Learning Programs: A Hierarchical Bayesian Approach

Percy Liang PLIANG@CS.BERKELEY.EDU
Computer Science Division, University of California, Berkeley, CA 94720, USA

Michael I. Jordan JORDAN@CS.BERKELEY.EDU
Computer Science Division and Department of Statistics, University of California, Berkeley, CA 94720, USA

Dan Klein KLEIN@CS.BERKELEY.EDU
Computer Science Division, University of California, Berkeley, CA 94720, USA

Abstract

We are interested in learning programs for multiple related tasks given only a few training examples per task. Since the program for a single task is underdetermined by its data, we introduce a nonparametric hierarchical Bayesian prior over programs which shares statistical strength across multiple tasks. The key challenge is to parametrize this multi-task sharing. For this, we introduce a new representation of programs based on combinatory logic and provide an MCMC algorithm that can perform safe program transformations on this representation to reveal shared inter-program substructures.

1. Introduction

A general focus in machine learning is the estimation of functions from examples. Most of the literature focuses on real-valued functions, which have proven useful in many classification and regression applications. This paper explores the learning of a different but also important class of functions—those specified most naturally by computer programs.

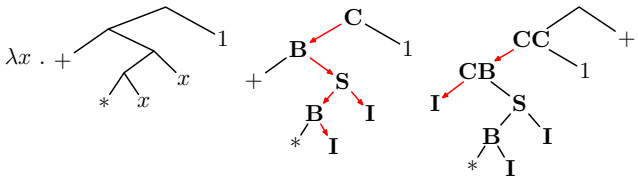
To motivate this direction of exploration, consider programming by demonstration (PBD) (Cypher, 1993). In PBD, a human demonstrates a repetitive task in a few contexts; the machine then learns to perform the task in new contexts. An example we consider in this paper is text editing (Lau et al., 2003). Suppose a user wishes to italicize all occurrences of the word *statistics*. If the user demonstrates italicizing two occurrences of *statistics*, can we generalize to the others? The solution to this italicization task can be represented compactly by a program: (1) move the cursor to the next occurrence of *statistics*, (2) insert `<i>`, (3) move to the end of the word, and (4) insert `</i>`.

From a learning perspective, the main difficulty with PBD is that it is only reasonable to expect one or two training examples from the user. Thus the program is underdetermined by the data: Although the user moved to the beginning of the word *statistics*, an alternate predicate might be after a space. Clearly, some sort of prior or complexity penalty over programs is necessary to provide an inductive bias. For real-valued functions, many penalties based on smoothness, norm, and dimension have been studied in detail for decades. For programs, what is a good measure of complexity (prior) that facilitates learning?

We often want to perform many related tasks (e.g., in text editing, another task might be to italicize the word *logic*). In this multi-task setting, it is natural to define a hierarchical prior (a joint measure of complexity) over multiple programs, which allows the sharing of statistical strength through the joint prior.

The key conceptual question is how to allow sharing between programs. Here, we can take inspiration from good software engineering principles: Programs should be structured modularly so as to enable code reuse. However, it is difficult to implement this intuition since programs typically have many internal dependencies; therefore, transforming programs safely into a modular form for statistical sharing without disrupting the program semantics requires care. Our solution is to build on *combinatory logic* (Schönfinkel, 1924), a simple and elegant formalism for building complex programs via composition of simpler subprograms. Its simplicity makes it conducive to probabilistic modeling.

Appearing in *Proceedings of the 26th International Conference on Machine Learning*, Haifa, Israel, 2010. Copyright 2010 by the author(s)/owner(s).



(a) lambda calculus (b) no variables (c) refactored

Figure 1. Three equivalent representations of the function $x \mapsto x^2 + 1$. (b) is variable-free; The routers at internal nodes encode how argument x should be routed down (as depicted by the arrows). (c) allows + to be refactored out; higher-order routers keep track of its original placement.

Section 2 presents our representation of programs based on combinatory logic. We then present our non-parametric hierarchical Bayesian prior over multiple programs (Section 3) and an MCMC inference algorithm (Section 4). Finally, Section 5 shows the merits of our approach on text editing.

2. Program Representation

The first order of business is to find a suitable language for expressing programs. Recall that we want a representation that highlights the modularity of the computation expressed. To do this, we develop a new version of combinatory logic. We first introduce it intuitively with lambda calculus as a reference point (Section 2.1) and then define it formally (Section 2.2).

2.1. Intuitive Description

Lambda calculus is a language for expressing computation in a functional paradigm[1] (see Hankin (2004) for an introduction). As a running example, consider the simple lambda calculus program that computes the function $x \mapsto x^2 + 1$ (Figure 1(a)). It will be useful to think of programs as binary trees where each node denotes the result of applying the left subtree to the right subtree. Functions are curried.

One issue with lambda calculus is long-range dependencies between places where a variable is bound (λx) and places where it is used (x). This non-locality necessitates the maintaining of an environment (mapping from variable names to values), making program transformations cumbersome.

To motivate combinatory logic, let us try to transform the function in Figure 1(a) into a variable-free form that preserves the information content: Replace the variable x with $\mathbf{I}$ and label internal nodes of the tree with a *router* (for now, one of $\mathbf{B}, \mathbf{C}, \mathbf{S}$) depending on whether x appeared in the right subtree, left subtree, or both, respectively. The result is Figure 1(b). To apply this function on an argument, we start the argument at the root of the transformed tree. The router at each node determines to which subtrees the argument should be sent. When the argument reaches $\mathbf{I}$, it replaces $\mathbf{I}$.

