Chapter 19

Learning as Bayesian inference over programs

Steven T. Piantadosi, Joshua S. Rule, and Joshua B. Tenenbaum

The previous chapters show that Bayesian models form a powerful approach to understanding cognitive processes and illustrate how they can use diverse representations including continuous spaces, graphs, logical formulas, and grammars. This chapter covers foundational ideas in Bayesian inference over programs. The idea that people consider algorithmically rich hypothesis spaces has been found to match human performance on learning tasks. The theory of program learning also provides a compelling picture of how to scale cognitive theories towards the breadth of structures and processes that people fluidly acquire. The idea that people do inference over programs can be motivated by the remarkably broad set of things that human children acquire—from learning games, to social rules, or new formal systems like logic and programming languages. Our ability to interface with external dynamical or computational systems—a car engine, a pet, or a cash register—likely requires us to form mental models of these objects. Our successes across a diverse range of systems suggest that we are able to learn over hypothesis spaces that are computationally powerful, perhaps effectively the space of all computations constrained by our limited memory.

If $\mathcal{H}$ is a space of computer programs and d is some observed data, often program inputs and outputs, this chapter outlines how one typically can define a prior $P(h)$ and a likelihood $P(d \mid h)$, as well as some of the considerations that go into both. We then discuss typical statistical inference schemes for programs, focusing primarily on Markov Chain Monte Carlo methods. We conclude with a discussion of advanced methods that learn subroutines, draw on a rich family of inferences, and learn to encode fundamentally new logical structures.

19.1 Background

Alan Turing's work in the mid-1900s to develop a mathematical model of computing (Turing, 1936, 1950) is widely recognized as foundational to computer science. It should also be celebrated as a profound discovery in cognitive psychology. At the time Turing was working, the word "computer" referred to *people* who performed computations.[1] Turing's formalization of computation (Turing-machines) was thus an abstract model of what humans could do by combining a few basic abilities. What Turing discovered is staggering: operations like contingently updating an internal state and interfacing it with memory allow a human computer to simulate *any other* logical machine. With the right program, a person—or mechanical/electronic computer—can simulate everything from physics to the stock market by combining elementary operations in new ways.[2]

This universal ability is articulated in the **Church-Turing thesis**, which holds that everything which can be computed can be computed on a Turing machine, or equivalently programmed into a standard computer. The key is that basic operations can compose to determine more complex behavior, just as a complex novel can be composed from a few dozen characters. Early in the history of computer science, logicians developed seemingly distinct models of computation (e.g., Turing machines, lambda calculus, combinatory logic, Post systems, and term rewriting) that were then found to be equal in computational power in that each system can simulate all of the others. Today, remarkably diverse systems—arrays of cells that blink on and off based on the behavior of their neighbors, billiard balls bouncing, simple systems of rewrite rules, and neural networks (Wolfram, 2002; Abelson & Sussman, 1996; Fredkin & Toffoli, 1982; Pérez, Marinković, & Barceló, 2019)—have been found capable of universal computation. This means that there are many possible systems which could form the basis of human computational abilities.

People's cognitive capacity for universal computation appears to be distinctive in animal cognition. We, and seemingly we alone, discover fundamentally new kinds of representations, internalize complex

[1] Turing's writing referred to computers as "he."

[2] Such scope is very much unlike early mechanical computers (Aspray, 2000). Charles Babbage Babbage (1864), for example, viewed computation as merely a "mechanism for assisting the human mind in executing the operations of arithmetic." His colleague, Ada Lovelace, was perhaps the first to see the real potential for other forms of computation (Lovelace, 1842), but this vision was not realized until the creation of digital computers in the mid-twentieth century.

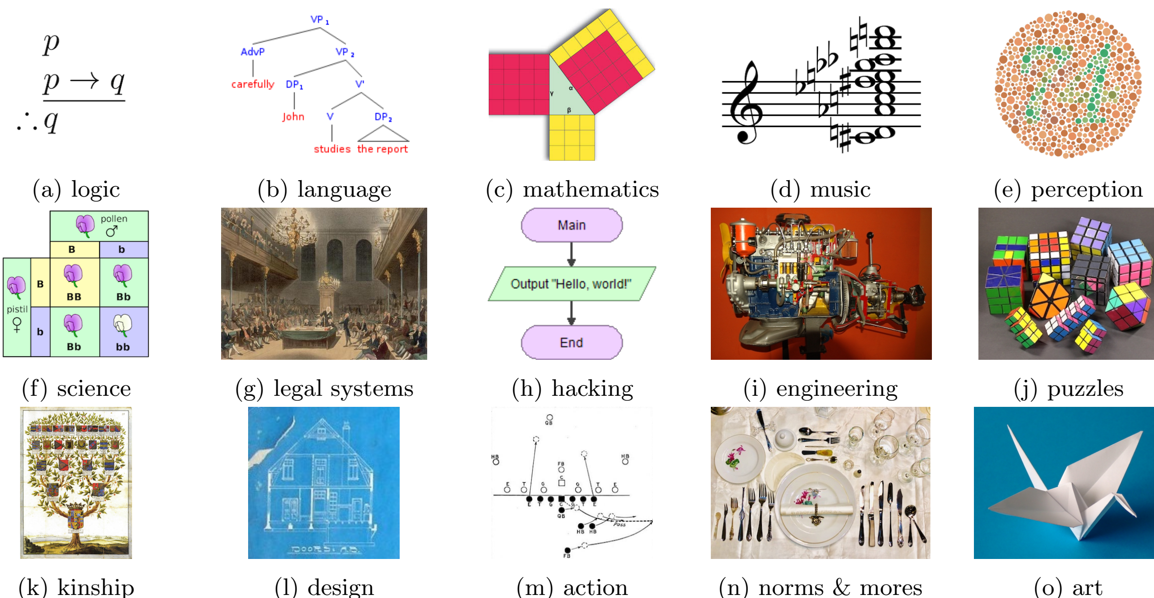

Figure 19.1: A few domains that can be described well using programs, usually because they involve multiple interacting or structural logical rules, or abstract processes.

procedures, and revise and improve complex algorithms. Some of these processes are the bread and butter of cognitive psychology: people can learn to count, read, reason, and understand physical systems. But human abilities are much broader. People can also learn to fly the space shuttle, play the cello, perform a quadruple Lutz, or repair accordions. We learn to manipulate wholly new logical systems like calculus, Rubik's cube, and the rules of legal systems. We learn to reason about social entities and think strategically about competitors. When compared to other species, our ability to model the world and leverage it in pursuit of our goals is virtually limitless.

This has led many to hypothesize that cognition must have the capacity to *infer* or *internalize* novel algorithmic processes—programs. Much of human knowledge is well-described using programs (Figure 19.1). In turn, program induction models have been used to model many aspects of learning across a variety of domains. For example, Amalric et al. (2017) develop a model of mathematical reasoning which describe geometrical patterns as functions from step numbers (i.e. step 1, step 2) to positions in space. Others have studied human induction of logical concepts (Goodman, Tenenbaum, Feldman, & Griffiths, 2008; Kemp, 2009, 2012; Piantadosi, Tenenbaum, & Goodman, 2016) with models that take feature-based descriptions of objects as input and compute true/false judgments about whether each object belongs to the concept. Lake, Salakhutdinov, and Tenenbaum (2015) presented a model of motor action which takes images of hand-written characters and produces new exemplars of the same character. A number of models learn theories that take relational data as input and predict novel relations as output for domains like magnetism, kinship, causality, biological taxonomies, and even spherical geometry (Kemp & Tenenbaum, 2008; Kemp, Tenenbaum, Niyogi, & Griffiths, 2010; Goodman, Ullman, & Tenenbaum, 2011; Ullman, Goodman, & Tenenbaum, 2012; Mollica & Piantadosi, 2021). Piantadosi, Tenenbaum, and Goodman (2012) and Piantadosi (under review) modeled the acquisition of the counting procedure as discovering a program for transforming sets of objects into number word labels. A number of other models deal with more abstract mathematical relationships, including binary sequences (Planton et al., 2021), numerical concepts like even and odd (Tenenbaum & Griffiths, 2001), fractals (Lake & Piantadosi, 2020), and functions over lists of natural numbers (Rule et al., under review). Siskind (1996) modeled lexical semantics, and other models have examined additional aspects of language, including: quantifier

semantics (Piantadosi, 2011), morphology (Ellis, 2020), syntax (Yang & Piantadosi, 2022), question answering (Rothe, Lake, & Gureckis, 2017), and pragmatics (Goodman & Lassiter, 2015; Goodman & Frank, 2016). Several others deal with visual reasoning over visual representations (Ellis, Morales, Sablé-Meyer, Solar-Lezama, & Tenenbaum, 2018), three-dimensional figures (Overlan, Jacobs, & Piantadosi, 2017), and even Bongard problems (Depeweg, Rothkopf, & Jäkel, 2018).

19.2 The hypothesis space

Most models of program learning do not work with Turing machines (c.f. (Graves, Wayne, & Danihelka, 2014; Kim & Bassett, 2022)) because behavioral work in cognitive science makes a convincing case that human cognition is **compositional**. As discussed in Chapter 18, Alonzo Church developed **lambda calculus** to formalize computation as function composition. In his framework, complex programs arise from gluing the input of one function to the output of another—e.g. composing f and g to form $f \circ g$. This kind of computation by composition survives today in functional programming languages like Haskell and Lisp which express programs as a composition of primitive functions rather than a sequence of steps. A similar compositional approach is dominant in formalizing natural language semantics (Heim & Kratzer, 1998; Steedman, 2000; Blackburn & Bos, 2005). For example, the meaning of a sentence like "I found the concertina that Pietro dropped" can be captured by composing the individual function for each word in the sentence. Compositionality in language is likely deeply related to compositionality in thinking, leading many to argue for a "language of thought" (Fodor & Pylyshyn, 1988; Fodor, 1975; Piantadosi & Jacobs, 2016; Goodman, Tenenbaum, & Gerstenberg, 2014) in which thoughts themselves are comprised of constituent elements into sentence-like structures. This compositional approach has also been used in learning models, which operate over the space of possible compositions. Siskind (1996), for example, formulated a word learning model where a word like "lift" would have the meaning

$$\texttt{lift(x,y)=CAUSE(x,GO(y,UP))}. \tag{19.1}$$

Here, `CAUSE`, `GO`, and `UP` are primitive operations that compose to express the meaning of `lift`. The expression in (19.1) means that "x lifts y" if `x` causes `y` to go up. This kind of model can be contrasted with models in which "lift" might be a conceptual primitive itself, so that the only problem learners face is determining the mapping from words to already-given meanings. In compositional models, learners face the much more difficult task of constructing a correct meaning from constituent parts, and finding the correct composition out of the large (often infinite) space of possibilities.

