An Introduction to Probabilistic Programming

Jan-Willem van de Meent
College of Computer and Information Science
Northeastern University
j.vandemeent@northeastern.edu

Brooks Paige
Alan Turing Institute
University of Cambridge
bpaige@turing.ac.uk

Hongseok Yang
School of Computing
KAIST
hongseok.yang@kaist.ac.kr

Frank Wood
Department of Computer Science
University of British Columbia
fwood@cs.ubc.ca

Contents

Abstract

This document is designed to be a first-year graduate-level introduction to probabilistic programming. It not only provides a thorough background for anyone wishing to use a probabilistic programming system, but also introduces the techniques needed to design and build these systems. It is aimed at people who have an undergraduate-level understanding of either or, ideally, both probabilistic machine learning and programming languages.

We start with a discussion of model-based reasoning and explain why conditioning as a foundational computation is central to the fields of probabilistic machine learning and artificial intelligence. We then introduce a simple first-order probabilistic programming language (PPL) whose programs define static-computation-graph, finite-variable-cardinality models. In the context of this restricted PPL we introduce fundamental inference algorithms and describe how they can be implemented in the context of models denoted by probabilistic programs.

In the second part of this document, we introduce a higher-order probabilistic programming language, with a functionality analogous to that of established programming languages. This affords the opportunity to define models with dynamic computation graphs, at the cost of requiring inference methods that generate samples by repeatedly executing the program. Foundational inference algorithms for this kind of probabilistic programming language are explained in the context of

an interface between program executions and an inference controller.

This document closes with a chapter on advanced topics which we believe to be, at the time of writing, interesting directions for probabilistic programming research; directions that point towards a tight integration with deep neural network research and the development of systems for next-generation artificial intelligence applications.

Acknowledgements

We would like to thank the very large number of people who have read through preliminary versions of this manuscript. Comments from the reviewers have been particularly helpful, as well as general interactions with David Blei and Kevin Murphy in particular. Some people we would like to individually thank are, in no particular order, Tobias Kohn, Rob Zinkov, Marcin Szymczak, Gunes Baydin, Andrew Warrington, Yuan Zhou, and Celeste Hollenbeck, as well as numerous other members of Frank Wood's Oxford and UBC research groups graciously answered the call to comment and contribute.

We would also like to acknowledge colleagues who have contributed intellectually to our thinking about probabilistic programming. First among these is David Tolpin, whose work with us at Oxford decisively shaped the design of the Anglican probabilistic programming language, and forms the basis for the material in Chapter 6. We would also like to thank Josh Tenenbaum, Dan Roy, Vikash Mansinghka, and Noah Goodman for inspiration, periodic but important research interactions, and friendly competition over the years. Chris Heunen, Ohad Kammar and Sam Staton helped us to understand subtle issues about the semantics of higher-order probabilistic programming languages. Lastly we would like to thank Mike Jordan for asking us to do this, providing the impetus to collate everything we thought we learned while having put together a NIPS tutorial years ago.

During the writing of this manuscript the authors received generous support from various granting agencies. Most critically, while all of the authors were at Oxford together, three of them were explicitly supported at various times by the DARPA under its Probabilistic Programming for Advanced Machine Learning (PPAML) (FA8750-14-2-0006) program. Jan-Willem van de Meent was additionally supported by startup funds from Northeastern University. Brooks Paige and Frank Wood were additionally supported by the Alan Turing Institute under the EPSRC grant EP/N510129/1. Frank Wood was also supported by Intel, DARPA via its D3M (FA8750-17-2-0093) program, and NSERC via its Discovery grant program. Hongseok Yang was supported by the Engineering Research Center Program through the National Research Foundation of Korea (NRF) funded by the Korean Government MSIT (NRF-2018R1A5A1059921), and also by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7068177).

Notation

Grammars

$c ::=$ A constant value or primitive function.
$v ::=$ A variable.
$f ::=$ A user-defined procedure.

$e ::=$ c | v | (let [v e_1] e_2) | (if e_1 e_2 e_3) | (f e_1 ... e_n)
| (c e_1 ... e_n) | (sample e) | (observe e_1 e_2)
An expression in the first-order probabilistic programming language (FOPPL).

$E ::=$ c | v | (if E_1 E_2 E_3) | (c E_1 ... E_n)
An expression in the (purely deterministic) target language.

$e ::=$ c | v | f | (if e e e) | (e e_1 ... e_n)
| (sample e) | (observe e e) | (fn [v_1 ... v_n] e)
An expression in the higher-order probabilistic programming language (HOPPL).

$q ::=$ e | (defn f [v_1 ... v_n] e) q
A program in the FOPPL or the HOPPL.

Sets, Lists, Maps, and Expressions

$C = \{c_1, \ldots, c_n\}$	A set of constants ($c_i \in C$ refers to elements).
$C = (c_1, \ldots, c_n)$	A list of constants (C_i indexes elements c_i).

$\mathcal{C} = [v_1 \mapsto c_1, \ldots, v_n \mapsto c_n]$	A map from variables to constants ($\mathcal{C}(v_i)$ indexes entries c_i).
$\mathcal{C}' = \mathcal{C}[v_i \mapsto c_i']$	A map update in which $\mathcal{C}'(v_i) = c_i'$ replaces $\mathcal{C}(v_i) = c_i$.
$\mathcal{C}(v_i) = c_i'$	An in-place update in which $\mathcal{C}(v_i) = c_i'$ replaces $\mathcal{C}(v_i) = c_i$.
$C = \text{dom}(\mathcal{C}) = \{v_1, \ldots, v_n\}$	The set of keys in a map.
$E =$ `(* v v)`	An expression literal.
$E' = E[v := c] =$ `(* c c)`	An expression in which a constant c replaces the variable v.
FREE-VARS(e)	The free variables in an expression.

Directed Graphical Models

$G = (V, A, \mathcal{P}, \mathcal{Y})$	A directed graphical model.
$V = \{v_1, \ldots, v_{\lvert V\rvert}\}$	The variable nodes in the graph.
$Y = \text{dom}(\mathcal{Y}) \subseteq V$	The observed variable nodes.
$X = V \setminus Y \subseteq V$	The unobserved variable nodes.
$y \in Y$	An observed variable node.
$x \in X$	An unobserved variable node.
$A = \{(u_1, v_1), \ldots, (u_{\lvert A\rvert}, v_{\lvert A\rvert})\}$	The directed edges (u_i, v_i) between parents $u_i \in V$ and children $v_i \in V$.
$\mathcal{P} = [v_1 \mapsto E_1, \ldots, v_{\lvert V\rvert} \mapsto E_{\lvert V\rvert}]$	The probability mass or density for each variable v_i, represented as a target language expression $\mathcal{P}(v_i) = E_i$
$\mathcal{Y} = [y_1 \mapsto c_1, \ldots, y_{\lvert Y\rvert} \mapsto c_{\lvert Y\rvert}]$	The observed values $\mathcal{Y}(y_i) = c_i$.
PA$(v) = \{u : (u, v) \in A\}$	The set of parents of a variable v.

Factor Graphs

$G = (V, F, A, \Psi)$	A factor graph.
$V = \{v_1, \ldots, v_{\lvert V\rvert}\}$	The variable nodes in the graph.
$F = \{f_1, \ldots, f_{\lvert F\rvert}\}$	The factor nodes in the graph.
$A = \{(v_1, f_1), \ldots, (v_{\lvert A\rvert}, f_{\lvert A\rvert})\}$	The undirected edges between variables v_i and factors f_i.
$\Psi = [f_1 \mapsto E_1, \ldots, f_{\lvert F\rvert} \mapsto E_{\lvert F\rvert}]$	Potentials for factors f_i, represented as target language expressions E_i.

Probability Densities

$p(Y, X) = p(V)$	The joint density over all variables.
$p(X)$	The prior density over unobserved variables.
$p(Y \mid X)$	The likelihood of observed variables Y given unobserved variables X.
$p(X \mid Y)$	The posterior density for unobserved variables X given unobserved variables Y.

$\mathcal{X} = [x_1 \mapsto c_1, \ldots, x_n \mapsto c_n]$	A trace of values $\mathcal{X}(x_i) = c_i$ associated with the instantiated set of variables $X = \mathrm{dom}(\mathcal{X})$.
$p(X = \mathcal{X}) = p(x_1 = c_1, \ldots, x_n = c_n)$	The probability density $p(X)$ evaluated at a trace $\mathcal{X}$.
$p_0(v_0 \,;\, c_1, \ldots, c_n)$	A probability mass or density function for a variable v_0 with parameters $c_1, \ldots, c_n$.
$P(v_0) = (p_0 \;\; v_0 \;\; c_1 \;\; \ldots \;\; c_n)$	The language expression that evaluates to the probability mass or density $p_0(v_0; c_1, \ldots, c_n)$.

1

Introduction

How do we engineer machines that reason? This is a question that has long vexed humankind. The answer to this question is fantastically valuable. There exist various hypotheses. One major division of hypothesis space delineates along lines of assertion: that random variables and probabilistic calculation are more-or-less an engineering requirement (Ghahramani, 2015; Tenenbaum et al., 2011) and the opposite (LeCun et al., 2015; Goodfellow et al., 2016). The field ascribed to the former camp is roughly known as Bayesian or probabilistic machine learning; the latter as deep learning. The first requires inference as a fundamental tool; the latter optimization, usually gradient-based, for classification and regression.

Probabilistic programming languages are to the former as automated differentiation tools are to the latter. Probabilistic programming is fundamentally about developing languages that allow the denotation of inference problems and evaluators that "solve" those inference problems. We argue that the rapid exploration of the deep learning, big-data-regression approach to artificial intelligence has been triggered largely by the emergence of programming language tools that automate the tedious and troublesome derivation and calculation of gradients for optimization.

Probabilistic programming aims to build and deliver a toolchain that does the same for probabilistic machine learning; supporting supervised, unsupervised, and semi-supervised inference. Without such a toolchain one could argue that the complexity of inference-based approaches to artificial intelligence systems are too high to allow rapid exploration of the kind we have seen recently in deep learning.

While such a next-generation artificial intelligence toolchain is of particular interest to the authors, the fact of the matter is that the probabilistic programming tools and techniques are already transforming the way Bayesian statistical analyses are performed. Traditionally the majority of the effort required in a Bayesian statistical analysis was in iterating model design where each iteration often involved a painful implementation of an inference algorithm specific to the current model. Automating inference, as probabilistic programming systems do, significantly lowers the cost of iterating model design leading to both a better overall model in a shorter period of time and all of the consequent benefits.

This introduction to probabilistic programming covers the basics of probabilistic programming from language design to evaluator implementation with the dual aim of explaining existing systems at a deep enough level that readers of this text should have no trouble adopting and using any of both the languages and systems that are currently out there and making it possible for the next generation of probabilistic programming language designers and implementers to use this as a foundation upon which to build.

This introduction starts with an important, motivational look at what a model is and how model-based inference can be used to solve many interesting problems. Like automated differentiation tools for gradient-based optimization, the utility of probabilistic programming systems is grounded in applications simpler and more immediately practical than futuristic artificial intelligence applications; building from this is how we will start.

1.1 Model-based Reasoning

Model-building starts early. Children build model airplanes then blow them up with firecrackers just to see what happens. Civil engineers build physical models of bridges and dams then see what happens in scale-model wave pools and wind tunnels. Disease researchers use mice as model organisms to simulate how cancer tumors might respond to different drug dosages in humans.

These examples show exactly what a model is: a stand-in, an imposter, an artificial construct designed to respond in the same way as the system you would like to understand. A mouse is not a human but it is often close enough to get a sense of what a particular drug will do at particular concentrations in humans anyway. A scale-model of an earthen embankment dam has the wrong relative granularity of soil composition but studying overtopping in a wave pool still tells us something about how an actual dam might respond.

As computers have become faster and more capable, numerical models have come to the fore and computer simulations have replaced physical models. Such simulations are by nature approximations. However, now in many cases they can be as exacting as even the most highly sophisticated physical models – consider that the US was happy to abandon physical testing of nuclear weapons.

Numerical models emulate stochasticity, i.e. using pseudorandom number generators, to simulate actually random phenomena and other uncertainties. Running a simulator with stochastic value generation leads to a many-worlds-like explosion of possible simulation outcomes. Every little kid knows that even the slightest variation in the placement of a firecracker or the most seemly minor imperfection of a glue joint will lead to dramatically different model airplane explosions. Effective stochastic modeling means writing a program that can produce all possible explosions, each corresponding to a particular set of random values, including for example the random final resting position of a rapidly dropped lit firecracker.

Arguably this intrinsic variability of the real world is the most significant complication for modeling and understanding. Did the mouse die in two weeks because of a particular individual drug sensitivity,

because of its particular phenotype, or because the drug regiment trial arm it was in was particularly aggressive? If we are interested in average effects, a single trial is never enough to learn anything for sure because random things almost always happen. You need a population of mice to gain any kind of real knowledge. You need to conduct several wind-tunnel bridge tests, numerical or physical, because of variability arising everywhere – the particular stresses induced by a particular vortex, the particular frailty of an individual model bridge or component, etc. Stochastic numerical simulation aims to computationally encompass the complete distribution of possible outcomes.

When we write model we generally will mean stochastic simulator and the measurable values it produces. Note, however, that this is not the only notion of model that one can adopt. Notably there is a related family of models that is specified solely in terms of an unnormalized density or "energy" function; this is treated in Chapter 3.

Models produce values for things we can measure in the real world; we call such measured values *observations*. What counts as an observation is model, experiment, and query specific – you might measure the daily weight of mice in a drug trial or you might observe whether or not a particular bridge design fails under a particular load.

Generally one does not observe every detail produced by a model, physical or numerical, and sometimes one simply cannot. Consider the standard model of physics and the large hadron collider. The standard model is arguably the most precise and predictive model ever conceived. It can be used to describe what can happen in fundamental particle interactions. At high energies these interactions can result in a particle jet that stochastically transitions between energy-equivalent decompositions with varying particle-type and momentum constituencies. It is simply not possible to observe the initial particle products and their first transitions because of how fast they occur. The energy of particles that make up the jet deposited into various detector elements constitute the observables.

So how does one use models? One way is to use them to falsify theories. To this one needs encode the theory as a model then simulate from it many times. If the population distribution of observations generated by the model is not in agreement with observations generated

by the real world process then there is evidence that the theory can be falsified. This describes science to a large extent. Good theories take the form of models that can be used to make testable predictions. We can test those predictions and falsify model variants that fail to replicate observed statistics.