One significance of the variable-free formalism is that we have eliminated the distinction between program and subprogram. Each subtree is now a valid standalone program, and thus a candidate for multi-task sharing. For example, $(\mathbf{S}\ (\mathbf{B}\ *\ \mathbf{I})\ \mathbf{I})$ denotes the square function, which could be useful elsewhere.

However, sometimes the desired unit of sharing does not appear as a subtree. For example, functions $x \mapsto x^2 + 1$ and $x \mapsto x^2 - 1$ have identical trees except for one leaf (which is + or −). To address this, we can pull the + (−) leaf to the top, augmenting the routers along the path. Figure 1(c) is the result of this refactoring operation. The left subtree of the root now denotes a higher-order function, which when applied to +, produces the function $x \mapsto x^2 + 1$. Refactoring creates new sharable modular subprograms, analogous to how a good programmer might.

2.2. Formal definition

Having provided some intuition, we now define our modified version of combinatory logic formally. Combinatory logic, invented by Schönfinkel in 1924 and further developed by Curry, is a variable-free formalism for expressing computation which actually predates its popular rival, lambda calculus. It has been mainly used in the study of computability and in the low-level compilation of functional languages.

Let $\mathcal{B}$ be a set of symbols called *primitive combinators*, known as the basis. A *combinator* is a binary tree whose leaves are *primitive combinators*. We write $(x\ y)$ to denote the tree with left and right subtrees x and y, and write $(x\ y\ z)$ for $((x\ y)\ z)$ (currying). We use an *interpretation function* $[\![\cdot]\!]$ to map each combinator (a syntactic expression) to its semantic *denotation*. For example, $[\![(+\ 1\ 1)]\!] = 2$ and $[\![(+\ 1)]\!]$ is the function $x \mapsto x + 1$. Given the denotation of primitive combinators ($[\![x]\!]$ for all $x \in \mathcal{B}$), we can define the denotation of all other combinators recursively: $[\![(x\ y)]\!]$ is the result of applying function $[\![x]\!]$ to argument $[\![y]\!]$.

A main theoretical result in combinatory logic is that a basis consisting of just two elements (called $\mathbf{S}$ and $\mathbf{K}$) suffices to build all computable functions. However, a

[1] We prefer functional languages to procedural ones because side effects complicate reasoning about program behavior.

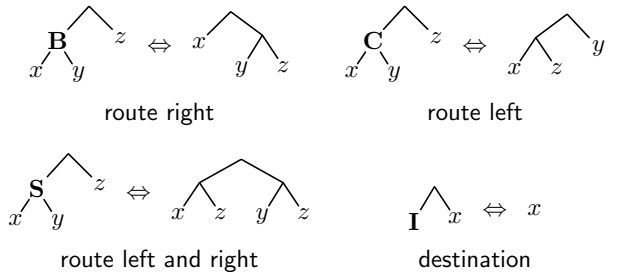


Figure 2. Equivalences defined by the first-order routers $\mathbf{B}, \mathbf{C}, \mathbf{S}, \mathbf{I}$, which hold for any combinators x, y, z. These are also among the transformations used during inference (Section 4.2.2).

major disadvantage with this basis is that the resulting combinators become quite large and cumbersome. To strike a balance between minimality and practical usability, we make two modifications to **SK** combinatory logic: (1) we introduce higher-order combinators to capture the intuition of routing; and (2) we place these combinators at internal nodes.

Define a *router* to be a primitive combinator represented by a finite sequence of elements from $\{\mathbf{B}, \mathbf{C}, \mathbf{S}\}$. Let $\mathcal{R}_k = \{\mathbf{B}, \mathbf{C}, \mathbf{S}\}^k$ (the set of k-th order routers), $\mathcal{R}_{\leq k} = \cup_{j=0}^{k} \mathcal{R}_j$ (routers up to order k), and $\mathcal{R} = \cup_{j=0}^{\infty} \mathcal{R}_j$ (all routers). For a router $\mathbf{r} \in \mathcal{R}$, its behavior is given by

$$(\mathbf{r}\ x\ y\ z_1 \cdots z_{|\mathbf{r}|}) = ((x\ z_{i_1} \cdots z_{i_n})\ (y\ z_{j_1} \cdots z_{j_m})), \quad (1)$$

where $i_1 < \cdots < i_n$ are indices i such that $r_i \in \{\mathbf{C}, \mathbf{S}\}$ and $j_1 < \cdots < j_m$ are indices j such that $r_j \in \{\mathbf{B}, \mathbf{S}\}$. Routers generalize the idea of function application: $\mathbf{r}$ first applies x and y to the appropriate subset of arguments $z_1, \ldots, z_{|\mathbf{r}|}$ to get x' and y', and then applies x' to y'.

While routers are just combinators, they play a vital structural role in a program, so we will treat them specially. Define a *combinator with routing* to be a binary tree where each internal node is labeled with a router. We write $(\mathbf{r}\ x\ y)$ for a combinator with router $\mathbf{r}$, left subtree x, and right subtree y; we simply write $(x\ y)$ if $|\mathbf{r}| = 0$. Figure 2 illustrates combinators with first-order routers and their behavior.