19.2.1 A grammar of hypotheses based on types

Like the example in (19.1), we will typically think of programs as **functions** that map inputs to outputs. `lift` in (19.1), for example, takes two objects `x` and `y` and returns a Boolean value corresponding to whether or not `x` lifted `y`. When building a program learning model, it is our job to choose appropriate inputs and outputs, which often shapes what function we learn as well as what primitives are necessary.[3] In typical program learning models, the hypothesis space consists of all semantically sensible ways of combining an assumed set of primitive operations. A learner couldn't even evaluate a hypothesis like `lift(x,y)=CAUSE(CAUSE,CAUSE)` or `lift(x,y)=GO(CAUSE,y)` because these make no sense.

The primary way that we ensure that programs are semantically coherent is to treat each primitive as having fixed input and output types. To illustrate, `lift` might take two things of type *object* as arguments. This means that the arguments `x` and `y` should be objects: you can lift a potato but not an

[3] For example, we could alternatively imagine a version of `lift` that takes as argument an object `x` and a context (e.g. set of objects) and returns the objects that `x` lifted.

abstract thing like a joke; you also can't lift a whole sentence or proposition like "Smoking cigarettes kills." The primitive `GO` might take an object and a direction, so `GO` can make a potato go left or right or up or down. Importantly, it cannot make a potato go "carrot" because it cannot take an object as its second argument. The primitive `CAUSE` is a little more interesting in that it takes an object (a cause-er) `x` and an event that happens and asserts that `x` caused the event. Thus, `CAUSE(x,GO(y,up))` means that `x` causes `y` to go up, this composition would evaluate in to `true` only when the causal relationship was true.

The assumed types can be enforced using a context-free grammar (see Chapter 16) to generate the full program composition. The nonterminals of this grammar specify the input and output types of each primitive function. For example, if `GO` takes an object and a direction and returns a Boolean truth value, we might write this in the grammar as "`BOOL` → `GO(OBJECT, DIRECTION)`." This means that when we are writing a program, any time we see a `BOOL` symbol, we may replace it with a call to the function `GO` applied to some `OBJECT` and some `DIRECTION`, both also derived from the grammar. Writing out more rules for this grammar gives,

$$\begin{aligned}\texttt{OBJECT} &\rightarrow \texttt{x} \mid \texttt{y} \mid \texttt{car} \mid \texttt{tree}\\ \texttt{DIRECTION} &\rightarrow \texttt{up} \mid \texttt{down} \mid \texttt{left} \mid \texttt{right} \mid \texttt{in} \mid \texttt{out}\\ \texttt{BOOL} &\rightarrow \texttt{GO(OBJECT, DIRECTION)} \mid \texttt{CAUSE(OBJECT, PROPOSITION)}\\ &\mid \texttt{or(BOOL, BOOL)} \mid \texttt{and(BOOL, BOOL)} \mid \texttt{not(BOOL)}.\end{aligned}$$

This grammar specifies a vast hypothesis space. Even this toy example contains 24 possible `GO`ing actions (each of 4 objects going in each of 6 directions) which can then be combined with conjunction (`and`), disjunction (`or`), and negation (`not`). Indeed, this combinatorial explosion is intentional. One of our chief design goals for program learning models is that a few built-in operations combine to express a huge variety of concepts. Such power is, as above, motivated by peoples' ability to acquire such a rich diversity of concepts. For example, learners with this system could hypothesize that x lifts y if a car makes `x` go right or a car makes `y` go left,

$$\texttt{lift(x, y) = or(CAUSE(car, GO(x, right)), CAUSE(car, GO(y, left)))}. \tag{19.2}$$

This hypothesis, though incorrect, is conceivable and possible—that is why we want the grammar to include it. That said, (19.2) is a bit unnatural, likely due to its complexity. This particular hypothesis uses eleven primitives and should perhaps be dispreferred a priori relative to simpler options, including the correct one. We can build a simplicity preference into our model by converting the grammar into a *probabilistic* grammar. This requires us to give a probability to each rule expansion (see Chapter 16). We can then compute the probability of a full expression by multiplying the probabilities of each rule expansion it uses. Such a probabilistic grammar can then be used to define the prior for the program learning model. Because the prior typically favors short programs, Bayesian program learning models usually will tend to prefer programs that *compress* the observed data into a concise description. This compression is an effect of the broader generalization effect typical of Bayesian learners (see Chapter 3).

19.2.2 A simple grammar of list functions

We can see many Bayesian inference effects at work in the domain of **list functions** where learners observe an unknown function $\mathcal{F}$ that transforms one list of numbers into another (Rule et al., under review). For example, we might be told only a single data point:

$$[3, 6, 1, 7, 3, 2, 9] \xrightarrow{\mathcal{F}} [9, 7, 6, 3, 3, 2, 1], \tag{19.3}$$

where the list on the left has been transformed to the list on the right by the unknown function $\mathcal{F}$. From this example, it is easy to guess that $\mathcal{F}$ sorts lists in descending order. After learning that, we could

readily apply $\mathcal{F}$ to a new list. Other data sets would lead to different guesses about what $\mathcal{F}$ does, and consequently different generalizations:

$$[3,6,1,7,3,2,9] \xrightarrow{\mathcal{F}} [2,6]. \tag{19.4}$$

This is consistent with at least a few generalizations: (i) remove odd numbers; (ii) keep only 6s and 2s; (iii) keep the second and sixth elements of the list; (iv) keep only the second from the front and second from the back, etc. These options highlight a strength of probabilistic models, namely that they can distribute belief across multiple hypotheses. These beliefs are sensitive to the amount and content of the data. For example, if we next saw two data points,

$$\begin{aligned} [3,6,1,7,3,2,9] &\xrightarrow{\mathcal{F}} [2,6] \\ [7,5,2] &\xrightarrow{\mathcal{F}} [2]. \end{aligned} \tag{19.5}$$

the data would be best supported under (i) and (ii) but not (iii) and (iv), leading to revised beliefs. One more data point might give:

$$\begin{aligned} [3,6,1,7,3,2,9] &\xrightarrow{\mathcal{F}} [2,6] \\ [7,5,2] &\xrightarrow{\mathcal{F}} [2] \\ [8,8,8] &\xrightarrow{\mathcal{F}} [8,8,8]. \end{aligned} \tag{19.6}$$

While it may feel at this point as though we have narrowed in on a single hypothesis, remember that there are infinitely many possible hypotheses that are consistent with any dataset. However, the prior will assign the shortest, simplest hypotheses that are consistent with the data the highest probability, potentially leading to strong beliefs about the right answer.

Introspecting on this example motivates some basic operations we might want in a formal model of the list domain, including operations to manipulate and select from lists:

$$\begin{aligned} \texttt{LIST} &\rightarrow \texttt{filter(PREDICATE, LIST)} \mid \texttt{pair(NUMBER, LIST)} \mid \\ &\quad \epsilon \mid \texttt{reverse(LIST)} \mid \texttt{rest(LIST)} \mid \texttt{lst} \\ \texttt{NUMBER} &\rightarrow \texttt{first(LIST)} \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \mid \texttt{9} \\ \texttt{PREDICATE} &\rightarrow \texttt{is-even} \mid \texttt{is-odd} \mid \texttt{is-prime} \mid \ldots \end{aligned}$$

Here, `lst` is the argument provided to $\mathcal{F}$; ϵ is an empty list; `pair` builds lists by prepending a number onto an existing list; `first` returns the first element of a list while `rest` returns everything except the first element; `filter` loops over a list, keeping only elements that satisfy its predicate; `reverse` reverses a list; and we have several built-in predicates for operations that adults learning $\mathcal{F}$ might be expected to know. Predicates are themselves functions that can be applied to numbers (e.g. they map `NUMBER` to `BOOL`). Perhaps unsurprisingly, these operations are similar to those found in list-processing programming languages designed for human use (Abelson & Sussman, 1996). Using these operations, we could formulate hypotheses like (i) as

$$\mathcal{F}\texttt{(lst) = filter(is-even, lst)} \tag{19.7}$$

or (iv) as

$$\mathcal{F}\texttt{(lst) = pair(first(rest(lst)), first(rest(reverse(lst))))}. \tag{19.8}$$

Like the grammar for `lift`, this grammar allows us to express many other hypotheses. For example, $\mathcal{F}$`(lst) = pair(9, rest(lst))` would replace the first element of the list with 9.

19.2.3 Variables and abstraction

The operations we have discussed so far fit into a (probabilistic) context-free grammar, but most programming language grammars are not context-free. One reason is that introducing a variable changes the programs that are valid: the expression `first(foo)` is only valid if `foo` has previously been declared as a variable. We face this problem if we introduce explicit variables into cognitive programs. For example, we might want to filter a list by whether a number is "even or prime" and in that case, we would have to include Boolean `or` and use it to define a subroutine

$$\texttt{even-or-prime(y) := or(is-even(y), is-prime(y)).} \tag{19.9}$$

The subroutine is needed because of the types: `or(is-even(y), is-prime(y))` returns a `BOOL`, not a `PREDICATE`, and so can't serve as the first argument of `filter`. If we just wrote `filter(or(is-even(y), is-prime(y)), lst`, it would not be clear that we intended `y` to be the variable mapped over each element of `lst`. Defining the function `even-or-prime` resolves these issues, so we can write,

$$\mathcal{F}\texttt{(lst) = filter(even-or-prime, lst).} \tag{19.10}$$

It is clunky to define an entire function `even-or-prime` if it is only going to be used once. Lambda calculus is useful for this because it allows us to define functions without having to give them names, often called **anonymous functions.** With lambda calculus, we would write (19.9) as

$$\lambda\ \texttt{y.or(is-even(y), is-prime(y)).} \tag{19.11}$$

Here, "λ`y`." simply means "what comes next is a function of `y`." The whole thing can then be substituted into the first `PREDICATE` argument of `filter` because this lambda expression takes a number (`y`) and returns a `BOOL`, and so matches the type of `PREDICATE`:

$$\mathcal{F}\texttt{(lst) = filter(}\lambda\texttt{y.or(is-even(y),is-prime(y)),lst).} \tag{19.12}$$

Many program induction libraries will allow lambda functions in their grammar.

