionist perspective on knowledge and development. 1995.
- James L. McClelland and Karalyn Patterson. Rules or connections in past-tense inflections: What does the evidence rule out? *Trends in cognitive sciences*, 6(11):465–472, 2002.
- James L. McClelland, Matthew M. Botvinick, David C. Noelle, David C. Plaut, Timothy T. Rogers, Mark S. Seidenberg, and Linda B. Smith. Letting structure emerge: connectionist and dynamical systems approaches to cognition. *Trends in cognitive sciences*, 14(8):348–356, 2010.
- Josh H. McDermott and Eero P. Simoncelli. Sound texture perception via statistics of the auditory periphery: evidence from sound synthesis. *Neuron*, 71(5):926–940, 2011.
- 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.
- Robert King Merton. *Social theory and social structure*. Simon and Schuster, 1968.
- Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

- Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.
- Marvin Minsky. *Society of mind*. Simon and Schuster, 1988.
- Stephen Muggleton and Wray Buntine. Machine invention of first-order predicates by inverting resolution. In John Laird, editor, *Machine Learning Proceedings 1988*, pages 339 – 352. Morgan Kaufmann, San Francisco (CA), 1988. ISBN 978-0-934613-64-4. doi: <https://doi.org/10.1016/B978-0-934613-64-4.50040-2>.
- Stephen H. Muggleton. Inductive logic programming. *New generation computing*, 8(4): 295–318, 1991.
- Stephen H. Muggleton. Inverse entailment and prolog. *New Generation Computing*, 13(3): 245–286, Dec 1995. ISSN 1882-7055. doi: 10.1007/BF03037227.
- Stephen H. Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine learning*, 94:25–49, 2014.
- 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.
- Gregory L. Murphy and Douglas L. Medin. The role of theories in conceptual coherence. *Psychological review*, 92(3):289, 1985.
- Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*, 2015.
- Allen Newell. *Unified theories of cognition*. Harvard University Press, 1994.
- Timothy J. O’Donnell. *Productivity and Reuse in Language: A Theory of Linguistic Computation and Storage*. MIT Press, 2015.
- Roland Olsson. *Inductive functional programming using incremental program transformation and execution of logic programs by iterative-deeping A* SLD-tree search*. PhD thesis, University of Oslo, 1994.
- Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, volume 50, pages 619–630. ACM, 2015.
- Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.
- Jean Piaget. *La construction du réel chez l’enfant*. 1937.
- Steven T. Piantadosi. The computational origin of representation and conceptual change. 2016.
- Steven T Piantadosi and Robert A Jacobs. Four problems solved by the probabilistic language of thought. *Current Directions in Psychological Science*, 25(1):54–59, 2016.

- Steven T. Piantadosi, Joshua B. Tenenbaum, and Noah D. Goodman. Bootstrapping in a language of thought: A formal model of numerical concept learning. *Cognition*, 123(2): 199–217, 2012.
- Steven T. Piantadosi, Joshua B. Tenenbaum, and Noah D. Goodman. The logical primitives of thought: Empirical foundations for compositional cognitive models. *Psychological review*, 123(4):392, 2016.
- Benjamin C. Pierce. *Types and programming languages*. MIT press, 2002.
- Steven Pinker and Michael T. Ullman. The past and future of the past tense. *Trends in cognitive sciences*, 6(11):456–463, 2002.
- Gordon D. Plotkin. A note on inductive generalization. *Machine intelligence*, 5(1):153–163, 1970.
- Riccardo Poli, William B. Langdon, Nicholas F. McPhee, and John R. Koza. *A field guide to genetic programming*. Lulu. com, 2008.
- Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.
- Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. 50(10):107–126, 2015.
- Ben Poole, Subhaneil Lahiri, Maithra Raghu, Jascha Sohl-Dickstein, and Surya Ganguli. Exponential expressivity in deep neural networks through transient chaos. In *NIPS*, 2016.
- Derek J. De Solla Price. Networks of scientific papers. *Science*, pages 510–515, 1965.
- Zenon W. Pylyshyn. Computation and cognition: Issues in the foundations of cognitive science. *Behavioral and Brain Sciences*, 3(1):111–132, 1980.
- Luc De Raedt. *Logical and Relational Learning*. 2008.
- Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.
- John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.
- Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. In *Advances in Neural Information Processing Systems*, pages 3788–3800, 2017.
- Timothy T. Rogers and James L. McClelland. *Semantic cognition: A parallel distributed processing approach*. MIT press, 2004.
- Josh Rule, Eric Schulz, Steven T. Piantadosi, and Joshua B. Tenenbaum. Learning list concepts through program induction. In *Proceedings of the fortieth annual conference of the cognitive science society*. Cognitive Science Society, 2018.
- David E. Rumelhart and James L. McClelland. Parallel distributed processing: explorations in the microstructure of cognition. volume 1. foundations. 1986.