2.3. Types

The final piece of our representation is types, which allows us to prohibit programs such as $(3\ \mathbf{I})$, invalid because an integer cannot be applied to a function. We will work with a monomorphic type system: Let $\mathcal{T}_0$ denote the set of *base types* (e.g., $\mathcal{T}_0 = \{\text{int}, \text{bool}\}$). Let $\mathcal{T}$ denote the (infinite) set of all types, defined to be the smallest set such that $\mathcal{T}_0 \subset \mathcal{T}$ and if $t_1, t_2 \in \mathcal{T}$, $t_1 \to t_2 \in \mathcal{T}$. The arrow operator is right associative, meaning $t_1 \to t_2 \to t_3 \equiv t_1 \to (t_2 \to t_3)$. For each type t, let $\mathcal{B}_t$ be the set of primitive combinators of that type. In the arithmetic domain, we have $\mathcal{B}_{\text{int}} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, $\mathcal{B}_{\text{int}\to\text{int}\to\text{int}} = \{+, -, *, /\}$, $\mathcal{B}_{\text{int}\to\text{int}\to\text{bool}} = \{<, >, =\}$, and $\mathcal{B}_{\text{bool}\to\text{int}\to\text{int}\to\text{int}} = \{\text{if}\}$.

Define $\mathcal{C}_t$, the set all combinators of type $t \in \mathcal{T}$, as follows. Write $t = a_1 \cdots \to \cdots a_{k(t)} \to b$, where b is a base type and $k(t)$ is the *arity* of type t. Define

$$T_{\mathbf{C}}(t, \mathbf{r}, s) \stackrel{\text{def}}{=} a_{i_1} \cdots \to \cdots a_{i_n} \to s \to b, \quad (2)$$

$$T_{\mathbf{B}}(t, \mathbf{r}, s) \stackrel{\text{def}}{=} a_{j_1} \cdots \to \cdots a_{j_m} \to s, \quad (3)$$

where $i_1 < \cdots < i_n$ are the indices i corresponding to $r_i \in \{\mathbf{C}, \mathbf{S}\}$, and $j_1 < \cdots < j_m$ are the indices j corresponding to $r_j \in \{\mathbf{B}, \mathbf{S}\}$. The idea is that for any $(\mathbf{r}, s)$, if x has type $T_{\mathbf{C}}(t, \mathbf{r}, s)$ and y has type $T_{\mathbf{B}}(t, \mathbf{r}, s)$, then $(\mathbf{r}\ x\ y)$ has type t. We define $\{\mathcal{C}_t\}$ to be the smallest sets that satisfy the following fixed point equations:

$$\mathcal{C}_t = \mathcal{B}_t \cup \bigcup_{\mathbf{r} \in \mathcal{R}_{\leq k(t)}, s \in \mathcal{T}} \{\mathbf{r}\} \times \mathcal{C}_{T_{\mathbf{C}}(t,\mathbf{r},s)} \times \mathcal{C}_{T_{\mathbf{B}}(t,\mathbf{r},s)}, \forall t \in \mathcal{T} \quad (4)$$

Let $\mathcal{C} \stackrel{\text{def}}{=} \cup_{t \in \mathcal{T}} \mathcal{C}_t$ be all well-typed combinators. This completes the description of our simply-typed routing-based combinatory logic. Its variable-free nature gives us a fully compositional representation of programs, which will exploit in the sequel.

3. Probabilistic Model

Our goal is to define a distribution over combinators $\mathcal{C}_t$ for each type $t \in \mathcal{T}$. We start with a simple PCFG model (Section 3.1), and then develop a model based on adaptor grammars (Section 3.2). Section 3.3 shows how we use this model for multi-task learning.

3.1. Probabilistic Context-Free Grammars

Given that $\mathcal{C}_t$ consists of binary trees, a starting point is to model them using a probabilistic context-free grammar (PCFG). The parameters of the PCFG model are as follows: λ_0, the probability of generating a terminal; $p_0^{\mathcal{B}}(z \mid t)$, a distribution over primitive combinators (including $\mathbf{I}$ if $t = a \to a$ for some $a \in \mathcal{T}$); $p_0^{\mathcal{R}}(\mathbf{r} \mid k)$, a distribution over routers of order k; and $p_0^{\mathcal{T}}(t)$ a distribution over types. Figure 3 describes the generative process: a call to GenIndep(t) returns a combinator of type t by either generating a primitive combinator or recursively generating a non-primitive combinator.