Both (19.9) and (19.12) introduce the variable `y`, which is not in the grammar. It is only the "λ `y`" (or "`even-or-prime(y)=...`") that introduces `y` as a variable and a new symbol, and it could have been called `z` or `symbol1244` without altering what the program computes. The way that variables dynamically change the underlying grammar is an annoying technical problem that gets handled in several ways across different implementations of cognitive models. The simplest solution is to build in a set number of variables and allow the learning model to sort out which ones go where, assuming an appropriate means of evaluating variables that are not defined. A more general approach that keeps the grammar context-free is to introduce higher-order functions, an approach taken in the Fleet software package. For example, we might introduce a disjunction function, let's call it `disj` that takes two `PREDICATE`s and returns a new `PREDICATE` of an argument `x`:

```
disj(p1, p2)(x) := or(p1(x), p2(x))
```

Here, `disj` is called a **higher-order function** because instead of taking lists or numbers as arguments, it takes entire functions (`PREDICATE`s). Then we might form an expression like

$$\mathcal{F}\texttt{(lst) = filter(disj(is_even, is_prime), lst)} \tag{19.13}$$

One downside to this approach is that it requires adding higher-order functions to the grammar that are conceptually close to other existing primitives. `disj`, for example, is nearly identical to `or`, save for their types.[4] A third approach is to actually change the grammar and add rules for variables once they are

[4]Some approaches include **type-raising operations** `T` such that `T(or)` was equivalent to `disj` (but `T` could be applied to any arbitrary function).

introduced, an approach taken in the earlier LOTlib3 software package. The model must ensure that variables are given unique names, but there is a standard scheme for doing so known as **De Bruijn indexing**. While explicitly changing the grammar is flexible, it also increases the complexity of the type system. These kinds of engineering choices are typically far outside the resolution we have for capturing human cognition—in fact, the priors for one approach can often be made similar or identical to priors for another.

Another reason that many programming languages are not context-free is that the type system itself may not be context-free. To see why this might happen, consider our current grammar. Expressions with type `NUMBER` will always evaluate to numbers, and `LIST` expressions will always reduce to lists of numbers. Suppose, however, that we also want to include lists of *lists* of numbers. We cannot accommodate this type without extending our language and grammar:

$$\begin{aligned}
\texttt{LIST}_\ell &\rightarrow \texttt{filter(PREDICATE}_\ell\texttt{, LIST}_\ell\texttt{)} \mid \texttt{pair(LIST}_n\texttt{, LIST}_\ell\texttt{)} \mid \texttt{rest(LIST}_\ell\texttt{)} \mid \texttt{...} \\
\texttt{LIST}_n &\rightarrow \texttt{filter(PREDICATE}_n\texttt{, LIST}_n\texttt{)} \mid \texttt{pair(NUMBER, LIST}_n\texttt{)} \mid \texttt{first(LIST}_\ell\texttt{)} \mid \texttt{...} \\
\texttt{NUMBER} &\rightarrow \texttt{first(LIST}_n\texttt{)} \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{...} \mid \texttt{9} \\
\texttt{PREDICATE}_n &\rightarrow \texttt{is-even} \mid \texttt{is-odd} \mid \texttt{is-prime} \mid \texttt{...} \\
\texttt{PREDICATE}_\ell &\rightarrow \texttt{is-empty} \mid \texttt{is-singleton} \mid \texttt{...}
\end{aligned}$$

This grammar has both lists ($\texttt{LIST}_n$) and lists of lists ($\texttt{LIST}_\ell$) but also contains many nearly redundant primitives, the same problem we saw with `even-or-prime`. The $\texttt{LIST}_\ell$ and $\texttt{LIST}_n$ versions of `filter`, for example, behave nearly identically. They both apply a predicate to each element in a list and keep only those elements for which the predicate is true. This problem only grows worse if we want lists of characters, lists of lists of characters, lists of predicates, lists of lists of lists, and so on. It would be convenient to have a single primitive for `filter` that could apply to all lists. We have looked so far at what are known as simple types (Pierce, 2002). Polymorphic type systems like the Hindley-Milner system[5] make this possible by adding variables to the type system itself. Instead of having $\texttt{LIST}_n$ and $\texttt{LIST}_\ell$, we have a single list type: `LIST x`. `x` is a variable that gets filled in with a concrete type like `NUMBER`. This approach allows us to provide a generic version of `filter`, for example, which takes a function from `x` to `BOOL` as its predicate and a `LIST x` as input. If `x` is `NUMBER`, it will filter a list of numbers. If `x` is `LIST NUM`, it will filter a list of list of numbers.

One reason that types are useful is that they allow us to encode information about how programs behave. When we have a program of type `NUMBER`, we know that it will evaluate to a number. As type systems grow more complex, they allow us to pack more information into the types. For example, consider the primitive `last`, which returns the last element of a list, and the primitive `length`, which returns the length of a list. Under simple types, these both take a `LIST` and return a `NUMBER`. With polymorphic types, however, we see that `length` takes a `LIST x` and returns a `NUMBER` while `last` takes a `LIST x` and returns an `x`. The polymorphic types contain more information and so differentiate `last` and `length`: they tell us that no matter what kind of list we give as input, `length` will return a `NUMBER` while `last` will always return the kind of thing contained in the input list. Packing more information into types is useful because it allows for a more complete description of what programs should be considered as potential solutions to a given problem. A more complete description is useful because it strengthens the learner's inductive bias, allowing it to rule out large groups of programs it would otherwise have needed to consider.

Other kinds of type systems can add even more information than polymorphic types. Type systems with typeclasses, for instance, allow you to state that a primitive applies to any type (i.e. using a type variable) but then restrict that type to a class of types for which other specific primitives are guaranteed to be implemented. For example, you might say that a `LIST x` can be sorted so long as type `x` has a

[5]Hindley-Milner is a type system for what are technically known as parameterically polymorphic types. Ad hoc polymorphism and subtyping are two other kinds of type polymorphism with different abilities and benefits.

comparison operator. Dependent types go so far as to encode entire first-order logical expressions in the type. The dependent type for a sorting function over `LIST x` can not only require that `x` have a comparison operator, but it can also state that the output list will be a permutation of the input list such that each successive element is greater than or equal to its predecessor. The program learning literature describes many techniques for leveraging complex type systems (Polikarpova, Kuraj, & Solar-Lezama, 2016; Osera & Zdancewic, 2015; Chlipala, 2022).

19.2.4 Recursion and conditionals

Often we will also be interested in modeling how people learn recursive functions. Recursion is hypothesized to be a core operation of human thought and language (Hauser, Chomsky, & Fitch, 2002; Corballis, 2014). It is also central in computer science because it allows complex problem to be solved with concise and elegant methods that break one problem into simpler sub-problems. For example, if we wanted to include recursive sorting algorithms in our list example, we would need the ability to call the currently defined function $\mathcal{F}$ on new input.

We define the primitive `recurse` which calls the function in which it is used. In this sense, `recurse` is a little unusual because the value it returns is not determined beforehand by a defined primitive. Its return value must be computed on the fly by the current definition of $\mathcal{F}$, meaning that its value will depend on its context of use. `recurse` will always have the same input type and output type as $\mathcal{F}$. In our example, `recurse` will take and return a list, so we could write functions like

$$\mathcal{F}\texttt{(lst)=pair(first(lst), pair(1, recurse(rest(lst))))} \tag{19.14}$$

This function is intended to insert a 1 in every other location—for example mapping $[5, 6, 7]$ to $[5, 1, 6, 1, 7, 1]$. But it is actually not a correct implementation because there is no end to the recursion: even when given an empty list, (19.14) function will try to call recurse on `rest(lst)`, which is another empty list. The recursion will never terminate and so the program will never return a value.

One solution is to introduce an `if` statement that recurses only when a condition is met. We can then write

$$\mathcal{F}\texttt{(lst)=pair(first(lst), pair(1, if(is-empty(lst), }\epsilon\texttt{, recurse(rest(lst)))))} \tag{19.15}$$

which only recurses when `lst` is not empty. The behavior of `if` is actually somewhat subtle. When we call a function in most programming languages, we evaluate its arguments first, so `filter(is-even, reverse(lst))` first calls `reverse(lst)` and then passes the value of that expression as the second argument to `filter`. If we do that with `if` in (19.15), we will *first* call `recurse(rest(lst))`, which will lead to another infinite recursion. The solution that programming languages have developed is that `if` is a special function that, unlike most other functions, does *not* evaluate all its arguments before being called. Instead, it evaluates the Boolean condition and then evaluates only one of its arguments, depending on the Boolean outcome. This is why `if` is a special keyword in most programming languages.[6]

19.2.5 Stochastic operations

Stochastic primitives are often useful when modeling the language of thought, because we sometimes want to model learners who themselves conceive of the world as being partially random. For example, learners might look at a very long sequence like

11001001000011111101101010100010001000010110100011000010001101001100

[6]Most languages also treat `and` and `or` specially via the related "short-circuit evaluation" where `and` only evaluates its second argument if the first is true; `or` only evaluates its second argument if the first is false. Lazy evaluation is an alternative way to execute programs that removes the need for short-circuit evaluation and other "special forms" (Abelson & Sussman, 1996).

and conceive of it as "random" when they cannot find a pattern in it, regardless of whether there actually is a pattern (like base-two digits of π). Such a sequence might therefore have a very concise hypothesis ("concatenate random digits") even though the data underlying it is complex or may look incompressible.

One easy way to introduce stochastic functions into the representation language is to add a function `flip` that returns "flips a coin" to determine whether to return true or false. `flip` is quite a different kind of operation from the ones listed above because now a program no longer necessarily has a single output. For example, the simple program

$$\mathcal{F}\texttt{(lst)=pair(if(flip(),1,2), lst)} \tag{19.16}$$

will return a distribution over two different lists: half the time a 1 appended onto the front of `lst`, and half the time a 2. Actually, we can think about (19.16) in a few different ways. We could think of it as a program that samples a random answer each time it is run. We could also think of it as implicitly specifying or representing a particular probability distribution. Similarly, different implementations will handle stochastic operations like `flip` in different ways. For example, some might store partial evaluations of programs and their associated probabilities. When encountering a `flip`, they push *both* outcomes onto a queue of incomplete traces to continue evaluating later. Other examples of stochastic operations might sample an element from a discrete set (e.g., an alphabet) or a continuous set (a distribution).

Importantly, once stochastic operations interface with operations like recursion, things can become more complex. For example,

$$\mathcal{F}\texttt{(lst)=pair(if(flip(),1,head(lst)), recurse(rest(lst)))} \tag{19.17}$$

will take a list like $[1, 2, 3, 4]$ and flip a coin to determine whether each element should be replaced by 1. The distribution of outcomes therefore assigns nonzero probability to sequences like $[1, 2, 3, 4]$, $[1, 1, 3, 4]$, $[1, 2, 1, 4]$, $[1, 2, 3, 1]$, $[1, 1, 1, 4]$, $[1, 1, 1, 1]$, etc. In this kind of model, the probability of the data is tied to the probability that the program took a particular route in its evaluation. For example, if the program maps

$$[1, 2, 3, 4] \xrightarrow{\mathcal{F}} [1, 2, 1, 4] \tag{19.18}$$

then this data would have a probability of $(1/2)^3$ since it doesn't matter how the first coin flip comes out (either way will put a 1 at the start of the list), but the remaining three have to come out specific ways in order to generate the output.