Models also can be used to make decisions. For instance when playing a game you either consciously or unconsciously use a model of how your opponent will play. To use such a model to make decisions about what move to play next yourself, you simulate taking a bunch of different actions, then pick one amongst them by simulating your opponent's reaction according to your model of them, and so forth until reaching a game state whose value you know, for instance, the end of the game. Choosing the action that maximizes your chances of winning is a rational strategy that can be framed as model-based reasoning. Abstracting this to life being a game whose score you attempt to maximize while living requires a model of the entire world, including your own physical self, and is where model-based probabilistic machine learning meets artificial intelligence.

A useful model can take a number of forms. One kind takes the form of a reusable, interpretable abstraction with a good associated inference algorithm that describes summary statistic or features extracted from raw observable data. Another kind consists of a reusable but non-interpretable and entirely abstract model that can accurately generate complex observable data. Yet another kind of model, notably models in science and engineering, takes the form of a problem-specific simulator that describes a generative process very precisely in engineering-like terms and precision. Over the course of this introduction it will become apparent how probabilistic programming addresses the complete spectrum of them all.

All model types have parameters. Fitting these parameters, when few, can sometimes be performed manually, by intensive theory-based reasoning and a priori experimentation (the masses of particles in the standard model), by measuring conditional subcomponents of a simulator (the compressive strength of various concrete types and their action under load), or by simply fiddling with parameters to see which values produce the most realistic outputs.

Automated model fitting describes the process of using algorithms to determine either point or distributional estimates for model parameters and structure. Such automation is particularly useful when the parameters of a model are uninterpretable or many. We will return to model fitting in Chapter 7 however it is important to realize that inference can be used for model learning too, simply by lifting the inference problem to include uncertainty about the model itself (e.g. see the neural network example in 2.3 and the program induction example in 5.3).

The key point now is to understand that models come in many forms, from scientific and engineering simulators in which the results of every subcomputation are interpretable to abstract models in statistics and computer science which are, by design, significantly less interpretable but often are valuable for predictive inference none-the-less.

1.1.1 Model Denotation

An interesting thing to think about, and arguably the foundational idea that led to the field of probabilistic programming, is how such models are denoted and, respectively, how such models are manipulated to compute quantities of interest.

To see what we mean about model denotation let us first look at a simple statistical model and see how it is denoted. Statistical models are typically denoted mathematically, subsequently manipulated algebraically, then "solved" computationally. By "solved" we mean that an inference problem involving conditioning on the values of a subset of the variables in the model is answered. Such a model denotation stands in contrast to simulators which are often denoted in terms of software source code that is directly executed. This also stands in contrast, though less so, to generative models in machine learning which usually take the form of probability distributions whose factorization properties can be read from diagrams like graphical models or factor graphs.

Nearly the simplest possible model one could write down is a beta-Bernoulli model for generating a coin flip from a potentially biased coin.

Such a model is typically denoted

$$\begin{aligned} x &\sim \text{Beta}(\alpha, \beta) \\ y &\sim \text{Bernoulli}(x) \end{aligned} \tag{1.1}$$

where α and β are parameters, x is a latent variable (the bias of the coin) and y is the value of the flipped coin. A trained statistician will also ascribe a learned, folk-meaning to the symbol $\sim$ and the keywords Beta and Bernoulli. For example $\text{Beta}(a, b)$ means that, given the value of arguments a and b we can construct what is effectively an object with two methods. The first method being a probability density (or distribution) function that computes

$$p(x|a, b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1},$$

and the second a method that draws exact samples from said distribution. A statistician will also usually be able to intuit not only that some variables in a model are to be observed, here for instance y, but that there is an inference objective, here for instance to characterize $p(x|y)$. This denotation is extremely compact, and being mathematical in nature means that we can use our learned mathematical algebraic skills to manipulate expressions to solve for quantities of interest. We will return to this shortly.

In this tutorial we will generally focus on conditioning as the goal, namely the characterization of some conditional distribution given a specification of a model in the form of a joint distribution. This will involve the extensive use of Bayes rule

$$p(X|Y) = \frac{p(Y|X)p(X)}{p(Y)} = \frac{p(X,Y)}{p(Y)} = \frac{p(X,Y)}{\int p(X,Y)dX}. \tag{1.2}$$

Bayes rule tells us how to derive a conditional probability from a joint, conditioning tells us how to rationally update our beliefs, and updating beliefs is what learning and inference are all about.

The constituents of Bayes rule have common names that are well known and will appear throughout this text: $p(Y|X)$ the likelihood, $p(X)$ the prior, $p(Y)$ the marginal likelihood (or evidence), and $p(X|Y)$ the

Table 1.1: Probabilistic Programming Models

X	Y
scene description	image
simulation	simulator output
program source code	program return value
policy prior and world simulator	rewards
cognitive decision making process	observed behavior

posterior. For our purposes a model is the joint distribution $p(Y, X) = p(Y|X)p(X)$ of the observations Y and the random choices made in the generative model X, also called latent variables.

The subject of Bayesian inference, including both philosophical and methodological aspects, is in and of itself worthy of book length treatment. There are a large number of excellent references available, foremost amongst them the excellent book by Gelman et al. (2013). In the space of probabilistic programming arguably the recent books by Davidson-Pilon (2015) and Pfeffer (2016) are the best current references. They all aim to explain what we expect you to gain an understanding of as you continue to read and build experience, namely, that conditioning a joint distribution – the fundamental Bayesian update – describes a huge number of problems succinctly.

Before continuing on to the special-case analytic solution to the simple Bayesian statistical model and inference problem, let us build some intuition about the power of both programming languages for model denotation and automated conditioning by considering Table 1.1. In this table we list a number of X,Y pairs where denoting the joint distribution of $P(X, Y)$ is realistically only doable in a probabilistic programming language and the posterior distribution $P(X|Y)$ is of interest. Take the first, "scene description" and "image." What would such a joint distribution look like? Thinking about it as $P(X, Y)$ is somewhat hard, however, thinking about $P(X)$ as being some kind of distribution over a so-called scene graph – the actual object geometries, textures, and poses in a physical environment – is not unimaginably

hard, particularly if you think about writing a simulator that only needs to stochastically generate reasonably plausible scene graphs. Noting that $P(X, Y) = P(Y|X)P(X)$ then all we need is a way to go from scene graph to observable image and we have a complete description of a joint distribution. There are many kinds of renderers that do just this and, although deterministic in general, they are perfectly fine to use when specifying a joint distribution because they map from some latent scene description to observable pixel space and, with the addition of some image-level pixel noise reflecting, for instance, sensor imperfections or Monte-Carlo ray-tracing artifacts, form a perfectly valid likelihood.

An example of this "vision as inverse graphics" idea (Kulkarni et al., 2015b) appearing first in Mansinghka et al. (2013) and then subsequently in Le et al. (2017b,a) took the image Y to be a Captcha image and the scene description X to include the obscured string. In all three papers the point was not Captcha-breaking per se but instead demonstrating both that such a model is denotable in a probabilistic programming language and that such a model can be solved by general purpose inference.

Let us momentarily consider alternative ways to solve such a "Captcha problem." A non-probabilistic programming approach would require gathering a very large number of Captchas, hand-labeling them all, then designing and training a neural network to regress from the image to a text string (Bursztein et al., 2014). The probabilistic programming approach in contrast merely requires one to write a program that generates Captchas that are stylistically similar to the Captcha family one would like to break – a *model* of Captchas – in a probabilistic programming language. Conditioning such a model on its observable output, the Captcha image, will yield a posterior distribution over text strings. This kind of conditioning is what probabilistic programming evaluators do.

Figure 1.1 shows a representation of the output of such a conditioning computation. Each Captcha/bar-plot pair consists of a held-out Captcha image and a truncated marginal posterior distribution over unique string interpretations. Drawing your attention to the middle of the bottom row, notice that the noise on the Captcha makes it more-or-less impossible to tell if the string is "aG8BPY" or "aG8RPY." The posterior distribution

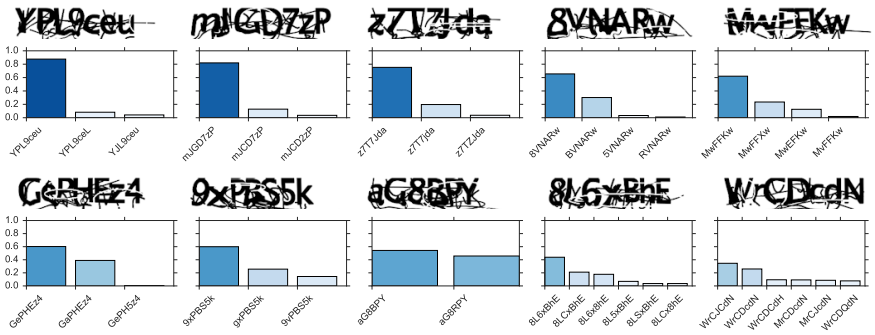

Figure 1.1: Posterior uncertainties after inference in a probabilistic programming language model of 2017 Facebook Captchas (reproduced from Le et al. (2017a))

$P(X|Y)$ arrived at by conditioning reflects this uncertainty.

By this simple example, whose source code appears in Chapter 5 in a simplified form, we aim only to liberate your thinking in regards to what a model is (a joint distribution, potentially over richly structured objects, produced by adding stochastic choice to normal computer programs like Captcha generators) and what the output of a conditioning computation can be like. What probabilistic programming languages do is to allow denotation of any such model. What this tutorial covers in great detail is how to develop inference algorithms that allow computational characterization of the posterior distribution of interest, increasingly very rapidly as well (see Chapter 7).

1.1.2 Conditioning

Returning to our simple coin-flip statistics example, let us continue and write out the joint probability density for the distribution on X and Y. The reason to do this is to paint a picture, by this simple example, of what the mathematical operations involved in conditioning are like and why the problem of conditioning is, in general, hard.

Assume that the symbol Y denotes the observed outcome of the coin flip and that we encode the event "comes up heads" using the mathematical value of the integer 1 and 0 for the converse. We will denote the bias of the coin, i.e. the probability it comes up heads, using

the symbol x and encode it using a real positive number between 0 and 1 inclusive, i.e. $x \in \mathbb{R} \cap [0, 1]$. Then using standard definitions for the distributions indicated by the joint denotation in Equation (1.1) we can write

$$p(x, y) = x^y(1 - x)^{1-y} \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1 - x)^{\beta-1} \tag{1.3}$$

and then use rules of algebra to simplify this expression to

$$p(x, y) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{y+\alpha-1}(1 - x)^{\beta-y}. \tag{1.4}$$

Note that we have been extremely pedantic here, using words like "symbol," "denotes," "encodes," and so forth, to try to get you, the reader, to think in advance about other ways one might denote such a model and to realize if you don't already that there is a fundamental difference between the symbol or expression used to represent or denote a meaning and the meaning itself. Where we haven't been pedantic here is probably the most interesting thing to think about: What does it mean to use rules of algebra to manipulate Equation (1.3) into Equation (1.4)? To most reasonably trained mathematicians, applying expression transforming rules that obey the laws of associativity, commutativity, and the like are natural and are performed almost unconsciously. To a reasonably trained programming languages person these manipulations are meta-programs, i.e. programs that consume and output programs, that perform semantics-preserving transformations on expressions. Some probabilistic programming systems operate in exactly this way (Narayanan et al., 2016). What we mean by semantics preserving in general is that, after evaluation, expressions in pre-simplified and post-simplified form have the same meaning; in other words, evaluate to the same object, usually mathematical, in an underlying formal language whose meaning is well established and agreed. In probabilistic programming semantics preserving generally means that the mathematical objects denoted correspond to the same distribution (Staton et al., 2016). Here, after algebraic manipulation, we can agree that, when evaluated on inputs x and y, the expressions in Equations (1.3) and (1.4) would evaluate to the same value and thus are semantically equivalent alternative denotations. In Chapter 7 we touch on some of the

challenges in defining the formal semantics of probabilistic programming languages.

That said, our implicit objective here is not to compute the value of the joint probability of some variables, but to do conditioning instead, for instance, to compute $p(x|y = \text{“heads}'')$. Using Bayes rule this is *theoretically* easy to do. It is just

$$p(x|y) = \frac{p(x,y)}{\int p(x,y)dx} = \frac{\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}x^{y+\alpha-1}(1-x)^{\beta-y}}{\int \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}x^{y+\alpha-1}(1-x)^{\beta-y}dx}. \tag{1.5}$$

In this special case the rules of algebra and semantics preserving transformations of integrals can be used to algebraically solve for an analytic form for this posterior distribution.

To start the preceding expression can be simplified to

$$p(x|y) = \frac{x^{y+\alpha-1}(1-x)^{\beta-y}}{\int x^{y+\alpha-1}(1-x)^{\beta-y}dx}. \tag{1.6}$$

which still leaves a nasty looking integral in the denominator. This is the complicating crux of Bayesian inference. This integral is in general intractable as it involves integrating over the entire space of the latent variables. Consider the Captcha example: simply summing over the latent character sequence itself would require an exponential-time operation.

This special statistics example has a very special property, called conjugacy, which means that this integral can be performed by inspection, by identifying that the integrand is the same as the non-constant part of the beta distribution and using the fact that the beta distribution must sum to one

$$\int x^{y+\alpha-1}(1-x)^{\beta-y}dx = \frac{\Gamma(\alpha+y)\Gamma(\beta-y+1)}{\Gamma(\alpha+\beta+1)}. \tag{1.7}$$

Consequently,

$$p(x|y) = \text{Beta}(\alpha+y, \beta-y+1), \tag{1.8}$$

which is equivalent to

$$x|y \sim \text{Beta}(\alpha+y, \beta-y+1). \tag{1.9}$$

There are several things that can be learned about conditioning from even this simple example. The result of the conditioning operation is a *distribution* parameterized by the observed or given quantity. Unfortunately this distribution will in general not have an analytic form because, for instance, we usually won't be so lucky that the normalizing integral has an algebraic analytic solution nor, in the case that it is not, will it usually be easily calculable.

This does not mean that all is lost. Remember that the $\sim$ operator is overloaded to mean two things, density evaluation and exact sampling. Neither of these are possible in general. However the latter, in particular, can be approximated, and often consistently even without being able to do the former. For this reason amongst others our focus will be on sampling-based characterizations of conditional distributions in general.

1.1.3 Query

Either way, having such a handle on the resulting posterior distribution, density function or method for drawing samples from it, allows us to ask questions, "queries" in general. These are best expressed in integral form as well. For instance, we could ask: what is the probability that the bias of the coin is greater than 0.7, given that the coin came up heads? This is mathematically denoted as

$$p(x > 0.7|y = 1) = \int \mathbb{I}(x > 0.7)p(x|y = 1)dx \tag{1.10}$$

where $\mathbb{I}(\cdot)$ is an indicator function which evaluates to 1 when its argument takes value true and 0 otherwise, which in this instance can be directly calculated using the cumulative distribution function of the beta distribution.