GenIndep(t):
  With probability λ_0: [primitive]
    Return a primitive $z \in \mathcal{B}_t$ according to $p_0^{\mathcal{B}}(z \mid t)$
  Else: [non-primitive]
    Generate a router $\mathbf{r} \in \mathcal{R}_{\leq k(t)}$ from $p_0^{\mathcal{R}}(\mathbf{r} \mid k(t))$
    Generate an intermediate type $s \in \mathcal{T}$ from $p_0^{\mathcal{T}}(s)$
    Recursively sample x from GenIndep($T_{\mathbf{C}}(t, \mathbf{r}, s)$)
    Recursively sample y from GenIndep($T_{\mathbf{B}}(t, \mathbf{r}, s)$)
    Return combinator $(\mathbf{r}\ x\ y)$

Figure 3. A distribution over programs based on a probabilistic context-free grammars.

for $t \in \mathcal{T}$: $C_t \leftarrow [\,]$ [initialize caches]
Definitions:
  N_t: number of distinct elements in C_t
  M_z: number of times z occurs in $C_{t(z)}$
  Return* z = add z to $C_{t(z)}$ and return z

GenCache(t):
  With probability $\frac{\alpha_0 + N_t d}{\alpha_0 + |C_t|}$: [construct]
    With probability λ_0: [primitive]
      Return* a primitive $z \in \mathcal{B}_t$ according to $p_0^{\mathcal{B}}(z \mid t)$
    Else: [non-primitive]
      Add a placeholder $z^\dagger$ to C_t
      Generate a router $\mathbf{r} \in \mathcal{R}_{\leq k(t)}$ from $p_0^{\mathcal{R}}(\mathbf{r} \mid k(t))$
      Generate an intermediate type $s \in \mathcal{T}$ from $p_0^{\mathcal{T}}(s)$
      Recursively sample x from GenCache($T_{\mathbf{C}}(t, \mathbf{r}, s)$)
      Recursively sample y from GenCache($T_{\mathbf{B}}(t, \mathbf{r}, s)$)
      Remove $z^\dagger$ from C_t
      Return* combinator $(\mathbf{r}\ x\ y)$
  Else: [fetch]
    Return* $z \in C_t$ with probability $\frac{M_z - d}{|C_t| - N_t d}$

Figure 4. Specifies a distribution over programs based on adaptor grammars. This model allows sharing of subprograms via caches.

GenIndep fully exploits the compositional structure of combinators, aligning it with conditional independence in the statistical world. Though attractive computationally, GenIndep's assumption of conditional independence—that the function and argument are independent conditioned on their types—is too strong, and we will weaken this assumption in the next model.

3.2. Adaptor Grammars

We create a richer model by leveraging two statistical ideas: (1) Bayesian nonparametric modeling, which allows us to relax the rigid compositionality of GenIndep and treat large subprograms atomically, and (2) Bayesian hierarchies, which allow these subprograms to be shared across tasks. In particular, we use adaptor grammars (Johnson et al., 2006), which are based on the Pitman-Yor process (Pitman & Yor, 1997) and ideas from the hierarchical Dirichlet process (Teh et al., 2006).

To capture the desired notion of sharing, we introduce a *cache* C_t for each type t, a list which stores all the combinators of type t that have been generated. The idea is that when asked to generate a combinator of type t, we can either return an existing one from C_t (achieving sharing) or a new combinator, which might be constructed from existing combinators from other caches (or even the same cache).

Figure 4 describes the generative process for the new model, which we call GenCache. The new model has two additional hyperparameters, a concentration $\alpha_0 > 0$ and a discount $0 < d < 1$, which determine the amount of desired sharing. Much of GenCache is the same as GenIndep. The major difference is the possibility of generating from the cache, which happens with probability proportional to $|C_t| - N_t d$, where N_t is the number of distinct combinators. A new combinator is generated with probability proportional to $\alpha_0 + N_t d$. Thus, the smaller α_0 and d are, the more sharing we have.

Note that for generating non-primitive combinators, we add a special placeholder $z^\dagger$ to C_t before we recurse. This is needed for the correctness of a recursively hierarchical Pitman-Yor process. A recursive call could return this placeholder, thereby creating a cyclic combinator. Cyclicity actually provides a very natural (but non-standard) way to implement recursion, which is commonly achieved by using variables. Cyclicity allows for direct self-reference without names and has been studied in programming language theory (Ariola & Blom, 1997).

Suppose that for some type t, we generate programs $Z_i = \text{GenIndep}(t)$ for $i = 1, \ldots, K$. GenCache induces a joint distribution $p(Z_1, \ldots, Z_K)$. Although the definition of GenCache is sequential, the induced distribution $p(Z_1, \ldots, Z_K)$ is actually exchangeable (the exact form is given in (5)). Therefore by de Finetti's theorem, there exists a random collection of distributions over programs $\{G_t\}_{t \in \mathcal{T}}$ such that $Z_1, \ldots, Z_K$ are independent.

3.3. Multi-task Learning

Having defined a prior over combinators, let us apply it to multi-task learning. Assume we have K tasks, and for each task $i = 1, \ldots, K$, we are given n training examples $\{(X_{ij}, Y_{ij})\}_{j=1}^n$. For each task i, we would like to infer a latent combinator program Z_i such that the program is *consistent* with those examples; that is, $(\llbracket Z_i \rrbracket\ X_{ij}) = Y_{ij}$ for all $j = 1, \ldots, n$.

We draw each program as follows: $Z_i = \text{GenCache}(t_i)$, where $t_i = t(X_{i1}) \to t(Y_{i1})$ is the type

signature of task i. A nice feature of our setup is that multi-task sharing can still occur across tasks with different type signatures, since the programs Z_is can be composed of common subprograms.