Many languages with stochastic operations, including Church (Goodman, Mansinghka, Roy, Bonawitz, & Tenenbaum, 2012), also include **memoization**. Memoization lets functions remember the outcome of specific coin flips or other random choices. To illustrate, imagine that we wanted to replace all of the 2s in a list with either 3 or 4 (but randomly one or the other). You might imagine writing a function like

```
1 F(lst) = if(flip(),
2              pair(if(equals(first(lst),2), 3), recurse(rest(lst))),
3              pair(if(equals(first(lst),2), 4), recurse(rest(lst)))).
```

This function would not be correct because each time `recurse` is called, a *different* value of `flip` would be computed. So this would make a new random choice in each recursive step, potentially picking different ones each time. One solution is to have a function `mem` that "remembers" the random choices made. For example, `mem(flip)` would be a function that chooses a value the first time it is called, and remembers that value the later times it is called, even across recursions. In this case, we could write

$$\mathcal{F}\texttt{(lst) = pair(if(first(lst)==2, if(mem(flip)(),2,3), rest(lst)), recurse(rest(lst))).} \tag{19.19}$$

19.3 Likelihoods for program models

To use a Bayesian model for program learning, we need more than the prior defined by the grammar over programs in our hypothesis space. We also need to specify a likelihood $P(d \mid h)$ describing how probable the data d would be if generated using program h. One basic challenge is faced in searching spaces of deterministic programs which is that each program only return a single output for a given input. Thus, most programs generate incorrect output. For example, we would have a very low probability of sampling a hypothesis from the prior and correctly predicting (19.3) above. One common way to address this is to "smooth" the likelihood by introducing noise into $P(d \mid h)$—namely assuming that the output is computed by first running the program for h and then randomly altering the output in some way. Adding such noise is useful because hypotheses that are approximately correct can receive partial credit for getting some of the output correct. This in turn provides a signal that helps search algorithms move through the space of programs in a way that improves fit to the data. Indeed, the choice of a helpful likelihood is often one of the most important choices for efficient inference.

Noise can be added to the likelihood in many ways. For example, we might assume in the domain of list functions that each element in an output list is corrupted by randomly sampling from $\{0, 1, 2, \ldots, 9\}$ with some small probability, perhaps 0.05. In this case, if the observed data list output was $[9, 7, 6, 3, 3, 2, 1]$ but the program predicted $[9, 7, 5, 3, 3, 2, 1]$, then the data would have probability

$$\left(0.95 + \frac{0.05}{10}\right)^6 \cdot \frac{0.05}{10} \tag{19.20}$$

since 6 of 7 data points could have been unchanged in the noise (probability 0.95) or changed via noise to their same values (probability 0.05/10), and one value, 5, had to be changed to its value. This computation illustrates that when computing the probability of noisy data, one must sum both the probability that it was generated by the hypothesis and the probability that it was generated by noise.

Noise in the above likelihood has no way to change the length of the list. It would therefore assign zero probability to $[9, 7, 6, 3, 3, 2, 1]$ if the hypothesis predicted $[9, 7, 6, 3, 3, 2]$ or $[9, 7, 6, 3, 3, 2, 1, 8]$. This situation is problematic because it means that the hypothesis must get the length right on *every* output—which is unlikely to occur by chance—or else the entire dataset is assigned zero probability. We can, however define other likelihoods that change the length. For example, Kashyap and Oommen (1984) define a probabilistic edit likelihood that allows one to compute the probability that one list is corrupted to another according to a series of insertions, deletions, and alterations, essentially a probabilistically coherent version of string edit (Levenshtein) distance. A faster but similar option, used in Yang and Piantadosi (2022), is to consider a noise model that first deletes from the end of the hypothesis' generated sequence and then appends randomly onto the end. Note that, as above, any string can come from any other by deleting everything, but hypotheses which share a prefix with the data will require fewer deletions and thus be assigned higher likelihood. This likelihood can not only be computed efficiently but also leads models to prefer to get the beginnings of lists correct. This ordering bias likely helps random search find hypotheses that match the initial stages of a recursive computation.

In settings other than lists or strings, the likelihood takes on a different form. For example, Piantadosi (2011) examined the learning of quantifier terms like "every" and "most", and showed that a program learning model could acquire programs for these terms. Quantifier terms are often formalized as functions on sets: "most A are B" (i.e. `most(A,B)`) is true if

$$|A \cap B| \; > \; |A \setminus B|. \tag{19.21}$$

In other words, "most barbers are happy" if the set of happy barbers (intersection of happy people and barbers) has greater cardinality than the set of non-happy barbers (set-difference between barbers and happy people). We therefore formulated a fairly standard hypothesis space with operations over sets and

cardinalities. When we first considered how to set up the likelihood of this model, it seemed plausible to use a likelihood which gave high probability to utterances that were most often true—e.g., the probability for a given data point (utterance in a context) would be high if the hypothesis evaluated it to be true, and false otherwise. We learned that this approach is doomed because the model will invariably learn meanings which are trivially true. For example, instead of learning a meaning like (19.21), the model might learn that $most(A, B) := true$. While it is tempting to try to rule out these trivial meanings somehow in the grammar, it won't work because a model can always write down expressions which are complex but trivially true (e.g., $|A \cup B| \geq |A|$). The example illustrates that a likelihood usually does need to be a proper probability distribution. Our initial likelihood was not quantifying the probability of the data in any sense. The problem was fixed by using a likelihood which evaluated all possible quantifiers in each context, and, with high probability, chose uniformly from those that were true. This size-principle likelihood (Tenenbaum, 2000; Tenenbaum & Griffiths, 2001, see Chapter 3) encourages the model to make quantifiers true in every context where they are possible, but penalizes them for being true in situations where they do not occur.

19.4 Computability concerns

When Turing first defined computability using Turing machines, he also showed that there is no algorithm to decide whether a given Turing machine will stop running and halt (Turing, 1936). There is, however, a much more powerful version of non-computability known as **Rice's theorem** (Rice, 1953) which holds that essentially no question about what a program does is computable (more technically: nontrivial "semantic properties" of programs are undecidable). No algorithm, for example, can take an arbitrary program as input and always determine whether that program ever outputs the number 5. No algorithm can decide if a program outputs only ascending numbers; or computes the function $f(x) = x^2$. That doesn't prevent us from gathering evidence about programs one way or the other—for example, we might run a program for a long time and see what it does—but Rice's theorem establishes that there is no procedure for definitely answering any questions like these that concern the output of the program.

Such theorems seem ominous for Bayesian inference over programs. When we hypothesize a program h, we are unsure whether h will output the observed data because this question cannot generally be answered. Consequently, it seems as though we cannot compute the likelihood because no algorithm can even compute the outputs! But, it is worth noting that psychologically, people may not face the same computability challenges in all domains where they learn programs. For example, if someone learns a program by observation—watching someone execute the steps to tying shoelaces or computing a derivative—then their task is substantially easier than if they were trying to induce an unseen algorithm. Cultural transmission of knowledge allows more complex representations—including programs (Thompson, Opheusden, Sumers, & Griffiths, 2022)—to be passed down with less learning difficulty, resulting in different forms of learning (from others vs. from nature respectively) (Chater & Christiansen, 2010).

In learning models that discover programs from data, there are several ways around computability concerns. One is to carefully construct the hypothesis space so that all programs are guaranteed to halt, but this can be difficult to do correctly. In this tradition, there are many computational or logical systems in which they key questions can be computed—these systems, though, necessarily, have less power than Turing machines. The most common solution is just to ignore the problem. While no algorithm can compute the e.g., likelihoods for all programs, many approximations seem to work fine in practice, likely because the programs in question tend to be relatively simple. If programs allow recursion or looping, then often when we run a program it will not halt in a reasonable amount of time. If a program doesn't halt after, say 1024 steps, we just consider it to have output the empty string or a null output. Or, we can solve this problem much more elegantly by using a prior that depends on the time and space resources that it uses. To illustrate the idea, if program h halts in time $\text{halt}(h)$, we might choose a prior proportional to $2^{-\text{halt}(h)}$. This favors programs that evaluate quickly and gives zero prior probability to

non-halting ($\text{halt}(h) = \infty$) programs. In Bayesian inference—in particular MCMC sampling—we could use this to reject non-halting programs because, as they run, their prior would eventually drop below an acceptance threshold. There are several formal versions of this idea centring on optimal efficient search through programs (Levin, 1973; Schmidhuber, 2002; Hutter, 2002). One simple intuition for search is possible in these spaces is to consider Levin Search (Levin, 1973), in which one interleaves computation of all programs. By dividing runtime resources between all possible programs, the non-halting programs do not prevent the halting programs from being computed. While these ideas are theoretically elegant, we have yet to find a domain where experiments with humans empirically distinguish speed-based priors from grammar-based priors.

19.5 Markov chain Monte Carlo for programs

The Metropolis-Hastings algorithm, a kind of Markov chain Monte Carlo (MCMC) algorithm (see Chapter 6), is the standard inference algorithm for program learning. Metropolis-Hastings needs us to specify how to compute a proposed new sample from the current one. The simplest approach is to sample de novo from the underlying grammar, but doing so fails to preserve any information about the current hypothesis. This means that such proposals lose information about what has worked well so far. We can keep this information by making proposals conditioned on the existing program. A standard way to do this is to choose just a subexpression of the current program and regenerate it from the grammar (Goodman et al., 2008). If the program is viewed as a tree, then this technique first chooses a subtree, and then replaces that subtree with a new subtree of the same type generated by sampling from the grammar. This scheme typically preserves most of the existing program (hopefully the useful parts) and changes just a small part of it (hopefully something that can be improved).

Consider a program like $\mathcal{F}$`(lst) = pair(first(lst), pair(first(lst), rest(lst)))` that duplicates the first element of the list. We can represent this as a tree

(19.22)

The standard Goodman et al. (2008) "regeneration" proposal would pick a subtree at random—we have denoted it here with the $*$—and generate a new proposal from the grammar by sampling a new subtree with the same type. For example, we might generate

(19.23)

in this case, proposing a new program that skips the second element of the list. In this proposal, we have only changed the program under the $*$ node. Note that it is possible to regenerate the entire program (e.g. a proposal from the prior) by proposing to the root. But if subtrees are chosen uniformly, then most of the time we will preserve some of the structure in the program.