Fortunately we can still answer queries when we only have the ability to sample from the posterior distribution owing to the Markov strong law of large numbers which states under mild assumptions that

$$\lim_{L\to\infty} \frac{1}{L} \sum_{\ell=1}^{L} f(X^\ell) \to \int f(X)p(X)dX, \qquad X^\ell \sim p(X), \tag{1.11}$$

for general distributions p and functions f. This technique we will exploit repeatedly throughout. Note that the distribution on the right

hand side is approximated by a set of L samples on the left and that different functions f can be evaluated at the same sample points chosen to represent p after the samples have been generated.

This more or less completes the small part of the computational statistics story we will tell, at least insofar as how models are denoted then algebraically manipulated. We highly recommend that unfamiliar readers interested in the fundamental concepts of Bayesian analysis and mathematical evaluation strategies common there to read and study the "Bayesian Data Analysis" book by Gelman et al. (2013).

The field of statistics long-ago, arguably first, recognized that computerized systemization of the denotation of models and evaluators for inference was essential and so developed specialized languages for model writing and query answering, amongst them BUGS (Spiegelhalter et al., 1995) and, more recently, STAN (Stan Development Team, 2014). We could start by explaining these and only these languages but this would do significant injustice to the emerging breadth and depth of the the field, particularly as it applies to modern approaches to artificial intelligence, and would limit our ability to explain, in general, what is going on under the hood in all kinds of languages not just those descended from Bayesian inference and computational statistics in finite dimensional models. What is common to all, however, is inference via conditioning as the objective.

1.2 Probabilistic Programming

The Bayesian approach, in particular the theory and utility of conditioning, is remarkably general in its applicability. One view of probabilistic programming is that it is about automating Bayesian inference. In this view probabilistic programming concerns the development of syntax and semantics for languages that denote conditional inference problems and the development of corresponding evaluators or "solvers" that computationally characterize the denoted conditional distribution. For this reason probabilistic programming sits at the intersection of the fields of machine learning, statistics, and programming languages, drawing on the formal semantics, compilers, and other tools from programming languages to build efficient inference evaluators for models and applica-

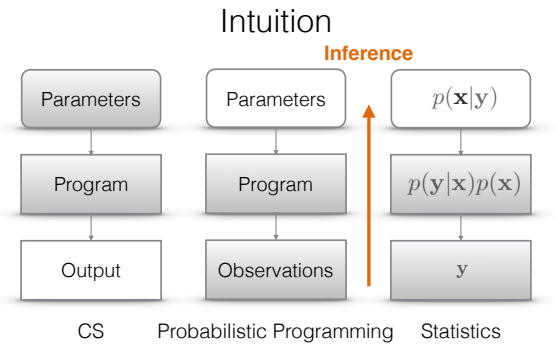

Figure 1.2: Probabilistic programming, an intuitive view.

tions from machine learning using the inference algorithms and theory from statistics.

Probabilistic programming is about doing statistics using the tools of computer science. Computer science, both the theoretical and engineering discipline, has largely been about finding ways to efficiently evaluate programs, given parameter or argument values, to produce some output. In Figure 1.2 we show the typical computer science programming pipeline on the left hand side: write a program, specify the values of its arguments or situate it in an evaluation environment in which all free variables can be bound, then evaluate the program to produce an output. The right hand side illustrates the approach taken to modeling in statistics: start with the output, the observations or data Y, then specify a usually abstract generative model $p(X, Y)$, often denoted mathematically, and finally use algebra and inference techniques to characterize the posterior distribution, $p(X \mid Y)$, of the unknown quantities in the model given the observed quantities. Probabilistic programming is about performing Bayesian inference using the tools of computer science: programming language for model denotation and statistical inference algorithms for computing the conditional distribution of program inputs that could have given rise to the observed program output.

Thinking back to our earlier example, reasoning about the bias of a coin is an example of the kind of inference probabilistic programming systems do. Our data is the outcome, heads or tails, of one coin flip. Our model, specified in a forward direction, stipulates that a coin and its bias is generated according to the hand-specified model then the coin flip outcome is observed and analyzed under this model. One challenge, the writing of the model, is a major focus of applied statistics research where "useful" models are painstakingly designed for every new important problem. Model learning also shows up in programming languages taking the name of program induction, machine learning taking the form of model learning, and deep learning, particularly with respect to the decoder side of autoencoder architectures. The other challenge is computational and is what Bayes rule gives us a theoretical framework in which to calculate: to computationally characterize the posterior distribution of the latent quantities (e.g. bias) given the observed quantity (e.g. "heads" or "tails"). In the beta-Bernoulli problem we were able to analytically derive the form of the posterior distribution, in effect, allowing us to transform the original inference problem denotation into a denotation of a program that completely characterizes the inverse computation.

When performing inference in probabilistic programming systems, we need to design algorithms that are applicable to any program that a user could write in some language. In probabilistic programming the language used to denote the generative model is critical, ranging from intentionally restrictive modeling languages, such as the one used in BUGS, to arbitrarily complex computer programming languages like C, C++, and Clojure. What counts as observable are the outputs generated from the forward computation. The inference objective is to computationally characterize the posterior distribution of all of the random choices made during the forward execution of the program given that the program produces a particular output.

There are subtleties, but that is a fairly robust intuitive definition of probabilistic programming. Throughout most of this tutorial we will assume that the program is fixed and that the primary objective is inference in the model specified by the program. In the last chapter we will talk some about connections between probabilistic programming

and deep learning, in particular through the lens of semi-supervised learning in the variational autoencoder family where parts of or the whole generative model itself, i.e. the probabilistic program or "decoder," is also learned from data.

Before that, though, let us consider how one would recognize or distinguish a probabilistic program from a non-probabilistic program. Quoting Gordon et al. (2014), "probabilistic programs are usual functional or imperative programs with two added constructs: the ability to draw values at random from distributions, and the ability to *condition* values of variables in a program via observations." We emphasize conditioning here. The meaning of a probabilistic program is that it simultaneously denotes a joint and conditional distribution, the latter by syntactically indicating where conditioning will occur, i.e. which random variable values will be observed. Almost all languages have pseudo-random value generators or packages; what they lack in comparison to probabilistic programming languages is syntactic constructs for conditioning and evaluators that implement conditioning. We will call languages that include such constructs probabilistic programming languages. We will call languages that do not but that are used for forward modeling stochastic simulation languages or, more simply, programming languages.

There are many libraries for constructing graphical models and performing inference; this software works by programmatically constructing a data structure which represents a model, and then, given observations, running graphical model inference. What distinguishes between this kind of approach and probabilistic programming is that a program is used to construct a model as a data structure, rather than considering the "model" that arises implicitly from direct evaluation of the program expression itself. In probabilistic programming systems, either a model data structure is constructed explicitly via a non-standard interpretation of the probabilistic program itself (if it can be, see Chapter 3), or it is a general Markov model whose state is the evolving evaluation environment generated by the probabilistic programming language evaluator (see Chapter 4). In the former case, we often perform inference by compiling the model data structure to a density function (see Chapter 3), whereas in the latter case, we employ

methods that are fundamentally generative (see Chapters 4 and 6).

1.2.1 Existing Languages

The design of any tutorial on probabilistic programming will have to include a mix of programming languages and statistical inference material along with a smattering of models and ideas germane to machine learning. In order to discuss modeling and programming languages one must choose a language to use in illustrating key concepts and for showing examples. Unfortunately there exist a very large number of languages from a number of research communities; programming languages: Hakaru (Narayanan et al., 2016), Augur (Tristan et al., 2014), R2 (Nori et al., 2014), Figaro (Pfeffer, 2009), IBAL (Pfeffer, 2001)), PSI (Gehr et al., 2016); machine learning: Church (Goodman et al., 2008), Anglican (Wood et al., 2014a) (updated syntax (Wood et al., 2015)), BLOG (Milch et al., 2005), Turing.jl (Ge et al., 2018), BayesDB (Mansinghka et al., 2015), Venture (Mansinghka et al., 2014), Probabilistic-C (Paige and Wood, 2014), webPPL (Goodman and Stuhlmüller, 2014), CPProb (Casado, 2017), (Koller et al., 1997), (Thrun, 2000); and statistics: Biips (Todeschini et al., 2014), LibBi (Murray, 2013), Birch (Murray et al., 2018), STAN (Stan Development Team, 2014), JAGS (Plummer, 2003), BUGS (Spiegelhalter et al., 1995)[1].

In this tutorial we will not attempt to explain each of the languages and catalogue their numerous similarities and differences. Instead we will focus on the concepts and implementation strategies that underlie most, if not all, of these languages. We will highlight one extremely important distinction, namely, between languages in which all programs induce models with a finite number of random variables and languages for which this is not true. The language we choose for the tutorial has to be a language in which a coherent shift from the former to the latter is possible. For this and other reasons we chose to write the tutorial using an abstract language similar in syntax and semantics to Anglican. Anglican is similar to WebPPL, Church, and Venture and is essentially a Lisp-like language which, by virtue of its syntactic simplicity, also makes for efficient and easy meta-programming, an approach many

[1] sincere apologies to the authors of any languages left off this list

implementors will take. That said the real substance of this tutorial is entirely language agnostic and the main points should be understood in this light.

We have left off of the preceding extensive list of languages both one important class of language – probabilistic logic languages ((Kimmig et al., 2011),(Sato and Kameya, 1997) – and sophisticated, useful, and widely deployed libraries/embedded domain-specific languages for modeling and inference (Infer.NET (Minka et al., 2010a), Factorie (McCallum et al., 2009), Edward (Tran et al., 2017), PyMC3 (Salvatier et al., 2016)). One link between the material presented in this tutorial and these additional languages and libraries is that the inference methodologies we will discuss apply to advanced forms of probabilistic logic programs (Alberti et al., 2016; Kimmig and De Raedt, 2017) and, in general, to the graph representations constructed by such libraries. In fact the libraries can be thought of as compilation targets for appropriately restricted languages. In the latter case strong arguments can be made that these are also languages in the sense that there is an (implicit) grammar, a set of domain-specific values, and a library of primitives that can be applied to these values. The more essential distinction is the one we have structured this tutorial around, that being the difference between static languages in which the denoted model can be compiled to a finite-node graphical model and dynamic languages in which no such compilation can be performed.

1.3 Example Applications

Before diving into specifics, let us consider some motivating examples of what has been done with probabilistic programming languages and how phrasing things in terms of a model plus conditioning can lead to elegant solutions to otherwise extremely difficult tasks.

We argue that, besides the obvious benefits that derive from having an evaluator that implements inference automatically, the main benefit of probabilistic programming is having additional expressivity, significantly more compact and readable than mathematical notation, in the modeling language. While it is possible to write down the mathematical formalism for a model of latents X and observables Y for each of the

examples shown in Table 1.1, doing so is usually neither efficient nor helpful in terms of intuition and clarity. We have already given one example, Captcha from earlier in this chapter. Let us proceed to more.

Constrained Simulation

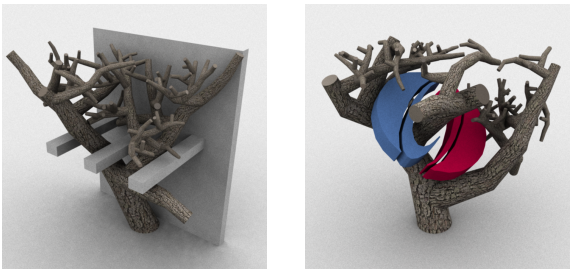

Figure 1.3: Posterior samples of procedurally generated, constrained trees (reproduced from (Ritchie et al., 2015))

The constrained procedural graphics (Ritchie et al., 2015) is a visually compelling and elucidating application of probabilistic programming. Consider how one makes a computer graphics forest for a movie or computer game. One does not hire one thousand designers to make each create a tree. Instead one hires a procedural graphics programmer who writes what we call a generative model – a stochastic simulator that generates a synthetic tree each time it is run. A forest is then constructed by calling such a program many times and arranging the trees on a landscape. What if, however, a director enters the design process and stipulates, for whatever reason, that the tree cannot touch some other elements in the scene, i.e. in probabilistic programming lingo we "observe" that the tree cannot touch some elements? Figure 1.3 shows examples of such a situation where the tree on the left must miss the back wall and grey bars and the tree on the right must miss the blue and red logo. In these figures you can see, visually, what we will examine in a high level of detail throughout the tutorial. The random choices made by the generative procedural graphics model correspond to branch elongation lengths, how many branches diverge from the trunk and subsequent branch locations, the angles that the diverged

branches take, the termination condition for branching and elongation, and so forth. Each tree literally corresponds to one execution path or setting of the random variables of the generative program. Conditioning with hard constraints like these transforms the prior distribution on trees into a posterior distribution in which all posterior trees conform to the constraint. Valid program variable settings (those present in the posterior) have to make choices at all intermediate sampling points that allow all other sampling points to take at least one value that can result in a tree obeying the statistical regularities specified by the prior and the specified constraints as well.

Program Induction

How do you automatically write a program that performs an operation you would like it to? One approach is to use a probabilistic programming system and inference to invert a generative model that generates normal, regular, computer program code and conditions on its output, when run on examples, conforming to the observed specification. This is the central idea in the work of Perov and Wood (2016) whose use of probabilistic programming is what distinguishes their work from the related literature (Gulwani et al., 2017; Hwang et al., 2011; Liang et al., 2010). Examples such as this, even more than the preceding visually compelling examples, illustrate the denotational convenience of a rich and expressive programming language as the generative modeling language. A program that writes programs is most naturally expressed as a recursive program with random choices that generates abstract syntax trees according to some learned prior on the same space. While models from the natural language processing literature exist that allow specification and generation of computer source code (e.g. adaptor grammars (Johnson et al., 2007)), they are at best cumbersome to denote mathematically.

Recursive Multi-Agent Reasoning

Some of the most interesting uses for probabilistic programming systems derive from the rich body of work around the Church and WebPPL

systems. The latter, in particular, has been used to study the mutually-recurisive reasoning among multiple agents. A number of examples on this are detailed in an excellent online tutorial (Goodman and Stuhlmüller, 2014).