4. Bayesian Inference

The goal of inference is to find the posterior distribution over the latent programs $\mathbf{Z} = (Z_1, \ldots, Z_K)$ given training examples $\{(X_{ij}, Y_{ij})\}$. We first explicate our distribution over $p(\mathbf{Z})$ (Section 4.1) and then discuss how to incorporate the training data and perform approximate inference via MCMC (Section 4.2.1).

4.1. Constraining the Prior

In Section 3, we used GenCache to define a joint distribution $p(\mathbf{Z})$. However, evaluating $p(\mathbf{Z})$ involves integrating out all the possible ways $\mathbf{Z}$ could have been generated. To avoid this marginalization, we introduce the following constraint: Let Q_1 be the event that each combinator which is constructed rather than fetched did not already exist in the cache; consequently, a combinator z which occurs M_z times in its cache $C_{t(z)}$ must have been constructed the first time and fetched the next $M_z - 1$ times. Also, let Q_2 be the event that $\mathbf{Z}$ contains no cyclic combinators, as cyclicity complicates inference. Let $Q = Q_1 \wedge Q_2$.[2]

The significance of Q is that $p(\mathbf{Z}, Q = 1)$ has an analytic expression. Let M_z and N_t be defined as in Figure 4. Then we have:

$$p(\mathbf{Z}, Q = 1) = \tag{5}$$
$$\prod_{t \in \mathcal{T}} \frac{\prod_{i=1}^{N_t}(\alpha_0 + (i-1)d) \prod_{z \in C_t} \psi(z) \prod_{i=1}^{M_z - 1}(i - d)}{\prod_{i=0}^{N_t - 1}(\alpha_0 + i)},$$

where

$$\psi(z) = \begin{cases} \lambda_0 p_0^{\mathcal{B}}(z \mid t) & z \text{ primitive} \\ (1-\lambda_0) p_0^{\mathcal{R}}(\mathbf{r}(z) \mid k(t(z))) p_0^{\mathcal{T}}(s(z)) & \text{otherwise.} \end{cases}$$

Note that as $\alpha_0, d \to 0$, $p(\mathbf{Z}, Q = 1)$ concentrates all probability mass on those $\mathbf{Z}$ which have the absolute smallest number of distinct subprograms, thus encouraging maximum sharing. Larger α_0 and d tend to be less forceful.

One might be tempted to change GenCache to enforce $Q = 1$ directly. This would correspond to defining a prior $p(\mathbf{Z} \mid Q = 1)$, which would be intractable to work with. Remember that GenCache is only used as a vehicle for defining the prior and is not used for forward generation.

[2] Johnson et al. (2006) implicitly assumed Q_1 and did not worry about Q_2 since their hierarchies are not recursive.

4.2. Incorporating Training Data

We now combine the likelihood $p(\mathbf{Y} \mid \mathbf{X}, \mathbf{Z})$ with the prior that we constructed in (5), yielding the posterior $p(\mathbf{Z} \mid \mathbf{X}, \mathbf{Y}, Q = 1)$. The likelihood is an indicator function

$$p(\mathbf{Y} \mid \mathbf{X}, \mathbf{Z}) = \prod_{i=1}^{K} \prod_{j=1}^{n} \mathbb{I}[([\![\mathbf{Z}_i]\!]\ \mathbf{X}_{ij}) = \mathbf{Y}_{ij}], \tag{6}$$

which is 1 iff all programs are consistent with the training examples. This sharply discontinuous likelihood creates a posterior whose support is disconnected, making it difficult to design an MCMC kernel that can jump across zero probability states and explore all programs. Our strategy will therefore be to rely on a restricted set of candidate correct programs to be provided.

We use a *candidate structure* to compactly represent an exponentially large set of programs, similar to the version space algebra of Lau et al. (2003). Formally, a candidate structure s is associated with the following: (1) a partial function f_s which specifies the desired computation; and (2) a set S_s, where each element is either a primitive combinator (element of $\mathcal{B}$) or a triple $(\mathbf{r}, s_1, s_2)$, where $\mathbf{r}$ is a router and s_1, s_2 are candidate structures. We require that f_s be compatible with $(\mathbf{r}, f_{s_1}, f_{s_2})$, meaning that for any extension g_{s_1} of f_{s_1} and g_{s_2} of f_{s_2}, $(\mathbf{r}\ g_{s_1}\ g_{s_2})$ is an extension of f_s.

Let $U(s)$ be the set of combinators defined by recursively walking down the structure and choosing elements in S_s to follow; formally,

$$U(s) = (S_s \cap \mathcal{B}) \cup \bigcup_{(\mathbf{r}, s_1, s_2) \in S_s} \{\mathbf{r}\} \times U(s_1) \times U(s_2). \tag{7}$$

It can be verified that any $z \in U(s)$ is an extension of f_s. Also, let $S_*(s) = \{s\} \cup \bigcup_{(\mathbf{r}, s_1, s_2) \in S_s} (S_*(s_1) \cup S_*(s_2))$ denote all candidate structures in s. Also, let $R(z)$ be the programs which can be obtained by refactoring z; this set is defined more precisely in Section 4.2.2.

We assume a candidate structure s_i is given for each task i. The target sampling distribution is then $p(\mathbf{Z} \mid \mathbf{Z} \in \mathbf{U}, Q = 1)$, where $\mathbf{U} = \prod_{i=1}^{K} \cup_{z \in U(s_i)} R(z)$, programs which can be refactored from some candidate. Our sampler uses two types of moves to explore $\mathbf{U}$: *candidate switching* moves (Section 4.2.1) and *refactoring* moves (Section 4.2.2).