The standard Metropolis-Hastings acceptance probability would then be

$$\min\left(1, \frac{P(h')P(d \mid h')}{P(h)P(d \mid h)} \cdot \frac{Q(h \mid h')}{Q(h' \mid h)}\right). \tag{19.24}$$

Some care must be taken in computing Q, the proposal probability because it is asymmetric. If s is the subtree in h that is replaced with s' in h', then

$$Q(h' \mid h) = \frac{P_{\mathcal{G}}(s')}{N(h)}$$

where $P_{\mathcal{G}}(s')$ is the probability of s' under grammar $\mathcal{G}$ and $N(h)$ is the total number of subtrees in h. Q here has multiplied the $1/N(h)$ chance of picking s in h times the probability of replacing s with s' according to $\mathcal{G}$.[7]

Tree-regeneration proposals are fast and simple, but many other proposal schemes are worth considering. Many of these alternatives are motivated by the observation that regeneration proposals destroy potentially useful information in the subtree being replaced. One way to address this issue is to also include **insert moves** which include the subtree being replaced as part of the newly generated subtree. For example, we might notice that `pair` takes a `NUMBER` and a `LIST` and returns a `LIST`. That means that if our tree had a `LIST` somewhere, like $\mathcal{F}$`(lst) = rest(rest(lst))`, we could select the (`rest(lst)`) and insert a `pair` node that re-uses the existing structure, as in

$$\mathcal{F}\texttt{(lst) = rest(pair(2,rest(lst)))}. \tag{19.25}$$

`rest(lst)` has been preserved from the initial hypothesis and we have inserted `pair(2,LIST)`. The **delete move** implements the mirror transformation of replacing a subtree with an appropriately typed subtree of itself. Other proposals that preserve structure include those that swap arguments (e.g. the two branches of an `if` statement) or swap functions while preserving their arguments (e.g. `and` for `or`). Ensuring that the parts being swapped have the appropriate types is especially important in these kinds of moves. One other kind of important proposal extracts subroutines out of an existing program, but these are difficult to implement.

19.6 Example model runs

Figure 19.2 shows a sample of 25 runs of MCMC over programs for a version of the number learning model in Piantadosi et al. (2012). This model captured children's rough developmental stages in learning to count, where they progress through a few intermediate levels of knowledge and behavioral ability before demonstrating full competence in saying what word goes with a given set of objects. The model captured the transitions between their intermediate knowledge states as the result of acquiring more data. With little data, the Bayesian inference would only justify short, simple programs that only worked partially, but as data accumulated, it would acquire a full counting procedure. Hypotheses in the model were set up as functions from sets to words, just like counting, and the model eventually discovered a recursive counting program,

$$\mathcal{F}(s)\texttt{ = if(equals(cardinality(s,1)),"one", next(recurse(setminus(s,select(s))))).} \tag{19.26}$$

[7]In computing a $Q(h' \mid h)$, we must actually sum over *all* the possible ways that h' can be derived from h (and symmetrically for $Q(h \mid h')$). In general, that will be a complex sum because there are many possible subtrees we could have proposed to in order to derive h', not just the one node that we did in fact choose. Conveniently, this "multiple path" problem can be ignored thanks to an auxiliary variable argument in which we imagine augmenting our hypothesis space with a variable saying which subtree to replace, which is sampled uniformly on each MCMC iteration. Conditioned on the auxiliary variable, there is only one proposal path.

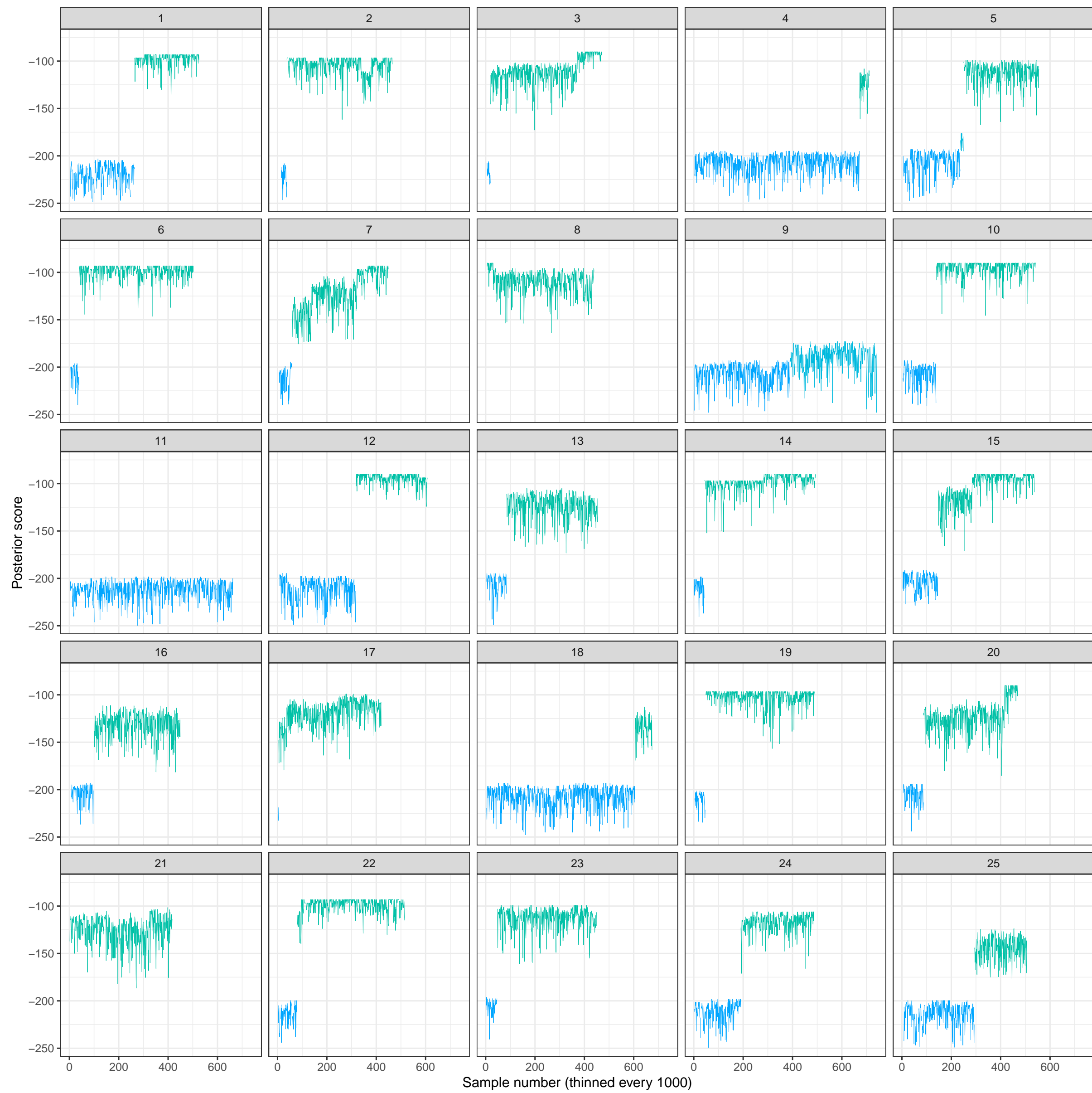

Figure 19.2: Example MCMC runs from a number learning model.

This program says the word "one" for sets containing one element, and otherwise it computes the word after (`next`) the word you get to by counting everything in s (the set to be counted) minus one element (`setminus(s,select(s))`). This emphasizes that in learning to count, children must come to understand how to update the set s to be counted through recursive or iterative processes that move them along the list of number words they know.

Figure 19.2 shows that MCMC with tree regeneration across 25 different models runs (subpanels), each run for 10 seconds using the Fleet software package. The horizontal axis in these plots shows the sample number (thinning by a factor of 1000) and the vertical axis shows the posterior score $\log(P(h)P(d \mid h))$ for the sampled hypothesis. Chains are colored according to the behavioral pattern they exhibit, or what numbers they get correct. This figure shows multiple runs to give a sense for what MCMC over programs is like. Roughly, within a run the MCMC chain will tend to linger on a mode—typically a program that behaves a certain way (same color). This is shown in the figure by the lines that jiggle up and down – these tend to be small, neutral proposals to a program that might slightly change their complexity but often won't affect the likelihood or functions that make a very minor difference to the likelihood, such as affecting performance on a rare data point. But sometimes, a chain will propose a change that leads to a substantial improvement in either prior (complexity) or fit (simplicity). When this happens, the posterior score takes a large jump upwards, and when this happens the model often changes behavior (color).

A good picture to have in mind is that abstract in "program space" there are modes of approximately equivalent programs that perform well, and the model improves by jumping from one mode to a better one. This jump is typically discrete because the programs are discrete, and so the different performance levels (posterior scores, vertical axis) are discrete. Because of this, most of the interesting changes that happen while sampling over program space tend to be large changes, rather than small incremental updates that one may be familiar with from MCMC over real-valued functions. Note, though, that a large change in performance or likelihood might actually correspond to a small change in the program itself. For example, we might replace `rest(lst)` with `recurse(rest(lst))` and change from a program that got nothing right to one that implements the correct recursion. Thus, the right thing to have in mind is that the modes in performance or likelihood may or may not be close to each other in program edit/proposal space. The mapping between the two is likely—possibly invariably—"brittle" so that little changes to a program can make or break it.

Random sampling is effective in this setup because a sampler, by definition, spends an amount of time on each hypothesis proportional to its posterior probability. This means that it proposes *more* from *better* hypotheses. In fact, the extent to which MCMC is better than e.g., enumeration tells us that there is some information about performance encoded into the proposal probabilities between programs.

Figure 19.3 shows a fancier version of this number model, from Piantadosi (under review). This version includes more primitives, such as ones for manipulating approximate quantity and noun types. This model also learns more complex programs because it derives operations on small sets (e.g., checking if a set has one element) from more basic operations like 1-to-1 matching. Despite this larger hypothesis space, the model is still able to sort out the computations involved in counting, including learning that numbers are not approximate, and that numbers don't refer to specific objects (e.g., "two" doesn't mean "two socks"). Visualizing this richer hypothesis space, Figure 19.3 shows a scatter plot of different hypotheses (points) plotted at their negative log prior (horizontal axis) and negative log likelihood (vertical axis). The best programs are the ones near the origin, corresponding to having a high prior and a high likelihood. The hypotheses in this figure are color according to their total posterior scores, and it illustrates how probability mass moves as more and more data point are accumulated. Initially, the best hypotheses are the ones which are close to the origin primarily along the horizontal axis (prior); with moderate amounts of data one might want the points to be close diagonally, minimizing both the prior and likelihood, and with lots of data it primarily matters if the values on the vertical axis are close to zero. Thus, learners could be expected to move down the left edge of these plots as data is accumulated.