The list goes on and could occupy a substantial part of a book itself. The critical realization to make is that, of course, any traditional statistical model can be expressed in a probabilistic programming framework, but, more importantly, so too can many others and with significantly greater ease. Models that take advantage of existing source code packages to do sophisticated nonlinear deterministic computations are particularly of interest. One exciting example application under consideration at the time of writing is to instrument the stochastic simulators that simulate the standard model and the detectors employed by the large hadron collider (Baydin et al., 2018). By "observing" the detector outputs, inference in the generative model specified by the simulation pipeline may prove to be able to produce the highest fidelity event reconstruction and science discoveries.

This last example highlights one of the principle promises of probabilistic programming. There exist a large number of software simulation modeling efforts to simulate, stochastically and deterministically, engineering and science phenomena of interest. Unlike in machine learning where often the true generative model is not well understood, in engineering situations (like building, engine, or other system modeling) the forward model is sometimes in fact incredibly well understood and already written. Probabilistic programming techniques and evaluators that work within the framework of existing languages should prove to be very valuable in disciplines where significant effort has been put into modeling complex engineering or science phenomena of interest and the power of general purpose inverse reasoning has not yet been made available.

1.4 A First Probabilistic Program

Just before we dig in deeply, it is worth considering at least one simple probabilistic program to informally introduce a bit of syntax and relate

a model denotation in a probabilistic programming language to the underlying mathematical denotation and inference objective. There will be source code examples provided throughout, though not always with accompanying mathematical denotation.

Recall the simple beta-Bernoulli model from Section 1.1. This is one in which the probabilistic program denotation is actually longer than the mathematical denotation. But that is largely unique to such trivially simple models. Here is a probabilistic program that represents the beta-Bernoulli model.

```
(let [prior (beta a b)
      x (sample prior)
      likelihood (bernoulli x)
      y 1]
  (observe likelihood y)
  x))
```

Program 1.1: The beta-Bernoulli model as a probabilistic program

This program is written in the Lisp dialect we will use throughout, and which we will define in glorious detail in the next chapter. Evaluating this program performs the same inference as described mathematically before, specifically to characterize the distribution on the return value `x` that is conditioned on the observed value `y`. The details of what this program means and how this is done form the majority of the remainder of this tutorial.

2

A Probabilistic Programming Language Without Recursion

In this and the next two chapters of this tutorial we will present the key ideas of probabilistic programming using a carefully designed first-order probabilistic programming language (FOPPL). The FOPPL includes most common features of programming languages, such as conditional statements (e.g. `if`) and primitive operations (e.g. `+`,`-`, etc.), and user-defined functions. The restrictions that we impose are that functions must be first order, which is to say that functions cannot accept other functions as arguments, and that they cannot be recursive.

These two restrictions result in a language where models describe distributions over a finite number of random variables. In terms of expressivity, this places the FOPPL on even footing with many existing languages and libraries for automating inference in graphical models with finite graphs. As we will see in Chapter 3, we can compile any program in the FOPPL to a data structure that represents the corresponding graphical model. This turns out to be a very useful property when reasoning about inference, since it allows us to make use of existing theories and algorithms for inference in graphical models.

A corollary to this characteristic is that the computation graph of any FOPPL program can be completely determined in advance. This

v ::= variable
c ::= constant value or primitive operation
f ::= procedure
e ::= c | v | (let [v e_1] e_2) | (if e_1 e_2 e_3)
| (f e_1 ... e_n) | (c e_1 ... e_n)
| (sample e) | (observe e_1 e_2)
q ::= e | (defn f [v_1 ... v_n] e) q

Language 2.1: First-order probabilistic programming language (FOPPL)

suggests a place for FOPPL programs in the spectrum between static and dynamic computation graph programs. While in a FOPPL program conditional branching might dictate that not all of the nodes of its computation graph are active in the sense of being on the control-flow path, it is the case that all FOPPL programs can be unrolled to computation graphs where all possible control-flow paths are explicitly and completely enumerated at compile time. FOPPL programs have static computation graphs.

Although we have endeavored to make this tutorial as self-contained as possible, readers unfamiliar with graphical models or wishing to brush up on them are encouraged to refer to the textbooks by Bishop (2006), Murphy (2012), or Koller and Friedman (2009), all of which contain a great deal of material on graphical models and associated inference algorithms.

2.1 Syntax

The FOPPL is a Lisp variant that is based on Clojure (Hickey, 2008). Lisp variants are all substantially similar and are often referred to as dialects. The syntax of the FOPPL is specified by the grammar in Language 2.1. A grammar like this formulates a set of production rules, which are recursive, from which all valid programs must be constructed.

We define the FOPPL in terms of two sets of production rules: one for expressions e and another for programs q. Each set of rules is shown on the right hand side of ::= separated by a |. We will here provide a very brief self-contained explanation of each of the production rules.

For those who wish to read about programming languages essentials in further detail, we recommend the books by Abelson et al. (1996) and Friedman and Wand (2008).

The rules for q state that a program can either be a single expression e, or a function declaration `(defn` f `...)` followed by any valid program q. Because the second rule is recursive, these two rules together state that a program is a single expression e that can optionally be preceded by one or more function declarations.

The rules for expressions e are similarly defined recursively. For example, in the production rule `(if` e_1 e_2 e_3`)`, each of the sub-expressions e_1, e_2, and e_3 can be expanded by choosing again from the matching rules on the left hand side. The FOPPL defines eight expression types. The first six are "standard" in the sense that they are commonly found in non-probabilistic Lisp dialects:

1. A constant c can be a value of a primitive data type such as a number, a string, or a boolean, a built-in primitive function such as `+`, or a value of any other data type that can be constructed using primitive procedures, such as lists, vectors, maps, and distributions, which we will briefly discuss below.

2. A variable v is a symbol that references the value of another expression in the program.

3. A let form `(let [`v e_1`]` e_2`)` binds the value of the expression e_1 to the variable v, which can then be referenced in the expression e_2, which is often referred to as the body of the let expression.

4. An if form `(if` e_1 e_2 e_3`)` takes the value of e_2 when the value of e_1 is logically true and the value of e_3 when e_1 is logically false.

5. A function application `(`f e_1 ... e_n`)` calls the user-defined function f, which we also refer to as a procedure, with arguments e_1 through e_n. Here the notation $e_1 \dots e_n$ refers to a variable-length sequence of arguments, which includes the case `(`f`)` for a procedure call with no arguments.

6. A primitive procedure applications `(`c e_1 ... e_n`)` calls a built-in function c, such as `+`.

The remaining two forms are what makes the FOPPL a probabilistic programming language:

7. A sample form (`sample` e) represents an unobserved random variable. It accepts a single expression e, which must evaluate to a distribution object, and returns a value that is a sample from this distribution. Distributions are constructed using primitives provided by the FOPPL. For example, (`normal 0.0 1.0`) evaluates to a standard normal distribution.

8. An observe form (`observe` e_1 e_2) represents an observed random variable. It accepts an argument e_1, which must evaluate to a distribution, and conditions on the next argument e_2, which is the value of the random variable.

Some things to note about this language are that it is simple, i.e. the grammar only has a small number of special forms. It also has no input/output functionality which means that all data must be inlined in the form of an expression. However, despite this relative simplicity, we will see that we can express any graphical model as a FOPPL program. At the same time, the relatively small number of expression forms makes it much easier to reason about implementations of compilation and evaluation strategies.

Relative to other Lisp dialects, the arguably most critical characteristic of the FOPPL is that, provided that all primitives halt on all possible inputs, potentially non-halting computations are disallowed; in fact, there is a finite upper bound on the number of computation steps and this upper bound can be determined in compilation time. This design choice has several consequences. The first is that all data needs to be inlined so that the number of data points is known at compile time. A second consequence is that FOPPL grammar precludes higher-order functions, which is to say that user-defined functions cannot accept other functions as arguments. The reason for this is that a reference to user-defined function f is in itself not a valid expression type. Since arguments to a function call must be expressions, this means that we cannot pass a function f' as an argument to another function f.

Finally, the FOPPL does not allow recursive function calls, although

the syntax does not forbid them. This restriction can be enforced via the scoping rules in the language. In a program q of the form

`(defn` f_1 `...)` `(defn` f_2 `...)` e

we can call f_1 inside of f_2, but not vice versa, since f_2 is defined after f_1. Similarly, we impose the restriction that we cannot call f_1 inside f_1, which we can intuitively think of as f_1 not having been defined yet. Enforcing this restriction can be done using a pre-processing step.

A second distinction between the FOPPL relative to other Lisp is that we will make use of vector and map data structures, analogous to the ones provided by Clojure:

- Vectors `(vector` e_1 ... e_n`)` are similar to lists. A vector can be represented with the literal `[`e_1 ... e_n`]`. This is often useful when representing data. For example, we can use `[1 2]` to represent a pair, whereas the expression `(1 2)` would throw an error, since the constant 1 is not a primitive function.

- Hash maps `(hash-map` e_1 e_1' ... e_n e_n'`)` are constructed from a sequence of key-value pairs e_i e_i'. A hash-map can be represented with the literal `{`e_1 e_1' ... e_n e_n'`}`.

Note that we have not explicitly enumerated primitive functions in the FOPPL. We will implicitly assume existence of arithmetic primitives like `+`, `-`, `*`, and `/`, as well as distribution primitives like `normal` and `discrete`. In addition we will assume the following functions for interacting with data structures

- `(first` e`)` retrieves the first element of a list or vector e.
- `(last` e`)` retrieves the last element of a list or vector e.
- `(append` e_1 e_2`)` appends e_2 to the end of a list or vector e_1.[1]
- `(get` e_1 e_2`)` retrieves an element at index e_2 from a list or vector e_1, or the element at key e_2 from a hash map e_1.

[1] Readers familiar with Lisp dialects will notice that `append` differs somewhat from the semantics of primitives like `cons`, which prepends to a list, or the Clojure primitive `conj` which prepends to a list and appends to a vector.

```
(defn observe-data [slope intercept x y]
  (let [fx (+ (* slope x) intercept)]
    (observe (normal fx 1.0) y)))

(let [slope (sample (normal 0.0 10.0))]
  (let [intercept  (sample (normal 0.0 10.0))]
    (let [y1 (observe-data slope intercept 1.0 2.1)]
    (let [y2 (observe-data slope intercept 2.0 3.9)]
    (let [y3 (observe-data slope intercept 3.0 5.3)]
    (let [y4 (observe-data slope intercept 4.0 7.7)]
    (let [y5 (observe-data slope intercept 5.0 10.2)]
      [slope intercept])))))))).
```

Program 2.2: Bayesian linear regression in the FOPPL.

- `(put` e_1 e_2 e_3`)` replaces the element at index/key e_2 with the value e_3 in a vector or hash-map e_1.
- `(remove` e_1 e_2`)` removes the element at index/key e_2 with the value e_2 in a vector or hash-map e_1.

Note that FOPPL primitives are pure functions. In other words, the `append`, `put`, and `remove` primitives do not modify e_1 in place, but instead return a modified copy of e_1. Efficient implementations of such functionality may be advantageously achieved via pure functional data structures (Okasaki, 1999).

Finally we note that we have not specified any type system or specified exactly what values are allowable in the language. For example, `(sample e)` will fail if at runtime `e` does not evaluate to a distribution-typed value.

Now that we have defined our syntax, let us illustrate what a program in the FOPPL looks like. Program 2.2 shows a simple univariate linear regression model. The program defines a distribution on lines expressed in terms of their slopes and intercepts by first defining a prior distribution on slope and intercept and then conditioning it using five observed data pairs. The procedure `observe-data` conditions the generative model given a pair `(x,y)`, by observing the value `y` from a normal centered around the value `(+ (* slope x) intercept)`. Using a

procedure lets us avoid rewriting observation code for each observation pair. The procedure returns the observed value, which is ignored in our case. The program defines a prior on `slope` and `intercept` using the primitive procedure `normal` for creating an object for normal distribution. After conditioning this prior with data points, the program return a pair `[slope intercept]`, which is a sample from the posterior distribution conditioned on the 5 observed values.

2.2 Syntactic Sugar

The fact that the FOPPL only provides a small number of expression types is a big advantage when building a probabilistic programming system. We will see this in Chapter 3, where we will define a translation from any FOPPL program to a Bayesian network using only 8 rules (one for each expression type). At the same time, for the purposes of writing probabilistic programs, having a small number of expression types is not always convenient. For this reason we will provide a number of alternate expression forms, which are referred to as syntactic sugar, to aid readability and ease of use.

We have already seen two very simple forms of syntactic sugar: `[...]` is a sugared form of `(vector ...)` and `{...}` is a sugared form for `(hash-map ...)`. In general, each sugared expression form can be desugared, which is to say that it can be reduced to an expression in the grammar in Language 2.1. This desugaring is done as a preprocessing step, often implemented as a macro rewrite rule that expands each sugared expression into the equivalent desugared form.

2.2.1 Let forms

The base let form `(let [`v e_1`]` e_2`)` binds a single variable v in the expression e_2. Very often, we will want to define multiple variables, which leads to nested let expressions like the ones in Program 2.2. Another distracting piece of syntax in this program is that we define dummy variables `y1` to `y5` which are never used. The reason for this is that we are not interested in the values returned by calls to `observe-data`; we are using this function in order to observe values, which is a side-effect

of the procedure call.

To accommodate both these use cases in let forms, we will make use of the following generalized let form

```
(let [v1 e1
      ⋮
      vn en]
  en+1 ... em−1 em).
```

This allows us to simplify the nested let forms in Program 2.2 to

```
(let [slope (sample (normal 0.0 10.0))
      intercept (sample (normal 0.0 10.0))]
  (observe-data slope intercept 1.0 2.1)
  (observe-data slope intercept 2.0 3.9)
  (observe-data slope intercept 3.0 5.3)
  (observe-data slope intercept 4.0 7.7)
  (observe-data slope intercept 5.0 10.2)
  [slope intercept])
```

This form of `let` is desugared to the following expression in the FOPPL

```
(let [v1 e1]
  ⋮
  (let [vn en]
    (let [_ en+1]
      ⋮
      (let [_ em−1]
        em)···))).
```

Here the underscore `_` is a second form of syntactic sugar, that will be expanded to a fresh (i.e. previously unused) variable. For instance

```
(let [_ (observe (normal 0 1) 2.0)] ...)
```

will be expanded by generating some *fresh variable* symbol, say `x284xu`,

```
(let [x284xu (observe (normal 0 1) 2.0)] ...)
```

We will assume each instance of `_` is a guaranteed-to-be-unique or fresh symbol that is generated by some `gensym` primitive in the implementing language of the evaluator. We will use the concept of a fresh variable extensively throughout this tutorial, with the understanding that fresh variables are unique symbols in all cases.