4.2.1. Candidate Switching

For purposes of switching candidates, it will be convenient to operate on an expanded set of random variables which parametrize $\mathbf{Z}$. Let $\mathbf{S}_* = \cup_{i=1}^{K} S_*(s_i)$ denote the candidate structures across all tasks. Define

$\mathbf{G} = \{G_s\}_{s \in S_*}$, where $G_s \in S_s$. Let $\mathbf{Z}(\mathbf{G})$ denote the K programs formed by following $\mathbf{G}$ down the candidate structures. Note that $\mathbf{G}$ also contains variables whose candidate structures specify subprograms not part of $\mathbf{Z}(\mathbf{G})$. Let $S_*(\mathbf{Z}, \mathbf{G})$ be only the candidate structures which are part of $\mathbf{Z}(\mathbf{G})$.

Given these structures the candidate switching move is now straightforward. We use the following Metropolis-Hastings proposal: choose $s \in \mathbf{S}_*(\mathbf{Z}, \mathbf{G})$ uniformly at random (with probability $\frac{1}{|\mathbf{S}_*(\mathbf{Z},\mathbf{G})|}$), and propose changing G_s to an element of S_s uniformly at random. This proposal is accepted with the usual Metropolis-Hastings probability, $\min\{1, \frac{p(\mathbf{Z}',Q=1)|\mathbf{S}_*(\mathbf{Z}',\mathbf{G})|}{p(\mathbf{Z},Q=1)|\mathbf{S}_*(\mathbf{Z},\mathbf{G})|}\}$, where $\mathbf{Z}'$ is the new state. The ratio of model probabilities can be computed according to (5).

4.2.2. Refactoring

So far, we can switch candidates and let the prior drive sampling to those programs in $\mathbf{U}$ with more sharing across tasks. However, as mentioned in Section 2.1, the potential for sharing is sometimes not immediate. Consider the two programs in $\mathbf{Z}^{(2)}$ (Figure 6) for computing the min and the max. Although the programs differ only in one leaf, this similarity is not reflected by examining the subprograms they share. Refactoring the programs to $\mathbf{Z}^{(4)}$ exposes the modularity while still preserving the same functionality, and indeed, $\mathbf{Z}^{(4)}$ has much higher likelihood than $\mathbf{Z}^{(2)}$.

We define a set of *refactoring transformations* $\mathcal{F}$, where each transformation $[f_1 \leftrightarrow f_2] \in \mathcal{F}$ is defined on a pair of combinator patterns. One example is the basic **B**-transformation $[((\mathbf{B}\ x\ y)\ z) \leftrightarrow (x\ (y\ z))]$, depicted at the top of Figure 2. This transformation states that for all combinators $x, y, z \in \mathcal{C}$, $((\mathbf{B}\ x\ y)\ z)$ has the same denotation as $(x\ (y\ z))$; we can therefore freely replace one with the other. Figure 2 lists three other basic transformations based on removing/adding **C**, **S**, and **I**.

These four basic transformations work for programs that take no arguments, which is clearly insufficient for our needs. For example, none of the basic transformations can account for the equivalence between programs $\mathbf{Z}_1^{(1)}$ and $\mathbf{Z}_1^{(3)}$ for computing $x - y + 1$. At the same time, modulo the presence of extra routers, the difference between the two at the core is just a **C**-transformation; therefore, we need a more general version. We add to $\mathcal{F}$ higher-order transformations which allow one to work when other routers are present (see Figure 5). We can apply a higher-order **C**-transformation with $\mathbf{r} = \emptyset$, $\mathbf{r}_0 = \mathbf{BB}$, $\mathbf{r}_1 = \emptyset$, $\mathbf{r}_0' = \mathbf{CC}$, $\mathbf{r}_1' = \mathbf{BB}$ to move between $\mathbf{Z}_1^{(1)}$ and $\mathbf{Z}_1^{(3)}$. Let $R(z)$ be the set of combinators reachable by applying transformations in $\mathcal{F}$ to z.

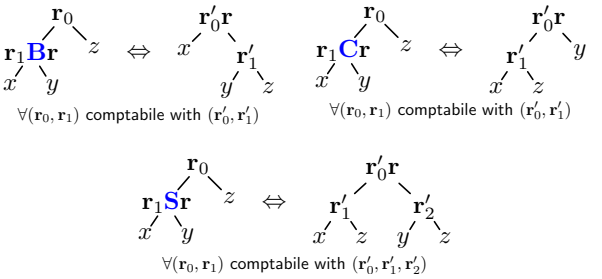


Figure 5. Templates specifying higher-order transformations, which allow refactoring in the presence of other routers. Let $\mathbf{r}_0$ and $\mathbf{r}_1$ be any routers that send arguments $a_1, \ldots, a_k$ each to some subset of $\{x, y, z\}$. After they apply, z will be routed down by the *core* **B**,**C**, or **S**. We require that $\mathbf{r}_0'$ and $\mathbf{r}_1'$ (and also $\mathbf{r}_2'$ in the case of **S**) be *compatible* with $\mathbf{r}_0$ and $\mathbf{r}_1$, that is, they route $a_1, \ldots, a_k$ to the same subset of $\{x, y, z\}$ in the new tree structure induced by the core router. There no constraints on $\mathbf{r}$.