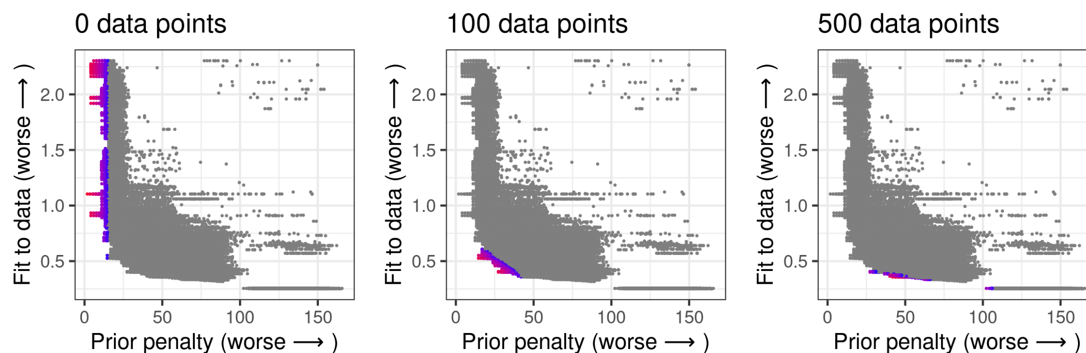

Figure 19.3: Priors vs. likelihood for a counting program model illustrating the tradeoff between teh two and the distribution of hypotheses in this space. Good models will be near the left-to-lower edge, meaning those whose prior can't be improved without a worse likelihood, or whose likelihood can't be improved without a worse prior.

It is important to remember though that the geometry of these plots are not themselves accessible to a learner. That is, from a hypothesis h it is not easy to determine which other hypotheses are approximately the same in terms of likelihood, because those hypotheses that are nearby in likelihood might require a big change in terms of the underlying program.

19.6.1 Two senses of learning

These figures illustrate how MCMC sampling progressively searches a hypothesis space. This kind of search may map onto human learning in at least two distinct ways. In some studies the program learning model is treated as an ideal-observer, a statistically rational solution to the problem of deciding what hypothesis a learner should choose. This approach explains **learning as accumulation of evidence**: learners change hypotheses (as in Figure 19.3) when they get more data. On this view, people perform approximate statistical inference at each amount of data; any non-adult-like hypotheses that early learners entertain are chosen because they represent the best solution given the available data. Several developmental effects have been modeled like this, including the shift from characteristic to defining features in conceptual development (Mollica & Piantadosi, 2021). This data-driven view also predicts other details of learning. For example, if learning is primarily about waiting for data, then that predicts a specific relationship between the mean and variance of the ages at which a given change occurs (Hidaka, 2013; Mollica & Piantadosi, 2017).

A potentially complementary approach treats **learning as search**. On this view, the primary limiting factor in development is how much of the hypothesis space learners have been able to explore. This idea seems to explain Figure 19.2 above, where the sampling model transitions between discrete hypotheses as the learner finds new solutions to a problem they have been considering. Ullman et al. (2012) argues for this point of view, pointing out that the drastic changes and reorganisations seen in search, often without additional evidence, mirror qualitative phenomena in child development. Likely, both data and search are important to capturing how children learn and revise their theories over time.

19.6.2 Inference hacks

Figure 19.2 shows 10 seconds of samples for each chain. In this time, for this model, Fleet will typically draw $400,000$–$600,000$ samples. The variability is because larger programs take longer to evaluate: the

chains which collect the most samples are the ones which did not achieve high posterior score. In this case, low posterior-score hypotheses are typically simpler (e.g. don't involve recursion) and so can be evaluated more quickly.

The discreteness of the hypothesis space has advantages and disadvantages. One primary disadvantage is that often a single sampling chain will not be able to escape its local minima and thus not find the best programs. This can be seen in Figure 19.2 as places where chains stay with low posterior scores. For programs, the best schemes seem to be parallel tempering schemes which run multiple chains at different temperatures and allow swapping between chains (see, e.g., Vousden, Farr, & Mandel, 2016; Earl & Deem, 2005). These methods are easy to implement, low-overhead, and allow MCMC sampling to mix between modes without requiring ad-hoc decisions about how long to run chains for or when to restart them.

One advantage of discrete spaces is that summary statistics about the posterior can be computed by storing the best hypotheses found and then explicitly computing the desired statistical quantities over them. For example, if $h_1, h_2, \ldots, h_{100}$ are the top 100 programs found, we may estimate a quantity like whether a given primitive is used by re-normalizing the posterior score on this finite set of hypotheses. For example, if $I_{\text{recurse}}(h_i)$ is whether or not h_i contains a recursion primitive, then we could approximate the posterior probability of using a recursion primitive as,

$$\sum_{i=1}^{100} \left(\frac{P(h_i)P(d \mid h_i)}{\sum_{j=1}^{100} P(h_j)P(d \mid h_j)} \right) \cdot I_{\text{recurse}}(h_i) \tag{19.27}$$

where the denominator of the first term is a normalizing term that is constant across all hypotheses h_i. The numerator is a hypothesis's posterior score, which is used to weight I_{recurse} (or equivalently add up the posterior probability of all hypotheses for which I_{recurse} is true). Estimation from the top hypotheses works because typically, nearly all of the posterior probability mass will be occupied by these highest-scoring hypotheses. By contrast, continuous spaces have no notion of the one hundred highest posterior hypotheses and must use samples directly.

One general approach that has been successful is to run parallel tempering over programs and store a discrete set of the top hypotheses found by any chain. When we compute learning curves or make predictions, we do so following Bayes rule but treating these top hypotheses as a fixed, finite hypothesis space. If we are interested in modeling developmental change as accumulation of evidence, we will run parallel tempering at various amounts of data and compute the fixed hypothesis space as the union of the top hypotheses found at any amount of data. The result is often on the order of thousands or tens of thousands of hypotheses. These numbers are trivial to process as a discrete, finite space for Bayesian inference—e.g. plotting learning curves or computing posterior predictive distributions—but sampling good hypotheses for this set is still a hard inference problem. In other words, the discreteness of the space allows us to partially separate things we want to compute from the posterior from the task of approximating that posterior with a finite set of hypotheses. Importantly, when we consider learning models with tens of thousands of hypotheses, we don't necessarily mean that learners represent them all. Learners with smart inference techniques (see below) might only represent a handful of hypotheses. Often, we as modelers want to ensure that there aren't possible hypotheses that we have missed, and so finding a large set of plausible hypotheses represents a kind of ideal observer model rather than a process model of learners' actual experiences.

19.7 Future directions

This chapter has reviewed several key ideas that form the basis of how we can think about modeling developmental processes as program learning. We now give a flavor for how these ideas can be extended and combined, turning to a few ways they have been explored in recent research.

19.7.1 Learning reusable components

The picture of program learning developed above focuses on acquiring a single program, typically a single function. One important extension to this idea is that of learning a library of functions that can be called as subroutines (Ellis et al., 2021, 2020; Dechter, Malmaud, Adams, & Tenenbaum, 2013; Talton et al., 2012; Cheyette & Piantadosi, 2017; Bowers et al., 2023). Intuitively, the ability to define subroutines extends the set of available primitives, which significantly impacts the inductive bias of most program learners. For example, we might want to represent a concept like "every other element in a list":

$$\texttt{every-other(lst) := if(empty(lst),}\epsilon\texttt{,pair(first(lst),recurse(rest(rest(lst)))))} \tag{19.28}$$

Having this primitive might allow us to easily learn a program like "all the odd elements followed by all the even elements":

$$\mathcal{F}\texttt{(lst) = append(every-other(lst),every-other(rest(lst))).} \tag{19.29}$$

Without `every-other`, $\mathcal{F}$ has a much longer description length (approximately as long as (19.29) plus twice (19.28)).[8] Thus, it can be especially useful to learn subroutines in the **multitask** setting, i.e. when a learner is trying to acquire more than one program. For example, imagine a program learner trying to solve thirty distinct problems that all involved picking out subsets of a list. Assume that the initial grammar, unlike the grammars we discussed earlier, did not include any sort of `filter` operation. In this case, it would be useful to define a shared library including `filter` and perhaps a set of common predicates.

Library learning poses an interesting statistical problem of how a learner should decide what to include in their library. One intuitive solution to this problem is to minimize total description length, i.e. the length of all the learned programs plus the length of each subroutine. Such a solution balances the cost of storing a subroutine against its potential to be reused many times. A number of mathematical models formalize this intuition (e.g., O'Donnell, 2015; Goodman et al., 2008; Ziv & Lempel, 1978), but because there are so many possible libraries to consider, these solutions are often difficult to apply in practice. One approximate technique that works well in simple settings is to run multiple MCMC chains, each of which is constrained to a fixed library structure (e.g. one chain might have a library of three subroutines and require each one to be called; another chain might have just one) (Yang & Piantadosi, 2022). The branch of program induction known as inductive logic programming has developed several other rule-based approaches to acquiring subroutines, which is known there as **predicate invention** (e.g. Muggleton, Lin, & Tamaddoni-Nezhad, 2015,).

An alternative approach that is particularly useful for multitask learners is to grow the library over time. For example, a learner might start with an initial grammar and solve 10% of the assigned problems with its initial grammar. It can examine these solutions to find repeated code, extract the repeated sections into subroutines, and add the new subroutines to its grammar. This newly extended grammar might now allow it to solve 20% of the problems, at which point it can look for new chunks of repeated code, and repeat the process. By iterating in this way several times, a learner can develop an increasingly useful library of subroutines and eventually solve very difficult problems which would be impractical to solve using the original grammar. The DreamCoder algorithm (Ellis et al., 2021, 2020) applies a refined version of this idea and is perhaps the most successful library learning model available as of this writing.