2.2.2 For loops

A second syntactic inconvenience in Program 2.2 is that we have to repeat the expression `(observe-data ...)` once for each data point. Just about any language provides looping constructs for this purpose. In the FOPPL we will make use of two such constructs. The first is the `foreach` form, which has the following syntax

```
(foreach c
  [v1 e1 ... vn en]
  e'1 ... e'k)
```

This form desugars into a vector containing c let forms

```
(vector
  (let [v1 (get e1 0)
        ⋮
        vn (get en 0)]
    e'1 ... e'k)
  ⋮
  (let [v1 (get e1 (- c 1))
        ⋮
        vn (get en (- c 1))]
    e'1 ... e'k))
```

Note that this syntax looks very similar to that of the let form. However, whereas let binds each variable to a single value, the foreach form associates each variable v_i with a sequence e_i and then maps over the values in this sequence for a total of c steps, returning a vector of results. If the length of any of the bound sequences is less than c, then let form will result in a runtime error.

With the foreach form, we can rewrite Program 2.2 without having to make use of the helper function `observe-data`

```
(let [y-values [2.1 3.9 5.3 7.7 10.2]
      slope (sample (normal 0.0 10.0))
      intercept (sample (normal 0.0 10.0))]
  (foreach 5
    [x (range 1 6)
     y y-values]
    (let [fx (+ (* slope x) intercept)]
```

```
      (observe (normal fx 1.0) y)))
  [slope intercept])
```

There is a very specific reason why we defined the foreach syntax using a constant for the number of loop iterations (`foreach` c `[...] ...`). Suppose we were to define the syntax using an arbitrary expression (`foreach` e `[...] ...`). Then we could write programs such as

```
(let [m (sample (poisson 10.0))]
  (foreach m []
    (sample (normal 0 1))))
```

This defines a program in which there is no upper bound on the number of times that the expression (`sample (normal 0 1)`) will be evaluated. By requiring c to be a constant, we can guarantee that the number of iterations is known at compile time.

Note that there are less obtrusive mechanisms for achieving the functionality of `foreach`, which is fundamentally a language feature that maps a function, here the body, over a sequence of arguments, here the `let`-like bindings. Such functionality is much easier to express and implement using higher-order language features like those discussed in Chapter 5.

2.2.3 Loop forms

The second looping construct that we will use is the loop form, which has the following syntax.

`(loop` c e f e_1 ... e_n`)`

Once again, c must be a non-negative integer *constant* and f a procedure, primitive or user-defined. This notation can be used to write most kinds of for loops. Desugaring this syntax rolls out a nested set of lets and function calls in the following precise way

```
(let [a1 e1
      a2 e2
         ⋮
      an en]
  (let [v0 (f 0 e a1 ... an)]
    (let [v1 (f 1 v0 a1 ... an)]
```

```
(defn regr-step [n r2 xs ys slope intercept]
  (let [x (get xn n)
        y (get ys n)
        fx (+ (* slope x) intercept)
        r (- y fx)]
    (observe (normal fx 1.0) y)
    (+ r2 (* r r))))

(let [xs [1.0 2.0 3.0 4.0  5.0]
      ys [2.1 3.9 5.3 7.7 10.2]
      slope (sample (normal 0.0 10.0))
      bias  (sample (normal 0.0 10.0))
      r2 (loop 5 0.0 regr-step xs ys slope bias)]
  [slope bias r2])
```

Program 2.3: The Bayesian linear regression model, written using the loop form.

```
(let [v2 (f 2 v1 a1 ... an)]
  ⋮
    (let [vc−1 (f (- c 1) vc−2 a1 ... an)]
      vc−1) ··· )))
```

where $v_0, \ldots, v_{c-1}$ and $a_0, \ldots, a_n$ are fresh variables. Note that the `loop` sugar computes an iteration over a fixed set of indices.

To illustrate how the `loop` form differs from the `foreach` form, we show a new variant of the linear regression example in Program 2.3. In this version of the program, we not only observe a sequence of values y_n according to a normal centered at $f(x_n)$, but we also compute the sum of the squared residuals $r^2 = \sum_{n=1}^{5}(y_n - f(x_n))^2$. To do this, we define a function `regr-step`, which accepts an argument `n`, the index of the loop iteration. It also accepts a second argument `r2`, which represents the sum of squares for the preceding datapoints. Finally it accepts the arguments `xs`, `ys`, `slope`, and `intercept`, which we have also used in previous versions of the program.

At each loop iteration, the function `regr-step` computes the residual $r = y_n - f(x_n)$ and returns the value `(+ r2 (* r r))`, which becomes the new value for `r2` at the next iteration. The value of the entire loop form is the value of the final call to `regr-step`, which is the sum of

squared residuals.

In summary, the difference between `loop` and `foreach` is that loop can be used to accumulate a result over the course of the iterations. This is useful when you want to compute some form of sufficient statistics, filter a list of values, or really perform any sort of computation that iteratively builds up a data structure. The `foreach` form provides a much more specific loop type that evaluates a single expression repeatedly with different values for its variables. From a statistical point of view, we can thing of `loop` as defining a sequence of dependent variables, whereas `foreach` creates conditionally independent variables.

2.3 Examples

Now that we have defined the fundamental expression forms in the FOPPL, along with syntactic sugar for variable bindings and loops, let us look at how we would use the FOPPL to define some models that are commonly used in statistics and machine learning.

2.3.1 Gaussian mixture model

We will begin with a three-component Gaussian mixture model (McLachlan and Peel, 2004). A Gaussian mixture model is a density estimation model often used for clustering, in which each data point y_n is assigned to a latent class z_n. We will here consider the following generative model

$$\sigma_k \sim \text{Gamma}(1.0, 1.0), \qquad \text{for } k = 1, 2, 3, \tag{2.1}$$

$$\mu_k \sim \text{Normal}(0.0, 10.0), \qquad \text{for } k = 1, 2, 3, \tag{2.2}$$

$$\pi \sim \text{Dirichlet}(1.0, 1.0, 1.0), \tag{2.3}$$

$$z_n \sim \text{Discrete}(\pi), \qquad \text{for } n = 1, \ldots, 7, \tag{2.4}$$

$$y_n | z_n = k \sim \text{Normal}(\mu_k, \sigma_k). \tag{2.5}$$

Program 2.4 shows a translation of this generative model to the FOPPL. In this model we first sample the mean `mu` and standard deviation `sigma` for 3 mixture components. For each observation `y` we then sample a class assignment `z`, after which we observe according to the likelihood of the sampled assignment. The return value from this

```
(let [data [1.1 2.1 2.0 1.9 0.0 -0.1 -0.05]
      likes (foreach 3 []
              (let [mu (sample (normal 0.0 10.0))
                    sigma (sample (gamma 1.0 1.0))]
                (normal mu sigma)))
      pi (sample (dirichlet [1.0 1.0 1.0]))
      z-prior (discrete pi)]
  (foreach 7 [y data]
    (let [z (sample z-prior)]
      (observe (get likes z) y)
      z)))
```

Program 2.4: FOPPL - Gaussian mixture model with three components

program is the sequence of latent class assignments, which can be used to ask questions like, "Are these two datapoints similar?", etc.

2.3.2 Hidden Markov model

As a second example, let us consider Program 2.5 which denotes a hidden Markov model (HMM) (Rabiner, 1989) with known initial state, transition, and observation distributions governing 16 sequential observations.

In this program we begin by defining a vector of data points `data`, a vector of transition distributions `trans-dists` and a vector of state likelihoods `likes`. We then loop over the data using a function `hmm-step`, which returns a sequence of states.

At each loop iteration, the function `hmm-step` does three things. It first samples a new state `z` from the transition distribution associated with the preceding state. It then observes data point at time `t` according to the likelihood component of the current state. Finally, it appends the state `z` to the sequence `states`. The vector of accumulated latent states is the return value of the program and thus the object whose joint posterior distribution is of interest.

```
(defn hmm-step [t states data trans-dists likes]
  (let [z (sample (get trans-dists
                       (last states)))]
    (observe (get likes z)
             (get data t))
    (append states z)))

(let [data [0.9 0.8 0.7 0.0 -0.025 -5.0 -2.0 -0.1
            0.0 0.13 0.45 6 0.2 0.3 -1 -1]
      trans-dists [(discrete [0.10 0.50 0.40])
                   (discrete [0.20 0.20 0.60])
                   (discrete [0.15 0.15 0.70])]
      likes [(normal -1.0 1.0)
             (normal 1.0 1.0)
             (normal 0.0 1.0)]
      states [(sample (discrete [0.33 0.33 0.34]))]]
  (loop 16 states hmm-step
    data trans-dists likes))
```

Program 2.5: FOPPL - Hidden Markov model

2.3.3 A Bayesian Neural Network

Traditional neural networks are fixed-dimension computation graphs which means that they too can be expressed in the FOPPL. In the following we demonstrate this with an example taken from the documentation for Edward (Tran et al., 2016), a probabilistic programming library based on fixed computation graph. The example shows a Bayesian approach to learning the parameters of a three-layer neural network with input of dimension one, two hidden layer of dimension ten, an independent and identically Gaussian distributed output of dimension one, and `tanh` activations at each layer. The program inlines five data points and represents the posterior distribution over the parameters of the neural network. We have assumed, in this code, the existence of matrix primitive functions, e.g. `mat-mul`, whose meaning is clear from context (matrix multiplication), sensible matrix-dimension-sensitive pointwise `mat-add` and `mat-tanh` functionality, vector of vectors matrix storage, etc.

```
(let [weight-prior (normal 0 1)
      W_0 (foreach 10 []
            (foreach 1 [] (sample weight-prior)))
      W_1 (foreach 10 []
            (foreach 10 [] (sample weight-prior)))
      W_2 (foreach 1 []
            (foreach 10 [] (sample weight-prior)))

      b_0 (foreach 10 []
            (foreach 1 [] (sample weight-prior)))
      b_1 (foreach 10 []
            (foreach 1 [] (sample weight-prior)))
      b_2 (foreach 1 []
            (foreach 1 [] (sample weight-prior)))

      x   (mat-transpose [[1] [2] [3] [4] [5]])
      y   [[1] [4] [9] [16] [25]]
      h_0 (mat-tanh (mat-add (mat-mul W_0 x)
                             (mat-repmat b_0 1 5)))
      h_1 (mat-tanh (mat-add (mat-mul W_1 h_0)
                             (mat-repmat b_1 1 5)))
      mu  (mat-transpose
             (mat-tanh
               (mat-add (mat-mul W_2 h_1)
                        (mat-repmat b_2 1 5))))]
  (foreach 5 [y_r y
              mu_r mu]
    (foreach 1 [y_rc y_r
                mu_rc mu_r]
      (observe (normal mu_rc 1) y_rc)))
  [W_0 b_0 W_1 b_1])
```

Program 2.6: FOPPL - A Bayesian Neural Network

This example provides an opportunity to reinforce the close relationship between optimization and inference. The task of estimating neural-network parameters is typically framed as an optimization in which the free parameters of the network are adjusted, usually via gradient descent, so as to minimize a loss function. This neural-network example can be seen as doing parameter learning too, except using the tools of inference to discover the posterior distribution over model parameters. In general, all parameter estimation tasks can be framed as inference simply by placing a prior over the parameters of interest as we do here.

It can also be noted that, in this setting, any of the activations of the neural network trivially could be made stochastic, yielding a stochastic computation graph (Schulman et al., 2015), rather than a purely deterministic neural network.

Finally, the point of this example is not to suggest that the FOPPL is *the* language that should be used for denoting neural network learning and inference problems, it is instead to show that the FOPPL is sufficiently expressive to neural networks based on fixed computation graphs. Even though we have shown only one example of a multilayer perceptron, it is clear that convolutional neural networks, recurrent neural networks of fixed length, and the like, can all be denoted in the FOPPL.

2.3.4 Translating BUGS models

The FOPPL language as specified is sufficiently expressive to, for instance, compile BUGS programs to the FOPPL. Program 2.7 shows one of the examples included with the BUGS system (OpenBugs, 2009). This model is a conjugate gamma-Poisson hierarchical model, which is to say that it has the following generative model:

$$a \sim \text{Exponential}(1), \tag{2.6}$$

$$b \sim \text{Gamma}(0.1, 1), \tag{2.7}$$

$$\theta_i \sim \text{Gamma}(a, b), \qquad \text{for } i = 1, \ldots, 10, \tag{2.8}$$

$$y_i \sim \text{Poisson}(\theta_i t_i) \qquad \text{for } i = 1, \ldots, 10. \tag{2.9}$$

Program 2.7 shows this model in the BUGS language. Program 2.8

```
# data
list(t = c(94.3, 15.7, 62.9, 126, 5.24,
           31.4, 1.05, 1.05, 2.1, 10.5),
     y = c(5, 1, 5, 14, 3, 19, 1, 1, 4, 22),
     N = 10)
# inits
list(a = 1, b = 1)
# model
{
  for (i in 1 : N) {
     theta[i] ~ dgamma(a, b)
     l[i] <- theta[i] * t[i]
     y[i] ~ dpois(l[i])
  }
  a ~ dexp(1)
  b ~ dgamma(0.1, 1.0)
}
```

Program 2.7: The Pumps example model from BUGS (OpenBugs, 2009).

show a translation to the FOPPL that was returned by an automated BUGS-to-FOPPL compiler. Note the similarities between these languages despite the substantial syntactic differences. In particular, both require that the number of loop iterations $N = 10$ is fixed and finite. In BUGS the variables whose values are known appear in a separate data block. The symbol $\sim$ is used to define random variables, which can be either latent or observed, depending on whether a value for the random variable is present. In our FOPPL the distinction between observed and latent random variables is made explicit through the syntactic difference between sample and observe. A second difference is that a BUGS program can in principle be used to compute a marginal on any variable in the program, whereas a FOPPL program specifies a marginal of the full posterior through its return value. As an example, in this particular translation, we treat θ_i as a nuisance variable, which is not returned by the program, although we could have used the loop construct to accumulate a sequence of θ_i values.