- Bertrand Russell. *Appendix B: The Doctrine of Types*, pages 523–528. Cambridge University Press, 1903.
- Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 305–316. ACM, 2013.
- Ute Schmid and Emanuel Kitzelmann. Inductive rule learning on the knowledge level. *Cognitive Systems Research*, 12(3-4):237–248, 2011.
- Jürgen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.
- Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61: 85–117, 2015.
- Moses Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische annalen*, 92(3-4):305–316, 1924.
- Laura Schulz. Finding new facts; thinking new thoughts. In *Advances in child development and behavior*, volume 43, pages 269–294. Elsevier, 2012a.
- Laura E Schulz. The origins of inquiry: Inductive inference and exploration in early childhood. *Trends in cognitive sciences*, 16(7):382–389, 2012b.
- Laura E. Schulz, Noah D. Goodman, Joshua B. Tenenbaum, and Adrianna C. Jenkins. Going beyond the evidence: Abstract laws and preschoolers responses to anomalous data. *Cognition*, 109(2):211–223, 2008.
- Thomas Serre, Aude Oliva, and Tomaso Poggio. A feedforward architecture accounts for rapid categorization. *Proceedings of the national academy of sciences*, 104(15):6424–6429, 2007.
- Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.
- Armando Solar-Lezama. *Program synthesis by sketching*. PhD thesis, University of California, Berkeley, 2008.
- Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM Sigplan Notices*, 41(11):404–415, 2006.
- Ray J Solomonoff. A formal theory of inductive inference. part i. *Information and control*, 7(1):1–22, 1964.
- Ray J Solomonoff. A system for incremental learning based on algorithmic probability. Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition, 1989.
- Lee Spector, Jon Klein, and Maarten Keijzer. The push3 execution stack and the evolution of control. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1689–1696. ACM, 2005.
- Elizabeth S. Spelke. Nativism, empiricism, and the origins of knowledge. *Infant Behavior and Development*, 21(2):181–200, 1998.

- Elizabeth S. Spelke and Katherine D. Kinzler. Core knowledge. *Developmental science*, 10(1):89–96, 2007.
- Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.
- Joshua B. Tenenbaum, Thomas L. Griffiths, and Charles Kemp. Theory-based bayesian models of inductive learning and reasoning. *Trends in cognitive sciences*, 10(7):309–318, 2006.
- Joshua B. Tenenbaum, Thomas L. Griffiths, and Sourabh Niyogi. Intuitive theories as grammars for causal inference. *Causal learning: Psychology, philosophy, and computation*, pages 301–322, 2007.
- Joshua B. Tenenbaum, Charles Kemp, Thomas L. Griffiths, and Noah D. Goodman. How to grow a mind: Statistics, structure, and abstraction. *science*, 331(6022):1279–1285, 2011.
- Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2013*, pages 135–152, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509586.
- Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. doi: 10.1112/plms/s2-42.1.230.
- Alan Mathison Turing. Computing machinery and intelligence. *Mind*, 59(236):433460, 1950. doi: 10.1093/mind/LIX.236.433.
- Tomer D. Ullman. *On the nature and origin of intuitive theories: learning, physics and psychology*. PhD thesis, Massachusetts Institute of Technology, 2015.
- Tomer D. Ullman, Noah D. Goodman, and Joshua B. Tenenbaum. Theory learning as stochastic search in the language of thought. *Cognitive Development*, 27(4):455–480, 2012.
- Alfred North Whitehead and Bertrand Russell. *Principia mathematica*, volume 2. University Press, 1912.
- James Woodward. *Making things happen: A theory of causal explanation*. New York: Oxford university press, 2003.
- Fan Yang, Zhilin Yang, and William W Cohen. Differentiable learning of logical rules for knowledge base reasoning. In *Advances in Neural Information Processing Systems*, pages 2319–2328, 2017.
- Ilker Yildirim and Robert A Jacobs. Learning multisensory representations for auditory-visual transfer of sequence category knowledge: a probabilistic language of thought approach. *Psychonomic bulletin & review*, 22(3):673–686, 2015.
- G. Udny Yule et al. A mathematical theory of evolution, based on the conclusions of dr. j. c. willis, f.r.s. *Phil. Trans. R. Soc. Lond. B*, 213(402-410):21–87, 1925.

George Kingsley Zipf. *Human behavior and the principle of least effort: An introduction to human ecology*. Addison-Wesley, 1949.