19.7.2 Learning church-encodings

There is an important sense in which program-learning models can create fundamentally new systems of knowledge, as is likely required for understanding human learning. Piantadosi (2021) considered program

[8] One word of warning: discussions about library learning often use confusing terminology. In one sense, it is correct to think about subroutines like `every-other` as new *primitives* since they are used in the grammar just like any other primitive. In another sense, `every-other` is a *derived* program that is itself defined only in terms of built-in operations.

learning over logical systems of **church encodings** (Pierce, 2002). A church encoding is an idea from logic and computer science where one can construct something in one logical system to mirror the dynamics of another (note that a "church encoding" is distinct from the "Church" programming language discussed in Chapter 18). This work showed, for example, how to learn representations like number, Boolean logic, quantification, dominance hierarchies, lists, etc. by encoding these into a minimalist underlying logic. To illustrate how this might work in lambda calculus, consider a mapping from the terms of Boolean logic (true, false, and, or, not) to "pure" lambda calculus (e.g., lambda calculus with only lambda terms):

$$\begin{aligned} true &:= \lambda a.\lambda b.a \\ false &:= \lambda c.\lambda d.d \\ and &:= \lambda p.\lambda q.pqp \\ or &:= \lambda r.\lambda s.rrs \\ not &:= \lambda t.\lambda u.\lambda v.tvu \end{aligned} \tag{19.30}$$

The definitions in (19.30) should be thought of as encoding a "program" for each symbol in Boolean logic. But importantly, these programs aren't defined in terms of underlying primitives that have inherent content (like `CAUSE` or `GO`). These symbols get mapped to essentially just a syntactic construction in lambda calculus. However, they allow us to follow the rules of lambda calculus to compute the answer to a question, like "what is the result of `or(true,false)`"? To do this, we substitute the lambda terms into this expression and evaluate them according to the rules of lambda calculus:

$$\begin{aligned} \texttt{or(true,false)} &:= ((\lambda r.\lambda s.rrs)(\lambda a.\lambda b.a)(\lambda c.\lambda d.d)) \\ &= ((\lambda s.(\lambda a.\lambda b.a)(\lambda a.\lambda b.a)s)(\lambda c.\lambda d.d)) \\ &= ((\lambda a.\lambda b.a)(\lambda a.\lambda b.a)(\lambda c.\lambda d.d)) \\ &= ((\lambda b.(\lambda a.\lambda b.a))(\lambda c.\lambda d.d)) \\ &= (\lambda a.\lambda b.a) \end{aligned} \tag{19.31}$$

The result is the representation for `true`, which is the correct answer. Alternatively, we could compute `or(false,false)`,

$$\begin{aligned} \texttt{or(false,false)} &:= ((\lambda r.\lambda s.rrs)(\lambda c.\lambda d.d)(\lambda c.\lambda d.d)) \\ &= ((\lambda s.(\lambda c.\lambda d.d)(\lambda c.\lambda d.d)s)(\lambda c.\lambda d.d)) \\ &= ((\lambda c.\lambda d.d)(\lambda c.\lambda d.d)(\lambda c.\lambda d.d))) \\ &= (\lambda d.d)(\lambda c.\lambda d.d))) \\ &= (\lambda c.\lambda d.d) \end{aligned} \tag{19.32}$$

The result is, correctly, `false`. The computations are tedious but profound: the dynamics of how lambda expressions evaluate have resulted in computing the correct outcome for an expression in Boolean logic. This is true for all propositions that we can compute in Boolean logic, e.g., `and(not(or(true,false)),not(true))` will evaluate to the correct answer when the corresponding lambda terms are substituted in for `and`, `or`, `not`, `true`, `false`.

This happens even though Boolean logic is not "built in" to the syntax of lambda calculus, at least not explicitly so. In other words, (19.30) provides a way to *encode* Boolean logic into lambda calculus, to trick the default dynamics of lambda calculus into representing some other system. This is one way in which minimal programming systems might come to represent richer systems of knowledge. Success in getting lambda calculus—or any of dozens of other systems—to encode a new domain is essentially always possible: lambda calculus is a Turing-complete system, and so any computation or computer program we wish can be compiled into an compositions of lambda calculus with no other primitives.

Learners learning church-encoding programs will therefore create new logical systems without needing primitives beyond e.g., the syntax of lambda calculus. Piantadosi (2021) considers such learners as a

developmental model using an even simpler system than lambda calculus, combinatory logic, which can also be more straightforwardly encoded into a neural network. Church encoding potentially answers deep questions about development because it shows how a Turing-complete system could be constructed without "building in" primitive concepts that human infants may not possess, like numbers, Boolean logic, family trees, simplest games, and other key structures. All of these can be discovered as structured computations, allowing learners to build systems of logic and computation as models that explain their observations of the world.

19.7.3 Learning like a hacker

Rule, Tenenbaum, and Piantadosi (2020)'s "Hacker-like" (HL) model has also explored resources other than simple sampling and search that learners might draw on in inference. the basic intuition is that when we observe data like

$$[3,6,1,7,3,2,9] \xrightarrow{\mathcal{F}} [3,7,1,8,3,3,9] \tag{19.33}$$

we might notice some suspicious relationships between the input and the outputs. For example, the two are the same length, which suggests that operations like `map` may be likely operations. Second, we might notice that when the output numbers differ from the input, they only differ by 1. And then we might notice that the only ones which differ are every other element in the list. Learning models can formalize these inferences using a grammar of inference steps rather than primitives. Rule et al.'s model maintains not just a hypothesized program, but a series of transformations that convert the input into the output. For example, one transformation would be to simply memorize the correct output; another might abstract a lambda function that maps +1 over the appropriate list elements, etc. Rule et al. show that this learning model dramatically out-performs random sampling in terms of the number of hypotheses it must actively consider. It often requires on the order of 50-100 hypotheses, bringing it much closer to human-scale in terms of search requirements. Notably, this model also provides a closer fit than alternatives to large-scale human experiments on list functions.

19.7.4 Learning over term-rewriting

In addition to these search moves, the HL model learns over a different representation than standard program primitives or lambda calculus. It learns programs over **term-rewriting** systems which allow the learner to state explicit rules about how a structured computation proceeds. To illustrate the idea of term rewriting, consider a basic tranformation that children learn in algebra class like,

$$a \cdot (x+y) = a \cdot x + a \cdot y. \tag{19.34}$$

or

$$e^{x+y} = e^x \cdot e^y. \tag{19.35}$$

These are learned as an explicit rule about manipulating equations, which are themselves tree-structured. For example, learners who acquire these rules would be able to manipulating a symbolic structure,

$$e^{a\cdot(x+y)} \to e^{a\cdot x + a \cdot y} \to e^{a\cdot x} \cdot e^{a \cdot y}. \tag{19.36}$$

Term rewriting thinks of this sequence of steps as formalizing the computation itself, where the system of rules that are used are the program. This computation may halt at some point, when no more transformations can be applied, or loop forever, just like a program. Typically these systems allow for nondeterminism, meaning that multiple rules can be applied at each point, and therefore evaluation requires exploring the tree of possible derivations (much like including `flip` requires exploring the space of execution paths). Importantly, term-rewriting systems have the full power of Turing machines.

HL considers learners who have a grammar for writing such transformations and must find a collection of them that capture the knowledge required in a given domain. For example, you might construct a theory of numbers in this domain that looks like how number is formalized in mathematical logic (e.g., the Peano axioms; Peano, 1889):

$$\begin{aligned} a + 0 &= a \\ a + S(b) &= S(a + b) \\ a \cdot 0 &= 0 \\ a \cdot S(b) &= a + (a \cdot b). \end{aligned} \tag{19.37}$$

These rules describe purely syntactic transformations of structures made out of symbols. On their own, no individual symbol has meaning. There is no sense, for example, in which + symbolizes addition other than the role which it plays in the rules above. A learner might acquire other rules, however, that places + in different roles. For example, if we were to swap $\cdot$ and + in the rules above, the system would still describe a theory of numbers, but one using $\cdot$ for addition and + for multiplication. If we were instead to include rules like $a+a = S(a)$ or $S(S(a)) = S(a)$, we would have an interesting computational system, but one that no longer captured a theory of natural number. The challenge for learners, then, is to find rules consistent with the dynamics of some domain (e.g explaining data in that domain). When this happens, term rewriting systems serve to define both the key structures and processes in that domain.

Term rewriting can do everything that the church-encoding models can do because lambda expressions (or equivalent systems) can be evaluated according to term-rewriting rules. But the theories and representations learned in term-rewriting often feel more intuitive because their dynamics are defined transparently in the way the terms rewrite, rather than obscurely encoded through the dynamics of another system like lambda calculus. One aspirational hope of these theories is that they will permit program learning models to acquire real systems of knowledge, like those required for reasoning in different domains and manipulating essentially arbitrary structures. One other useful feature of term rewriting systems is that it is easy to change the set of symbols over which transformations are defined. Unlike church-encoding, symbols do not need to be defined in terms of some base system before they can be used. Simply by adding rules which use a new symbol, it begins to become part of the networks of computations that the system describes. Removing a symbol from the rules similarly removes it from the system. In this sense, a hypothesis space defined over the set of term rewriting systems is capable of genuine conceptual change.

The resulting systems of knowledge end up looking more like rich cognitive theories. As was thought for some time in artificial intelligence work, program learning over large sets of propositional rules may be a productive way towards humanlike knowledge. The programming language Prolog for example works in a fundamentally different way than most of the programs considered here in that its programs are declarations of logical relationships. These programs determine the behavior of a backtracking search that, essentially, explores logical consequences of the initial statements. Term-rewriting and Prolog-like representations may help program learning move towards acquiring richer systems of knowledge, beyond single programs, and Prolog representations have been extensively considered in inductive inference (Muggleton, 1995; Muggleton & De Raedt, 1994).

19.8 Conclusion

This chapter discussed the essential ideas behind modeling learning as programming. This approach accounts for the breadth of conceptual structures that people appear to readily internalize and closely links studies of human learning to theories in artificial intelligence, machine learning, and probabilistic inference. The computational ideas we have outlined—including grammars over programs, random sampling, church-encoding, etc.—provide a compelling toolset for refining debates about learning and development

which have typically only relied on informal, philosophical notions. They are also empirically effective: program learning models explain many aspects of learning across a wide variety of domains. Together, these results show that Bayesian inference over programs is a powerful tool for understanding human learning.

References

Abelson, H., & Sussman, G. (1996). *Structure and interpretation of computer programs.* MIT Press.

Amalric, M., Wang, L., Pica, P., Figueira, S., Sigman, M., & Dehaene, S. (2017). The language of geometry: Fast comprehension of geometrical primitives and rules in human adults and preschoolers. *PLoS Computational Biology*, *13*(1), e1005273.

Aspray, W. (2000). *Computing before computers.* Iowa State University Press.

Babbage, C. (1864). *Passages from the life of a philosopher.* Longman.

Blackburn, P., & Bos, J. (2005). *Representation and inference for natural language: A first course in computational semantics.* Center for the Study of Language and Information.

Bowers, M., Olausson, T. X., Wong, L., Grand, G., Tenenbaum, J. B., Ellis, K., & Solar-Lezama, A. (2023). Top-down synthesis for library learning. *Proceedings of the ACM on Programming Languages*, *7*(POPL), 1182–1213.

Chater, N., & Christiansen, M. H. (2010). Language acquisition meets language evolution. *Cognitive Science*, *34*(7), 1131–1157.