These minor differences aside, the BUGS language and the FOPPL essentially define equivalent families of probabilistic programs. An ad-

```
(defn data []
  [[94.3 15.7 62.9 126 5.24 31.4 1.05 1.05 2.1 10.5]
   [5 1 5 14 3 19 1 1 4 22]
   [10]])

(defn t [i] (get (get (data) 0) i))
(defn y [i] (get (get (data) 1) i))

(defn loop-iter [i _ alpha beta]
  (let [theta (sample (gamma a b))
        l     (* theta (t i))]
    (observe (poisson l) (y i))))

(let [a (sample (exponential 1))
      b (sample (gamma 0.1 1.0))]
  (loop 10 nil loop-iter a b)
  [a b])
```

Program 2.8: FOPPL - the Pumps example model from BUGS

vantage of writing this text using the FOPPL rather than an existing language like BUGS is that FOPPL program are comparatively easy to reason about and manipulate, since there are only 8 expression forms in the language. In the next chapter we will exploit this in order to mathematically define a translation from FOPPL programs to Bayesian networks and factor graphs, keeping in mind that all the basic concepts that we will employ also apply to other probabilistic programming systems, such as BUGS.

2.4 A Simple Purely Deterministic Language

There is no optimal place to put this section so it appears here, although it is very important for understanding what is written in the remainder of this tutorial.

In subsequent chapters it will become apparent that the FOPPL can be understood in two different ways – one way as being a language for specifying graphical-model data-structures on which traditional inference algorithms may be run, the other as a language that requires a

non-standard interpretation in some implementing language to characterize the denoted posterior distribution.

In the case of graphical-model construction it will be necessary to have a language for purely deterministic expressions. To foreshadow, this language will be used to express link functions in the graphical model. More precisely, and contrasting to the usual definition of link function from statistics, the pure deterministic language will encode functions that take values of parent random variables and produce distribution objects for children. These link functions cannot have random variables inside them; such a variable would be another node in the graphical model instead.

Moreover we can further simplify this link function language by removing user defined functions, effectively requiring their function bodies, if used, to be inlined. This yields a cumbersome language in which to manually program but an excellent language to target and evaluate because of its simplicity.

2.4.1 Deterministic Expressions

We will call expressions in the FOPPL that do not involve user-defined procedure calls and involve only deterministic computations, e.g. `(+ (/ 2.0 6.0) 17)` "0th-order expressions". Such expressions will play a prominent role when we consider the translation of our probabilistic programs to graphical models in the next chapter. In order to identify and work with these deterministic expressions we define a language with the following extremely simple grammar:

$c ::=$ constant value or primitive operation
$v ::=$ variable
$E ::= c \mid v \mid$ `(if` E_1 E_2 E_3`)` $\mid$ `(`c $E_1 \ldots E_n$`)`

Language 2.2: Sub-language for purely deterministic computations

Note that neither sample nor observe statements appear in the syntax, and that procedure calls are allowed only for primitive operations, not for defined procedures. Having these constraints ensures that expressions E cannot depend on any probabilistic choices or conditioning.

The examples provided in this chapter should convince you that many common models and inference problems from statistics and machine learning can be denoted as FOPPL programs. What remains is to translate FOPPL programs into other mathematical or programming language formalisms whose semantics are well established so that we can define, at least operationally, the semantics of FOPPL programs, and, in so doing, establish in your mind a clear idea about how probabilistic programming languages that are formally equivalent in expressivity to the FOPPL can be implemented.

3

Graph-Based Inference

3.1 Compilation to a Graphical Model

Programs written in the FOPPL specify probabilistic models over finitely many random variables. In this section, we will make this aspect clear by presenting the translation of these programs into finite graphical models. In the subsequent sections, we will show how this translation can be exploited to adapt inference algorithms for graphical models to probabilistic programs.

We specify translation using the following ternary relation $\Downarrow$, similar to the so called big-step evaluation relation from the programming language community.

$$\rho, \phi, e \Downarrow G, E \tag{3.1}$$

In this relation, ρ is a mapping from procedure names to their definitions, ϕ is a logical predicate for the flow control context, and e is an expression we intend to compile. This expression is translated to a graphical model G and an expression E in the deterministic sub-language described in Section 2.4.1. The expression E is deterministic in the sense that it does not involve sample nor observe. It describes the return value of the original expression e in terms of random variables in G. Vertices in

G represent random variables, and arcs dependencies among them. For each random variable in G, we will define a probability density or mass in the graph. For observed random variables, we additionally define the observed value, as well as a logical predicate that indicates whether the observe expression is on the control flow path, conditioned on the values of the latent variables.

Definition of a Graphical Model

We define a graphical model G as a tuple $(V, A, \mathcal{P}, \mathcal{Y})$ containing (i) a set of vertices V that represent random variables; (ii) a set of arcs $A \subseteq V \times V$ (i.e. directed edges) that represent conditional dependencies between random variables; (iii) a map $\mathcal{P}$ from vertices to deterministic expressions that specify the probability density or mass function for each random variable; (iv) a partial map $\mathcal{Y}$ that for each observed random variable contains a pair (E, Φ) consisting of a deterministic expression E for the observed value, and a predicate expression Φ that evaluates to `true` when this observation is on the control flow path.

Before presenting a set of translation rules that can be used to compile any FOPPL program to a graphical model, we will illustrate the intended translation using a simple example:

```
(let [z (sample (bernoulli 0.5))
      mu (if (= z 0) -1.0 1.0)
      d (normal mu 1.0)
      y 0.5]
  (observe d y)
  z)
```

Program 3.1: A simple example FOPPL program.

This program describes a two-component Gaussian mixture with a single observation. The program first samples z from a Bernoulli distribution, based on which it sets a likelihood parameter μ to -1.0 or 1.0, and observes a value $y = 0.5$ from a normal distribution with mean μ. This program defines a joint distribution $p(y = 0.5, z)$. The inference problem is then to characterize the posterior distribution $p(z \mid y)$. Figure 3.1 shows the graphical model and pure deterministic link functions that correspond to Program 3.1.

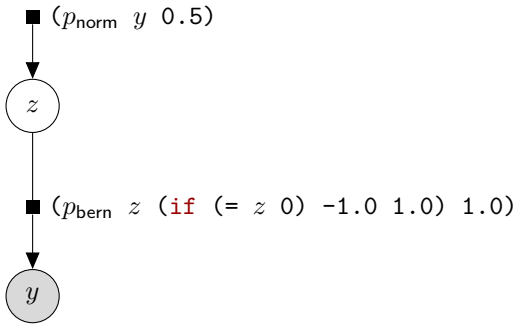

Figure 3.1: The graphical model corresponding to Program 3.1

In the evaluation relation $\rho, \phi, e \Downarrow G, E$, the source code of the program is represented as a single expression e. The variable ρ is an empty map, since there are no procedure definitions. At the top level, the flow control predicate ϕ is `true`. The graphical model $G = (V, A, \mathcal{P}, \mathcal{Y})$ and the result expression E that this program translates to are

$$
\begin{aligned}
V &= \{z, y\}, \\
A &= \{(z, y)\}, \\
\mathcal{P} &= [z \mapsto (p_{\mathsf{bern}}\ z\ 0.5), \\
&\qquad y \mapsto (p_{\mathsf{norm}}\ y\ (\texttt{if}\ (=\ z\ 0)\ -1.0\ 1.0)\ 1.0)], \\
\mathcal{Y} &= [y \mapsto 0.5] \\
E &= \ z
\end{aligned}
$$

The vertex set V of the net G contains two variables, whereas the arc set A contains a single pair (z, y) to mark the conditional dependence relationship between these two variables. In the map P, the probability mass for z is defined as the target language expression $(p_{\mathsf{bern}}\ z\ 0.5)$. Here p_{bern} refers to a function in the target languages that implements probability mass function for the Bernoulli distribution. Similarly, the density for y is defined using p_{norm}, which implements the probability density function for the normal distribution. Note that the expression for the program variable `mu` has been substituted into the density for y. Finally, the map $\mathcal{Y}$ contains a single entry that holds the observed value for y.

Assigning Symbols to Variable Nodes

In the above example we used the mathematical symbol z to refer to the random variable associated with the expression `(sample (bernoulli 0.5))` and the symbol y to refer to the observed variable with expression `(observe d y)`. In general there will be one node in the network for each sample and observe expression that is evaluated in a program. In the above example, there also happens to be a program variable `z` that holds the value of the sample expression for node z, and a program variable `y` that holds the observed value for node y, but this is of course not necessarily always the case. A particularly common example of this arises in programs that have procedures. Here the same sample and observe expressions in the procedure body can be evaluated multiple times. Suppose for example that we were to modify our program as follows:

```
(defn norm-gamma
  [m l a b]
  (let [tau (sample (gamma a b))
        sigma (/ 1.0 (sqrt tau))
        mu (sample (normal m (/ sigma (sqrt l))))]
    (normal mu sigma))))

(let [z (sample (bernoulli 0.5))
      d0 (norm-gamma -1.0 0.1 1.0 1.0)
      d1 (norm-gamma 1.0 0.1 1.0 1.0)]
  (observe (if (= z 0) d0 d1) 0.5)
  z)
```

In this version of our program we define two distributions `d0` and `d1` which are created by sampling a mean `mu` and a precision `tau` from a normal-gamma prior. We then observe either according to `d0` or `d1`. Clearly the mapping from random variables to program variables is less obvious here, since each sample expression in the body of `norm-gamma` is evaluated twice.

Below, we will define a general set of translation rules that compile a FOPPL program to a graphical model, in which we assign each vertex in the graphical model a newly generated unique symbol. However, when discussing programs in this tutorial, we will generally explicitly give

names to returns from sample and observe expressions that correspond to program variables to aid readability.

Recognize that assigning a label to each vertex is a way of assigning a unique "address" to each and every random variable in the program. Such unique addresses are important for the correctness and implementation of generic inference algorithms. In Chapter 6, Section 6.2 we develop a more explicit mechanism for addressing in the more difficult situation where not all control flow paths can be completely explored at compile time.

If Expressions in Graphical Models

When compiling a program to a graphical model, if expressions require special consideration. Before we set out to define translation rules that construct a graphical model for a program, we will first spend some time building intuition about how we would like these translation rules to treat if expressions. Let us start by considering a simple mixture model, in which only the mean is treated as an unknown variable:

```
(let [z (sample (bernoulli 0.5))
      mu (sample (normal (if (= z 0) -1.0 1.0) 1.0))
      d (normal mu 1.0)
      y 0.5]
  (observe d y)
  z)
```

Program 3.2: A one-point mixture with unknown mean

This is of course a really strange way of writing a mixture model. We define a single likelihood parameter μ, which is either distributed according to Normal$(-1, 1)$ when $z = 0$ and according to Normal$(1, 1)$ when $z = 1$. Typically, we would think of a mixture model as having two components with parameter μ_0 and μ_1 respectively, where z selects the component. A more natural way to write the model might be

```
(let [z (sample (bernoulli 0.5))
      mu0 (sample (normal -1.0 1.0))
      mu1 (sample (normal 1.0 1.0))
      d0 (normal mu0 1.0)
      d1 (normal mu1 1.0)
      y 0.5]
```

```
(observe (if (= z 0) d0 d1) y)
z)
```

Program 3.3: One-point mixture with explicit parameters

Here we sample parameters μ_0 and μ_1, which then define two component likelihoods `d0` and `d1`. The variable z then selects the component likelihood for an observation y.

Even though the second program defines a joint density on four variables $p(y, \mu_1, \mu_0, z)$, whereas the first program defines a density on three variables $p(y, \mu, z)$, it seems intuitive that these programs are equivalent in some sense. The equivalence that we would want to achieve here is that both programs define the same marginal posterior on z

$$p(z \mid y) = \int p(z, \mu \mid y) d\mu = \int \int p(z, \mu_0, \mu_1 \mid y) d\mu_0 d\mu_1.$$

So is there a difference between these two programs when both return `z`? The second program of course defines additional intermediate variables `d0` and `d1`, but these do not change the set of nodes in the corresponding graphical model. The essential difference is that in the first program, the if expression is placed *inside* the sample expression for `mu`, whereas in the second it sits *outside*. If we wanted to make the second program as similar as possible to the first, then we could write

```
(let [z (sample (bernoulli 0.5))
      mu0 (sample (normal -1.0 1.0))
      mu1 (sample (normal 1.0 1.0))
      mu (if (= z 0) mu0 mu1)
      d (normal mu 1.0)
      y 0.5]
  (observe d y)
  z)
```

Program 3.4: One-point mixture with explicit parameters simplified

In other words, because we have moved the if expression, we now need two sample expressions rather than one, resulting in a network with 4 nodes rather than 3. However, the distributions on return values of the programs should be equivalent.

This brings us to what turns out to be a fundamental design choice in probabilistic programming systems. Suppose we were to modify the above program to read

```
(let [z (sample (bernoulli 0.5))
      mu (if (= z 0)
           (sample (normal -1.0 1.0))
           (sample (normal 1.0 1.0)))
      d (normal mu 1.0)
      y 0.5]
  (observe d y)
  z)
```

Program 3.5: One-point mixture with samples inside if.

Is this program now equivalent to the first program, or to the second? The answer to this question depends on how we evaluate if expressions in our language.

In almost all mainstream programming languages, if expressions are evaluated in a lazy manner. In the example above, we would first evaluate the predicate `(= z 0)`, and then either evaluate the consequent branch, `(sample (normal -1.0 1.0))`, or the alternative branch, `(sample (normal 1.0 1.0))`, but never both. The opposite of a lazy evaluation strategy is an eager evaluation strategy. In eager evaluation, an if expression is evaluated like a normal function call. We first evaluate the predicate and both branches. We then return the value of one of the branches based on the predicate value.

If we evaluate if expressions lazily, then the program above is more similar to Program 3.2, in the sense that the program evaluates two sample expressions. If we use eager if, then the program evaluates three sample expressions and is therefore equivalent to Program 3.4. As it turns out, both strategies evaluation strategies offer certain advantages.

Suppose that we use μ_0 and μ_1 to refer to the sample expressions in both branches, then the joint $p(y, \mu_0, \mu_1, z)$ would have a conditional dependence structure[1]

$$p(y, \mu_0, \mu_1, z) = \; p(y \,|\, \mu_0, \mu_1, z) p(\mu_0|z) p(\mu_1|z) p(z).$$

[1]It might be tempting to instead define a distribution $p(y, \mu, z)$ as in the first program, by interpreting the entire if expression as a single random variable μ. For this particular example, this would work since both branches sample from a normal distribution. However, if we were for example to modify the $z = 1$ branch to sample from a Gamma distribution instead of a normal, then $\mu \in (-\infty, \infty)$ when $z = 0$ and $\mu \in (0, \infty)$ when $z = 1$, which means that the variable μ would no longer have a well-defined support.

Here the likelihood $p(y|\mu_0, \mu_1, z)$ is relatively easy to define,

$$p(y|\mu_0, \mu_1, z) = p_{\mathsf{norm}}(y; \mu_z, 1). \tag{3.2}$$

When translating our source code to a graphical model, the target language expression $\mathcal{P}(y)$ that evaluates this probability would read `(`p_{norm} y `(if (= z 0)` μ_0 μ_1`) 1)`.