We can turn the set of transformations $\mathcal{F}$ into a Metropolis-Hastings proposal as follows: First choose a transformation f uniformly from those in $\mathcal{F}$ that involve routers of some bounded order (to keep the set finite). Then, choose a task i and subtree z of Z_i uniformly at random. If f is applicable at z, propose replacing z with $f(z)$. The proposal is accepted or rejected according to the usual Metropolis-Hastings acceptance ratio.

Note that refactoring disrupts candidate structures. When f is applied, we remove any affected candidate structures from $S_*(\mathbf{Z}, \mathbf{G})$ (all descendants of the transformed tree) add them back when f is undone. Candidate switching moves will simply skip over those structures that do not contribute to $\mathbf{Z}$.

5. Experiments

We first illustrate our model in a simple arithmetic domain (Section 5.1) and then present experiments in the text editing domain (Section 5.2).

5.1. An Arithmetic Example

Consider the two tasks shown at the top of Figure 6. For the candidate structure s_i of each task $i \in \{1, 2\}$, we set S_{s_i} to (the degenerate candidate structures corresponding to) $\{Z_i^{(1)}, Z_i^{(2)}\}$ and initialize the sampler to $\mathbf{Z}^{(1)}$. There are two generalizations of the training examples: one using arithmetic operations ($\mathbf{Z}^{(1)}$) and

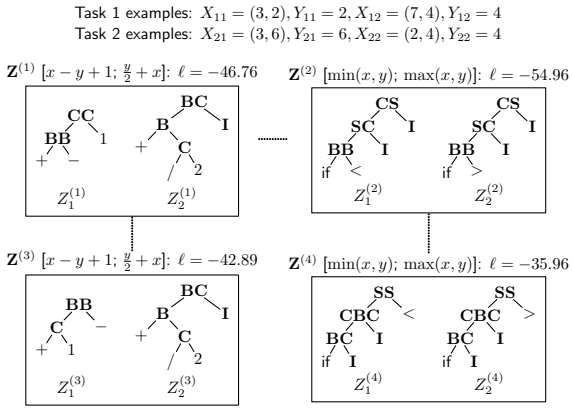


Figure 6. An arithmetic example: Each of the four boxes represent a state $\mathbf{Z}$ could be in; the dotted edges represent paths that our sampler must follow. $\mathbf{Z}^{(1)}$ and $\mathbf{Z}^{(2)}$ are provided by the candidate structures, whereas $\mathbf{Z}^{(3)}$ and $\mathbf{Z}^{(4)}$ are reachable only by refactoring. Although $\mathbf{Z}^{(1)}$ is simpler than $\mathbf{Z}^{(2)}$, as confirmed by the log-likelihood (ℓ), the true simplicity of $\mathbf{Z}^{(2)}$ can be revealed only by refactoring into $\mathbf{Z}^{(4)}$, which has the highest ℓ.

one using comparison operations ($\mathbf{Z}^{(2)}$). As explained in Figure 6, while the former yields smaller individual programs, the latter, after refactoring, is simpler when considered as a whole.

5.2. Text Editing

We now turn to text editing. We can represent the editing process functionally by encapsulating the state of the editor into a variable s of type state, which contains the contents of the buffer, the cursor position, the selection extent, and the contents of the clipboard. Figure 7 describes the primitive combinators.

We took 24 editing scenarios obtained from Tessa Lau, with substantial but not complete overlap with those reported in Lau et al. (2003). Each example consists of a sequence of user actions. Suppose we have two examples [(move 10), (insert *hello*)] and [(move 28), (insert *hello*)]. We construct candidate structures as follows: The root candidate structure is a composition of the primitive combinators corresponding to those actions; in our example, (**B** (**C** insert s) (**C** move s')), where s and s' are candidate structures such that f_s : state → int returns 10 and 28, respectively, on the initial states of the two examples; and $f_{s'}$: state → string returns *hello* on both. Next, for each candidate structure s for which f_s : state → int, we have a small set of rules for constructing S_s by considering possible ways of returning the desired integer: returning an absolute offset x, returning a relative offset (+ (pos s) x), matching a string (find s q w), etc., for various values of x, w, q. For each candidate structure s for which f_s : state → string, we construct S_s in a similar vein.

Base types: state, int, string
Primitive combinators:
$\ldots, -2, -1, 0, 1, 2, \ldots, +, -$, string-append
(pos s): cursor position of state s
(caseNum s): index of the current example
(find s q w): position of k-th first/last occurrence of w in s after/before (pos s); exact variant is specified by q
(coarse-find s w): same as find, but operates on a coarsened version of w and s (e.g., "aa aaaa00x" replaces "at ICML10!")
(begin-word s): position of beginning of next word
(whitespace s): position of next whitespace character
(end-of-file s): position at end of file
(move s i): new state where the cursor position is i
(select s i j): new state where contents between i and j are selected
(paste s): new state where clipboard contents are inserted at the current position
(cut s): new state where the selected text is cut to the clipboard
(copy s): new state where the selected text is copied to the clipboard
(delete s i) new state where the text between (pos s) and i is deleted
(insert s w) new state where w has been inserted at the current position
(delete-selection s w) new state where the selected text in s is deleted

Figure 7. Description of the text editing domain. Our primitives are similar to the ones used in Lau et al. (2003),

Recall that refactoring allows us to expose new subprograms. Using the full set of transformations $\mathcal{F}$ is too general for text editing, so we replace $\mathcal{F}$ as follows to target two types of desired sharing: We use **B** transformations on the composition of user actions at the top of the candidate structure; this corresponds to forming a hierarchical grouping via tree rotations. Second, we allow extraction/unextraction of string-typed primitive combinators to the top of the program using a single program transformation, as in Figure 1(b).