Cheyette, S., & Piantadosi, S. (2017). Knowledge transfer in a probabilistic language of thought. In *Proceedings of the 39th Annual Meeting of the Cognitive Science Society.*

Chlipala, A. (2022). *Certified programming with dependent types: A pragmatic introduction to the Coq proof assistant.* MIT Press.

Corballis, M. C. (2014). *The recursive mind.* Princeton University Press.

Dechter, E., Malmaud, J., Adams, R. P., & Tenenbaum, J. B. (2013). Bootstrap learning via modular concept discovery. In *Proceedings of the 29th International Joint Conference in Artificial Intelligence (IJCAI)* (pp. 1302–1309).

Depeweg, S., Rothkopf, C. A., & Jäkel, F. (2018). Solving Bongard problems with a visual language and pragmatic reasoning. *arXiv preprint arXiv:1804.04452.*

Earl, D. J., & Deem, M. W. (2005). Parallel tempering: Theory, applications, and new perspectives. *Physical Chemistry Chemical Physics*, *7*(23), 3910–3916.

Ellis, K. (2020). *Algorithms for learning to induce programs.* Unpublished doctoral dissertation, Massachusetts Institute of Technology.

Ellis, K., Morales, L., Sablé-Meyer, M., Solar-Lezama, A., & Tenenbaum, J. (2018). Learning libraries of subroutines for neurally–guided Bayesian program induction. In *Advances in Neural Information Processing Systems* (pp. 7815–7825).

Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Morales, L., Hewitt, L., Cary, L., Solar-Lezama, A., & Tenenbaum, J. B. (2021). Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (pp. 835–850).

Ellis, K., Wong, L., Nye, M., Sable-Meyer, M., Cary, L., Morales, L., Hewitt, L., Solar-Lezama, A., & Tenenbaum, J. B. (2020). Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning.

Fodor, J. A. (1975). *The language of thought.* Harvard University Press.

Fodor, J. A., & Pylyshyn, Z. W. (1988). Connectionism and cognitive architecture: A critical analysis. *Cognition*, *28*(1-2), 3–71.

Fredkin, E., & Toffoli, T. (1982). Conservative logic. *International Journal of Theoretical Physics*, *21*(3), 219–253.

Goodman, N., Mansinghka, V., Roy, D. M., Bonawitz, K., & Tenenbaum, J. B. (2012). Church: a language for generative models. *arXiv preprint arXiv:1206.3255.*

Goodman, N. D., & Frank, M. C. (2016). Pragmatic language interpretation as probabilistic inference. *Trends in Cognitive Sciences*, *20*(11), 818–829.

Goodman, N. D., & Lassiter, D. (2015). Probabilistic semantics and pragmatics: Uncertainty in language and thought. In S. Lappin & C. Fox (Eds.), *The handbook of contemporary semantic theory* (2nd ed.). Wiley-Blackwell.

Goodman, N. D., Tenenbaum, J. B., Feldman, J., & Griffiths, T. L. (2008). A rational analysis of rule-based concept learning. *Cognitive Science*, *32*(1), 108–154.

Goodman, N. D., Tenenbaum, J. B., & Gerstenberg, T. (2014). *Concepts in a probabilistic language of thought* (Tech. Rep.). Center for Brains, Minds and Machines (CBMM).

Goodman, N. D., Ullman, T. D., & Tenenbaum, J. B. (2011). Learning a theory of causality. *118*(1), 110–119.

Graves, A., Wayne, G., & Danihelka, I. (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401.*

Hauser, M. D., Chomsky, N., & Fitch, W. T. (2002). The faculty of language: what is it, who has it, and how did it evolve? *Science*, *298*(5598), 1569–1579.

Heim, I., & Kratzer, A. (1998). *Semantics in generative grammar.* Wiley-Blackwell.

Hidaka, S. (2013). A computational model associating learning process, word attributes, and age of acquisition. *PLoS One*, *8*(11), e76242.

Hutter, M. (2002). The fastest and shortest algorithm for all well-defined problems. *International Journal of Foundations of Computer Science*, *13*(03), 431–443.

Kashyap, R. L., & Oommen, B. J. (1984). Spelling correction using probabilistic methods. *Pattern Recognition Letters*, *2*(3), 147–154.

Kemp, C. (2009). Quantification and the language of thought. In *Advances in Neural Information Processing Systems 22.*

Kemp, C. (2012). Exploring the conceptual universe. *Psychological Review*, *119*(4), 685–722.

Kemp, C., & Tenenbaum, J. B. (2008). The discovery of structural form. *Proceedings of the National Academy of Sciences*, *105*(31), 10687–10692.

Kemp, C., Tenenbaum, J. B., Niyogi, S., & Griffiths, T. L. (2010). A probabilistic model of theory formation. *Cognition*, *114*, 165-196.

Kim, J. Z., & Bassett, D. S. (2022). A neural programming language for the reservoir computer. *arXiv preprint arXiv:2203.05032*.

Lake, B. M., & Piantadosi, S. T. (2020). People infer recursive visual concepts from just a few examples. *Computational Brain & Behavior*, *3*(1), 54–65.

Lake, B. M., Salakhutdinov, R., & Tenenbaum, J. B. (2015). Human-level concept learning through probabilistic program induction. *Science*, *350*(6266), 1332–1338.

Levin, L. A. (1973). Universal sequential search problems. *Problemy Peredachi Informatsii*, *9*(3), 115–116.

Lovelace, A. A. (1842). Sketch of the analytical engine invented by charles babbage, by lf menabrea, officer of the military engineers, with notes upon the memoir by the translator. *Taylor's Scientific Memoirs*, *3*, 666–731.

Mollica, F., & Piantadosi, S. T. (2017). How data drive early word learning: A cross-linguistic waiting time analysis. *Open Mind*, *1*(2), 67–77.

Mollica, F., & Piantadosi, S. T. (2021). Logical word learning: The case of kinship. *Psychonomic Bulletin & Review*, 1–34.

Muggleton, S. (1995). Inverse entailment and progol. *New Generation Computing*, *13*(3), 245–286.

Muggleton, S., & De Raedt, L. (1994). Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, *19*, 629–679.

Muggleton, S. H., Lin, D., & Tamaddoni-Nezhad, A. (2015). Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, *100*(1), 49–73.

O'Donnell, T. J. (2015). *Productivity and reuse in language: A theory of linguistic computation and storage.* MIT Press.

Osera, P.-M., & Zdancewic, S. (2015). Type-and-example-directed program synthesis. *ACM SIGPLAN Notices*, *50*(6), 619–630.

Overlan, M., Jacobs, R., & Piantadosi, S. T. (2017). Learning abstract visual concepts via probabilistic program induction in a language of thought. *Cognition*, *168*, 320–334.

Peano, G. (1889). *Arithmetices principia: nova methodo.* Fratres Bocca.

Pérez, J., Marinković, J., & Barceló, P. (2019). On the Turing completeness of modern neural network architectures. *arXiv preprint arXiv:1901.03429*.

Piantadosi, S. T. (2011). *Learning and the language of thought.* Unpublished doctoral dissertation, Massachusetts Institute of Technology.

Piantadosi, S. T. (2021). The computational origin of representation. *Minds and machines*, *31*(1), 1–58.

Piantadosi, S. T. (under review). The algorithmic origins of counting.

Piantadosi, S. T., & Jacobs, R. A. (2016). Four problems solved by the probabilistic language of thought. *Current Directions in Psychological Science*, *25*(1), 54–59.

Piantadosi, S. T., Tenenbaum, J. B., & Goodman, N. D. (2012). Bootstrapping in a language of thought: A formal model of numerical concept learning. *Cognition*, *123*(2), 199-217.

Piantadosi, S. T., Tenenbaum, J. B., & Goodman, N. D. (2016). The logical primitives of thought: Empirical foundations for compositional cognitive models. *Psychological Review*, *123*(4), 392.

Pierce, B. C. (2002). *Types and programming languages.* MIT Press.

Planton, S., Kerkoerle, T. van, Abbih, L., Maheu, M., Meyniel, F., Sigman, M., Wang, L., Figueira, S., Romano, S., & Dehaene, S. (2021). A theory of memory for binary sequences: Evidence for a mental compression algorithm in humans. *PLoS Computational Biology*, *17*(1), e1008598.

Polikarpova, N., Kuraj, I., & Solar-Lezama, A. (2016). Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, *51*(6), 522–538.

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, *74*(2), 358–366.

Rothe, A., Lake, B. M., & Gureckis, T. (2017). Question asking as program generation. In *Advances in Neural Information Processing Systems* (pp. 1046–1055). Curran Associates.

Rule, J. S., Piantadosi, S. T., Cropper, A., Ellis, K., Nye, M., & Tenenbaum, J. B. (under review). Human-level and human-like: an efficient model of symbolic learning.

Rule, J. S., Tenenbaum, J. B., & Piantadosi, S. T. (2020). The child as hacker. *Trends in Cognitive Sciences*, *24*(11), 900–915.

Schmidhuber, J. (2002). The speed prior: a new simplicity measure yielding near-optimal computable predictions. In *International Conference on Computational Learning Theory* (pp. 216–228).

Siskind, J. (1996). A computational study of cross-situational techniques for learning word-to-meaning mappings. *Cognition*, *61*, 31–91.

Steedman, M. (2000). *The syntactic process.* MIT Press.

Talton, J., Yang, L., Kumar, R., Lim, M., Goodman, N. D., & Měch, R. (2012). Learning design patterns with Bayesian grammar induction. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* (pp. 63–74).

Tenenbaum, J. B. (2000). Rules and similarity in concept learning. In *Advances in Neural Information Processing Systems 12.*

Tenenbaum, J. B., & Griffiths, T. L. (2001). Generalization, similarity, and Bayesian inference. *Behavioral and Brain Sciences*, *24*, 629-641.

Thompson, B., Opheusden, B. van, Sumers, T., & Griffiths, T. (2022). Complex cognitive algorithms preserved by selective social learning in experimental populations. *Science*, *376*(6588), 95–98.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math*, *58*(345-363), 5.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, *59*(236), 433–460.

Ullman, T. D., Goodman, N. D., & Tenenbaum, J. B. (2012). Theory learning as stochastic search in the language of thought. *Cognitive Development*, *27*(4), 455–480.

Vousden, W., Farr, W. M., & Mandel, I. (2016). Dynamic temperature selection for parallel tempering in Markov chain Monte Carlo simulations. *Monthly Notices of the Royal Astronomical Society*, *455*(2), 1919–1937.

Wolfram, S. (2002). *A new kind of science.* Wolfram Media.

Yang, Y., & Piantadosi, S. T. (2022). One model for the learning of language. *Proceedings of the National Academy of Sciences.*

Ziv, J., & Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, *24*(5), 530–536.