The real question is how to define the probabilities for μ_0 and μ_1. One choice could be to simply set the probability of unevaluated branches to 1. One way to do this in this particular example is to write

$$p(\mu_0|z) = p_{\mathsf{norm}}(\mu_0; -1, 1)^z$$
$$p(\mu_1|z) = p_{\mathsf{norm}}(\mu_1; 1, 1)^{1-z}.$$

In the target language we could achieve the same effect by using if expressions defining $P(\mu_0)$ as `(if (= z 0) (`p_{norm} μ_0 `-1.0 1.0) 1.0)` and defining $\mathcal{P}(\mu_1)$ as `(if (not (= z 0)) (`p_{norm} μ_1 `1.0 1.0) 1.0)`.

On first inspection this design seems reasonable. Much in the way we would do in a mixture model, we either include $p(\mu_0|z = 0)$ or $p(\mu_1|z = 1)$ in the probability, and assume a probability 1 for unevaluated branches, i.e. $p(\mu_0|z = 1)$ and $p(\mu_1|z = 0)$.

On closer inspection, however, it is not obvious what the support of this distribution should have. We might naively suppose that $(y, \mu_0, \mu_1, z) \in \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \{0, 1\}$, but this definition is problematic. To see this, let us try to calculate the marginal likelihood $p(y)$,

$$\begin{aligned}
p(y) = \;& p(y, z = 0) + p(y, z = 1), \\
= \;& p(z = 0) \int d\mu_0 d\mu_1 \, p(y, \mu_0, \mu_1 | z = 0) \\
& + p(z = 1) \int d\mu_0 d\mu_1 \, p(y, \mu_0, \mu_1 | z = 1), \\
= \;& 0.5 \int d\mu_1 \left(\int d\mu_0 \, p_{\mathsf{norm}}(y; \mu_0, 1) p_{\mathsf{norm}}(\mu_0; -1, 1) \right) \\
& + 0.5 \int d\mu_0 \left(\int d\mu_1 \, p_{\mathsf{norm}}(y; \mu_1, 1) p_{\mathsf{norm}}(\mu_1; 1, 1) \right), \\
= \;& \infty.
\end{aligned}$$

So what is going on here? This integral does not converge because we have not assumed the correct support: We cannot marginalize

$\int_{\mathbb{R}} d\mu_0\, p(\mu_0|z=0)$ and $\int_{\mathbb{R}} d\mu_1\, p(\mu_1|z=1)$ if we assume $p(\mu_0|z=0)=1$ and $p(\mu_1|z=1)=1$. These uniform densities effectively specify improper priors on unevaluated branches.

In order to make lazy evaluation of if expressions more well-behaved, we could choose to define the support of the joint as a union over supports for individual branches

$$(y, \mu_0, \mu_1, z) \in (\mathbb{R} \times \mathbb{R} \times \{\texttt{nil}\} \times \{0\}) \cup (\mathbb{R} \times \{\texttt{nil}\} \times \mathbb{R} \times \{1\}). \quad (3.3)$$

In other words, we could restrict the support of variables in unevaluated branches to some special value `nil` to signify that the variable does not exist. Of course this can result in rather complicated definitions of the support in probabilistic programs with many levels of nested if expressions.

Could eager evaluation of branches yield a more straightforward definition of the probability distribution associated with a program? Let us look at Program 3.5 once more. If we use eager evaluation, then this program is equivalent to Program 3.3 which defines a distribution

$$p(y, \mu_0, \mu_1, z) = p(y|\mu_0, \mu_1, z)p(z)p(\mu_0)p(\mu_1).$$

We can now simply define $p(\mu_0) = p_{\mathsf{norm}}(\mu_0; -1, 1)$ and $p(\mu_1) = p_{\mathsf{norm}}(\mu_1; 1, 1)$ and assume the same likelihood as in the equation in (3.2). This defines a joint density that corresponds to what we would normally assume for a mixture model. In this evaluation model, sample expressions in both branches are always incorporated into the joint.

Unfortunately, eager evaluation would lead to counter-intuitive results when observe expressions occur in branches. To see this, Let us consider the following form for our program

```
(let [z (sample (bernoulli 0.5))
      mu0 (sample (normal -1.0 1.0))
      mu1 (sample (normal 1.0 1.0))
      y 0.5]
  (if (= z 0)
    (observe (normal mu0 1) y)
    (observe (normal mu1 1) y))
  z)
```

Program 3.6: One-point mixture with observes inside if.

Clearly it is not the case that eager evaluation of both branches is equivalent to lazy evaluation of one of the branches. When performing eager evaluation, we would be observing two variables y_0 and y_1, both with value 0.5. When performing lazy evaluation, only one of the two branches would be included in the probability density. The lazy interpretation is a lot more natural here. In fact, it seems difficult to imagine a use case where you would want to interpret observe expressions in branches in a eager manner.

So where does all this thinking about evaluation strategies for if expressions leave us? Lazy evaluation of if expressions makes it difficult to characterize the support of the probability distribution defined by a program when branches contain sample expressions. However, at the same time, lazy evaluation is essential in order for branches containing observe expressions to make sense. So have we perhaps made a fundamentally flawed design choice by allowing sample and observe to be used inside if branches?

It turns out that this is not necessarily the case. We just need to understand that observe and sample expressions affect the marginal posterior on a program output in very different ways. Sample expressions that are not on the flow control path cannot affect the values of any expressions outside their branch. This means they can be safely incorporated into the model as auxiliary variables, since they do not affect the marginal posterior on the return value. This guarantee does not hold for observed variables, which as a rule change the posterior on the return value when incorporated into a graphical model.[2]

Based on this intuition, the solution to our problem is straightforward: We can assign probability 1 to observed variables that are not on the same flow control path. Since observed variables have constant values, the interpretability of their support is not an issue in the way it is with sampled variables. Conversely we assign the same probability to sampled variables, regardless of the branch they occur in. We will describe how to accomplish this in the following sections.

[2]The only exception to this rule is observe expressions that are conditionally independent of the program output, which implies that the graphical model associated with the program could be split into two independent networks out of which one could be eliminated without affecting the distribution on return values.

Support-Related Subtleties

As a last but important bit of understanding to convey before proceeding to the translation rules in the next section it should be noted that the following two programs are allowed by the FOPPL and are not problematic despite potentially appearing to be.

```
(let [z (sample (poisson 10))
      d (discrete (range 1 z))]
  (sample d))
```

Program 3.7: Stochastic and potentially infinite discrete support

```
(let [z (sample (flip 0.5))
      d (if z (normal 1 1) (gamma 1 1)]
  (sample d)).
```

Program 3.8: Stochastic support and type

Program 3.7 highlights a subtlety of FOPPL language design and interpretation, that being that the distribution `d` has support that has potentially infinite cardinality. This is not problematic for the simple reason that samples from `d` cannot be used as a loop bound and therefore cannot possibly induce an unbounded number of random variables. It does serve as an indication that some care should be taken when reasoning about such programs and writing inference algorithms for the same. As is further highlighted in Program 3.8, which adds a seemingly innocuous bit of complexity to the control-flow examples from earlier in this chapter, neither the support nor the distribution type of a random variable need be the same between two different control flow paths. The fact that the support might be quite large can yield substantial value-dependent variation in inference algorithm runtimes. Moreover, inference algorithm implementations must have distribution library support that is robust to the possibility of needing to score values outside of their support.

Translation rules

Now that we have developed some intuition for how one might translate a program to a data structure that represents a graphical model and

have been introduced to several subtleties that arise in designing ways to do this, we are in a position to formally define a set of translation rules. We define the $\Downarrow$ relation for translation using the so called inference-rules notation from the programming language community. This notation specifies a recursive algorithm for performing the translation succinctly and declaratively. The inference-rules notation is

$$\frac{top}{bottom} \tag{3.4}$$

It states that if the statement *top* holds, so does the statement *bottom*. For instance, the rule

$$\frac{\rho, \phi, e \Downarrow G, E}{\rho, \phi, (- e) \Downarrow G, (- E)} \tag{3.5}$$

says that if e gets translated to G, E under ρ and ϕ, then its negation is translated to $G, (- E)$ under the same ρ and ϕ.

The grammar for the FOPPL in Language 2.1 describes 8 distinct expression types: (i) constants, (ii) variable references, (iii) let expressions, (iv) if expressions, (v) user-defined procedure applications, (vi) primitive procedure applications, (vi) sample expressions, and finally (viii) observe expressions. Aside from constants and variable references, each expression type can have sub-expressions. In the remainder of this section, we will define a translation rule for f type, under the assumption that we are already able to translate its sub-expressions, resulting in a set of rules that can be used to define the translation of every possible expression in the FOPPL language in a recursive manner.

Constants and Variables We translate constants c and variables z in FOPPL to themselves and the empty graphical model:

$$\frac{}{\rho,\ \phi,\ c \Downarrow G_{\mathsf{emp}},\ c} \qquad \frac{}{\rho,\ \phi,\ z \Downarrow G_{\mathsf{emp}},\ z}$$

where G_{emp} is the tuple $(\emptyset, \emptyset, [], [])$ and represents the empty graphical model.

Let We translate `(let [`v e_1`]` e_2`)` by first translating e_1, then substituting the outcome of this translation for v in e_2, and finally translating

the result of this substitution:

$$\frac{\rho,\, \phi,\, e_1 \Downarrow G_1,\, E_1 \qquad \rho,\, \phi,\, e_2[v := E_1] \Downarrow G_2,\, E_2}{\rho,\, \phi,\, (\texttt{let}\ [v\ e_1]\ e_2) \Downarrow (G_1 \oplus G_2),\, E_2}$$

Here $e_2[v := E_1]$ is a result of substituting E_1 for v in the expression e_1 (while renaming bound variables of e_2 if needed). $G_1 \oplus G_2$ is the combination of two disjoint graphical models: when $G_1 = (V_1,\, A_1,\, \mathcal{P}_1,\, \mathcal{Y}_1)$ and $G_2 = (V_2,\, A_2,\, \mathcal{P}_2,\, \mathcal{Y}_2)$,

$$(G_1 \oplus G_2) = (V_1 \cup V_2,\, A_1 \cup A_2,\, \mathcal{P}_1 \oplus \mathcal{P}_2,\, \mathcal{Y}_1 \oplus \mathcal{Y}_2)$$

where $\mathcal{P}_1 \oplus \mathcal{P}_2$ and $\mathcal{Y}_1 \oplus \mathcal{Y}_2$ are the concatenation of two finite maps with disjoint domains. This combination operator assumes that the input graphical models G_1 and G_2 use disjoint sets of vertices. This assumption always holds because every graphical model created by our translation uses fresh vertices, which do not appear in other networks previously generated.

We would like to note that this translation rule has not been optimized for computational efficiency. Because E_2 is replace by v in E_2, we will evaluate E_1 once for each occurrence of v. We could avoid these duplicate computations by incorporating deterministic nodes into our graph, but we omit this optimization in favor of readability.

If Our translation of the if expression is straightforward. It translates all the three sub-expressions, and puts the results from these translations together:

$$\frac{\begin{array}{c}\rho,\, \phi,\, e_1 \Downarrow G_1,\, E_1 \qquad \rho,\, (\texttt{and}\ \phi\ E_1),\, e_2 \Downarrow G_2,\, E_2 \\ \rho,\, (\texttt{and}\ \phi\ (\texttt{not}\ E_1)),\, e_3 \Downarrow G_3,\, E_3\end{array}}{\rho,\, \phi,\, (\texttt{if}\ e_1\ e_2\ e_3) \Downarrow (G_1 \oplus G_2 \oplus G_3),\, (\texttt{if}\ E_1\ E_2\ E_3)}$$

As we have explained already, the graphical models G_1, G_2 and G_3 use disjoint vertices, and so their combination $G_1 \oplus G_2 \oplus G_3$ is always defined. When we translate the sub-expressions for the consequent and alternative branches, we conjoin the logical predicate ϕ with the expression E_1 or its negation. The role of this logical predicate was established before; it being useful for including or excluding observe

statements that are on or off the current-sample control-flow path. It will be used in the upcoming translation of observe statements.

None of the rules for an expression e so far extends graphical models from e's sub-exressions with any new vertices. This uninteresting treatment comes from the fact that the programming constructs involved in these rules perform deterministic, not probabilistic, computations, and the translation uses graphical models to express random variables. The next two rules about `sample` and `observe` show this usage.

Sample We translate sample expressions using the following rule:

$$\frac{\begin{array}{ll}\rho,\, \phi,\, e \Downarrow (V, A, \mathcal{P}, \mathcal{Y}),\, E & \text{Choose a fresh variable } v \\ Z = \textsc{free-vars}(E) & F = \textsc{score}(E, v) \neq \bot\end{array}}{\rho,\, \phi,\, (\texttt{sample}\ e) \Downarrow (V \cup \{v\},\, A \cup \{(z, v) \mid z \in Z\},\, \mathcal{P} \oplus [v \mapsto F],\, \mathcal{Y}),\, v}$$

This rule states that we translate `(sample` e`)` in three steps. First, we translate the argument e to a graphical model $(V, A, \mathcal{P}, \mathcal{Y})$ and a deterministic expression E. Both the argument e and its translation E represent the same distribution, from which `(sample e)` samples. Second, we choose a fresh variable v, collect all free variables in E that are used as random variables of the network, and set Z to the set of these variables. Finally, we convert the expression E that denotes a distribution, to the probability density or mass function F of the distribution. This conversion is done by calling SCORE, which is defined as follows:

$\textsc{score}(\texttt{(if}\ E_1\ E_2\ E_3\texttt{)},\, v) = \texttt{(if}\ E_1\ F_2\ F_3\texttt{)}$
(when $F_i = \textsc{score}(E_i, v)$ for $i \in \{2, 3\}$ and it is not $\bot$)

$\textsc{score}(\texttt{(}c\ E_1 \ldots E_n\texttt{)},\, v) = \texttt{(}p_c\ v\ \ E_1 \ldots E_n\texttt{)}$
(when c is a constructor for distribution and p_c its pdf or pmf)

$\textsc{score}(E,\, v) = \bot$
(when E is not one of the above cases)

The $\bot$ (called "bottom", indicating terminating failure) case happens when the argument e in `(sample` e`)` does not denote a distribution. Our translation fails in that case.