Having defined our candidate structures and allowable set of refactorings, we now apply MCMC to infer the programs. We set the hyperparameters of the model to $\alpha_0 = 1$ and $d = 0.2$. We perform 1000 passes over our training data, applying both candidate and refactoring transformations, and annealing from a temperature of 10 down to 1 during the first 900 iterations. During the final 100 iterations, we collected samples of $\mathbf{Z}$. On a test input X on task i, we perform approximate Bayesian averaging by predicting the most common output ($[\![Z_i]\!]\ X$) over samples Z_i.

We compared our approach with two baselines: (1)

Table 1. Average test error rates across all text editing tasks (the mean is reported over 10 trials) with n training examples per task. Note that using an independent prior actually works substantially worse than using a uniform prior due to an overly-aggressive penalization of the program size. Using a joint prior over all tasks offers substantial improvements.

n	2	3	4	5
Uniform prior	19.6	17.0	7.0	2.7
Independent prior	25.4	21.8	20.9	12.1
Joint prior	13.9	9.5	5.9	3.4

using a uniform prior over programs (in $\mathbf{U}$), (2) and using our GenCache model but treating each task independently. Table 1 shows our results as the number of training examples n varies. Independent learning performs worse than no learning, but joint learning works best.

6. Related Work

Combinatory logic (without routing) has been used to learn programs in genetic programming (Briggs & O'Neill, 2006) with the goal of facilitating program transformations as in our work, but this approach does not provide a declarative prior on programs. Lau et al. (2003) used a hand-crafted prior. In contrast, we learn a distribution over programs from multiple tasks.

An important special case of functional programs are logical formulae, programs that return boolean values. Bayesian inference has been used to induce logical formulae using a PCFG in several contexts, e.g., in representing natural language semantics (Piantadosi et al., 2008) and cognitive concepts (Goodman et al., 2008b).

A different point of convergence of programming languages and probabilistic modeling is in Church (Goodman et al., 2008a). They infer the random trace of a fixed (stochastic) program, whereas we infer the (deterministic) program itself, although Mansinghka (2009) did show that Church, being universal, can be used in principle to infer programs by forward simulation and rejection.

7. Conclusion

We have presented a hierarchical Bayesian model of combinator programs which enables multi-task sharing of subprograms. One of the main new ideas is refactoring to reveal shared subprograms via safe transformations. Programs are rich objects which have been studied at length from a logical perspective. Treating them as objects of statistical inference raises many new and exciting challenges.

Acknowledgments We thank Tessa Lau for providing us with the text editing dataset and anonymous reviewers for their comments.

References

Ariola, Z. M. and Blom, S. Cyclic lambda calculi. In *Theoretical Aspects of Computer Software*, pp. 77–106, 1997.

Briggs, F. and O'Neill, M. Functional genetic programming with combinators. In *Third Asian-Pacific workshop on Genetic Programming*, pp. 110–127, 2006.

Cypher, A. *Watch what I do: Programming by demonstration.* MIT Press, 1993.

Goodman, N. D., Mansighka, V. K., Roy, D., Bonawitz, K., and Tenenbaum, J. B. Church: a language for generative models. In *Uncertainty in Artificial Intelligence (UAI)*, 2008a.

Goodman, N. D., Tenenbaum, J. B., Feldman, J., and Griffiths, T. L. A rational analysis of rule-based concept learning. *Cognitive Science*, 32:108–154, 2008b.

Hankin, C. *An Introduction to Lambda Calculi for Computer Scientists.* Lightning Source, 2004.

Johnson, M., Griffiths, T., and Goldwater, S. Adaptor grammars: A framework for specifying compositional nonparametric Bayesian models. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 641–648, Cambridge, MA, 2006. MIT Press.

Lau, T., Wolfman, S., Domingos, P., and Weld, D. S. Programming by demonstration using version space algebra. *Machine Learning*, 53:111–156, 2003.

Mansinghka, V. *Natively Probabilistic Computation.* PhD thesis, MIT, 2009.

Piantadosi, S. T., Goodman, N. D., Ellis, B. A., and Tenenbaum, J. B. A Bayesian model of the acquisition of compositional semantics. In *Proceedings of the Thirtieth Annual Conference of the Cognitive Science Society*, 2008.

Pitman, J. and Yor, M. The two-parameter Poisson-Dirichlet distribution derived from a stable subordinator. *Annals of Probability*, 25:855–900, 1997.

Schönfinkel, M. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92:305–316, 1924.

Teh, Y. W., Jordan, M. I., Beal, M., and Blei, D. Hierarchical Dirichlet processes. *Journal of the American Statistical Association*, 101:1566–1581, 2006.