Observe Our translation for observe expressions (`observe` e_1 e_2) is analogous to that of sample expressions, but we additionally need to account for the observed value e_2, and the predicate ϕ:

$$\frac{\begin{array}{ll}\rho,\, \phi,\, e_1 \Downarrow G_1,\, E_1 & \rho,\, \phi,\, e_2 \Downarrow G_2,\, E_2 \\ (V, A, \mathcal{P}, \mathcal{Y}) = G_1 \oplus G_2 & \text{Choose a fresh variable } v \\ F_1 = \textsc{score}(E_1, v) \neq \bot & F = (\texttt{if}\ \phi\ F_1\ 1) \\ Z = (\textsc{free-vars}(F_1) \setminus \{v\}) & \textsc{free-vars}(E_2) \cap V = \emptyset \\ B = \{(z, v) : z \in Z\} & \end{array}}{\rho,\, \phi,\, (\texttt{observe}\ e_1\ e_2) \Downarrow (V \cup \{v\},\, A \cup B,\, P \oplus [v \mapsto F],\, \mathcal{Y} \oplus [v \mapsto E_2]),\, E_2}$$

This translation rule first translates the sub-expressions e_1 and e_2. We then construct a network $(V, A, \mathcal{P}, \mathcal{Y})$ by merging the networks of the sub-expressions and pick a new variable v that will represent the observed random variable. As in the case of sample statements, the deterministic expression E_1 that is obtained by translating e_1 must evaluate to a distribution. We use the SCORE function to construct an expression F_1 that represents the probability mass or density of v under this distribution. We then construct a new expression $F = (\texttt{if}\ \phi\ F_1\ 1)$ to ensure that the probability of the observed variable evaluates to 1 if the observe expression occurs in a branch that was not followed. The free variables in this expression are the union of the free variables in E_1, the free variables in ϕ and the newly chosen variable v. We add a set of arcs B to the network, consisting of edges from all free variables in F to v, excluding v itself. Finally we add the expression F to $\mathcal{P}$ and store the observed value E_2 in $\mathcal{Y}$.

In order for this notion of an observed random variable to make sense, the expression E_2 must be fully deterministic. For this reason we require that $\textsc{free-vars}(E_2) \cap V = \emptyset$, which ensures that E_2 cannot reference any other random variables in the graphical model. Translation fails when this requirement is not met. Remember that FREE-VARS refers to all unbound variables in an expression. Also note an important consequence of E_2 being a value, namely, although the return value of an observe may be used in subsequent computation, no graphical model edges will be generated with the observed random variable as a parent. An alternative rule could return a `null` or `nil` value in place of E_2 and,

as a result, might potentially be "safer" in the sense of ensuring clarity to the programmer. Not being able to bind the observed value would mean that there is no way to imagine that an edge could be created where one was not.

Procedure Call The remaining two cases are those for procedure calls, one for a user-defined procedure f and one for a primitive function c. In both cases, we first translate arguments. In the case of primitive functions we then translate the expression for the call by substituting translated arguments into the original expression, and merging the graphs for the arguments

$$\frac{\rho,\, \phi,\, e_i \Downarrow G_i,\, E_i \text{ for all } 1 \leq i \leq n}{\rho,\, \phi,\, (c\, e_1 \ldots e_n) \Downarrow G_1 \oplus \ldots \oplus G_n,\, (c\, E_1 \ldots E_n)}$$

For user-defined procedures, we additionally transform the procedure body. We do this by replacing all instances of the variable v_i with the expression for the argument E_i

$$\frac{\begin{array}{ll}\rho,\, \phi,\, e_i \Downarrow G_i,\, E_i \text{ for all } 1 \leq i \leq n & \rho(f) = (\texttt{defn}\ f\ [v_1 \ldots v_n]\ e) \\ \rho,\, \phi,\, e[v_1 := E_1, \ldots v_n := E_n] \Downarrow G,\, E & \end{array}}{\rho,\, \phi,\, (f\ e_1 \ldots e_n) \Downarrow G_1 \oplus \ldots \oplus G_n \oplus G,\, E}$$

3.2 Evaluating the Density

Before we discuss algorithms for inference in FOPPL programs, first we make explicit how we can use this representation of a probabilistic program to evaluate the probability of a particular setting of the variables in V. The Bayesian network $G = (V, A, \mathcal{P}, \mathcal{Y})$ that we construct by compiling a FOPPL program is a mathematical representation of a directed graphical model. Like any graphical model, G defines a probability density on its variables V. In a directed graphical model, each node $v \in V$ has a set of parents

$$\text{PA}(v) := \{u : (u, v) \in A\}. \tag{3.6}$$

The joint probability of all variables can be expressed as a product over conditional probabilities

$$p(V) = \prod_{v \in V} p(v \mid \text{PA}(v)). \tag{3.7}$$

In our graph G, each term $p(v \mid \text{PA}(v))$ is represented as a deterministic expression $\mathcal{P}(v) = (c\ v\ E_1\ \ldots\ E_n)$, in which c is either a probability mass function (for discrete variables) or a probability density function (for continuous variables) and $E_1, \ldots, E_n$ are expressions that evaluate to parameters $\theta_1, \ldots, \theta_n$ of this mass or density function.

Implicit in this notation is the fact that each expression has some set of free variables. In order to evaluate an expression to a value, we must specify values for each of these free variables. In other words, we can think of each of these expressions E_i as a mapping from values of free variables to a parameter value. By construction, the set of parents $\text{PA}(v)$ is nothing but the free variables in $\mathcal{P}(v)$ exclusive of v

$$\text{PA}(v) = \text{FREE-VARS}(\mathcal{P}(v)) \setminus \{v\}. \tag{3.8}$$

Thus, the expression $\mathcal{P}(v)$ can be thought of as a function that maps the v and its parents $\text{PA}(v)$ to a probability or probability density. We will therefore from now on treat these two as equivalent,

$$p(v \mid \text{PA}(v)) \equiv \mathcal{P}(v). \tag{3.9}$$

We can decompose the joint probability $p(V)$ into a prior and a likelihood term. In our specification of the translation rule for observe, we require that the expression $\mathcal{Y}(v)$ for the observed value may not have free variables. Each expression $\mathcal{Y}(v)$ will hence simplify to a constant when we perform partial evaluation, a subject we cover extensively in Section 3.2.2 of this chapter. We will use Y to refer to all the nodes in V that correspond to observed random variables, which is to say $Y = \text{dom}(\mathcal{Y})$. Similarly, we can use X to refer to all nodes in V that correspond to unobserved random variables, which is to say $X = V \setminus Y$. Since observed nodes $y \in Y$ cannot have any children we can re-express the joint probability in Equation (3.7) as

$$p(V) = p(Y, X) = p(Y \mid X)p(X), \tag{3.10}$$

where

$$p(Y \mid X) = \prod_{y \in Y} p(y \mid \text{PA}(y)), \qquad p(X) = \prod_{x \in X} p(x \mid \text{PA}(x)). \tag{3.11}$$

In this manner, a probabilistic program defines a joint distribution $p(Y, X)$. The goal of probabilistic program *inference* is to characterize the posterior distribution

$$p(X \mid Y) = p(X, Y)/p(Y), \qquad p(Y) := \int dX \, p(X, Y). \tag{3.12}$$

3.2.1 Conditioning with Factors

Not all inference problems for probabilistic programs target a posterior $p(X \mid Y)$ that is defined in terms of unobserved and observed random variables. There are inference problems in which there is no notion of observed data, but it is possible to define some notion of loss, reward, or fitness given a choice of X. In the probabilistic programs written in the FOPPL, the `sample` statements in a probabilistic program define a prior $p(X)$ on the random variables, whereas the `observe` statements define a likelihood $p(Y \mid X)$. To support a more general notion of soft constraints, we can replace the likelihood $p(Y \mid X)$ with a strictly positive potential $\psi(X)$ to define an unnormalized density

$$\gamma(X) = \psi(X)p(X). \tag{3.13}$$

In this more general setting, the goal of inference is to characterize a target density $\pi(X)$, which we define as

$$\pi(X) := \gamma(X)/Z, \qquad Z := \int dX \, \gamma(X). \tag{3.14}$$

Here $\pi(X)$ is the analogue to the posterior $p(X \mid Y)$, the unnormalized density $\gamma(X)$ is the analogue to the joint $p(Y, X)$, and the normalizing constant Z is the analogue to the marginal likelihood $p(Y)$.

From a language design point of view, we can now ask how the FOPPL would need to be extended in order to support this more general form of soft constraint. For a probabilistic program in the FOPPL, the potential function is a product over terms

$$\psi(X) = \prod_{y \in Y} \psi_y(X_y), \tag{3.15}$$

where we define ψ_y and X_y as

$$\psi_y(X_y) := p(y{=}\mathcal{Y}(y) \mid \text{PA}(y)) \equiv \mathcal{P}(y)[y := \mathcal{Y}(y)] \tag{3.16}$$

$$X_y := \text{FREE-VARS}(\mathcal{P}(y)) \setminus \{y\} = \text{PA}(y). \tag{3.17}$$

Note that $\mathcal{P}(y)[y := \mathcal{Y}(y)]$ is just some expression that evaluates to either a probability mass or a probability density if we specify values for its free variables X_y. Since we never integrate over y, it does not matter whether $\mathcal{P}(y)$ represents a (normalized) mass or density function. We could therefore in principle replace $\mathcal{P}(y)$ by any other expression with free variables X_y that evaluates to a number ≥ 0.

One way to support arbitrary potential functions is to provide a special form `(factor log-p)` that takes an arbitrary log probability `log-p` (which can be both positive or negative) as an argument. We can then define a translation rule that inserts a new node v with probability $\mathcal{P}(v) =$ `(exp log-p)` and observed value `nil` into the graph:

$$\frac{\rho,\ \phi,\ e \Downarrow (V, A, \mathcal{P}, \mathcal{Y}),\ E \qquad F = \texttt{(if } \phi \texttt{ (exp } E\texttt{) 1)} \qquad \text{Choose a fresh variable } v}{\rho,\ \phi,\ \texttt{(factor } e\texttt{)} \Downarrow (V,\ A,\ \mathcal{P} \oplus [v \mapsto F],\ \mathcal{Y} \oplus [v \mapsto \texttt{nil}]),\ \texttt{nil}}$$

In practice, we don't need to provide separate special forms for `factor` and `observe`, since each can be implemented as a special case of the other. One way of doing so is to define `factor` as a procedure

```
(defn factor [log-p]
  (observe (factor-dist log-p) nil))
```

in which `factor-dist` is a constructor for a "pseudo" distribution object with corresponding potential

$$p_{\text{factor-dist}}(y; \lambda) := \begin{cases} \exp\lambda & y = \texttt{nil} \\ 0 & y \neq \texttt{nil} \end{cases} \tag{3.18}$$

We call this a pseudo distribution, because it defines a (unnormalized) potential function, rather than a normalized mass or density.

Had we defined the FOPPL language using `factor` as the primary conditioning form, then we could have implemented a primitive procedure `(log-prob dist v)` that returns the log probability mass or density for a value `v` under a distribution `dist`. This would then allow us to define `observe` as a procedure

```
(defn observe [dist v]
  (factor (log-prob dist v))
  y)
```

3.2.2 Partial Evaluation

An important and necessary optimization for our compilation procedure is to perform a partial evaluation step. This step pre-evaluates expressions E in the target language that do not contain any free variables, which means that they take on the same value in every execution of the program. It turns out that partial evaluation of these expressions is necessary to avoid the appearance of "spurious" edges between variables that are in fact not connected, in the sense that the value of the parent does not affect the conditional density of the child.

Because our target language is very simple, we only need to consider if-expressions and procedure calls. We can update the compilation rules for these expressions to include a partial evaluation step

$$\frac{\rho,\, \phi,\, e_1 \Downarrow G_1,\, E_1 \qquad \rho,\, \textsc{eval}(\texttt{(and}\ \phi\ E_1\texttt{)}),\, e_2 \Downarrow G_2,\, E_2 \qquad \rho,\, \textsc{eval}(\texttt{(and}\ \phi\ \texttt{(not}\ E_1\texttt{))}),\, e_3 \Downarrow G_3,\, E_3}{\rho,\, \phi,\, \texttt{(if}\ e_1\ e_2\ e_3\texttt{)} \Downarrow (G_1 \oplus G_2 \oplus G_3),\, \textsc{eval}(\texttt{(if}\ E_1\ E_2\ E_3\texttt{)})}$$

and

$$\frac{\rho,\, e_i \Downarrow G_i,\, E_i \text{ for all } 1 \leq i \leq n}{\rho,\, \phi,\, \texttt{(}c\ e_1 \ldots e_n\texttt{)} \Downarrow G_1 \oplus \ldots \oplus G_n,\, \textsc{eval}(\texttt{(}c\ E_1 \ldots E_n\texttt{)})}$$

The partial evaluation operation $\textsc{eval}(e)$ can incorporate any number of rules for simplifying expressions. We will begin with the rules

$\textsc{eval}(\texttt{(if}\ c_1\ E_2\ E_3\texttt{)}) = E_2$
when c_1 is logically true

$\textsc{eval}(\texttt{(if}\ c_1\ E_2\ E_3\texttt{)}) = E_3$
when c_1 is logically false

$\textsc{eval}(\texttt{(}c\ c_1\ \ldots\ c_n\texttt{)}) = c'$
when calling c with arguments $c_1, \ldots, c_n$ evaluates to c'

$\textsc{eval}(E) = E$
in all other cases

These rules state that an if statement (`if` E_1 E_2 E_3) can be simplified when $E_1 = c_1$ can be fully evaluated, by simply selecting the expression for the appropriate branch. Primitive procedure calls can be evaluated when all arguments can be fully evaluated.

In order to accommodate partial evaluation, we additionally modify the definition of the SCORE function. Distributions in the FOPPL are constructed using primitive procedure applications. This means that a distribution with constant arguments such as (`beta 1 1`) will be partially evaluated to a constant c. To account for this, we need to extend our definition of the SCORE conversion with one rule

$$\text{SCORE}(c,\, v) = (p_c\ v)$$
(when c is a distribution and p_c is its pdf or pmf)

To see how partial evaluation also reduce the number of edges in the Bayesian network, let us consider the expression (`if true` v_1 v_2). This expression nominally references two random variables v_1 and v_2. After partial evaluation, this expression simplifies to v_1, which eliminates the spurious dependence on v_2.

Another practical advantage of partial evaluation is that it gives us a simple way to identify expressions in a program that are fully deterministic (since such expression will be partially evaluated to constants). This is useful when translating observe statements (`observe` e_1 e_2), in which the expression e_2 must be deterministic. In programs that use the (`loop` c v e e_1 ... e_n) syntactic sugar, we can now substitute any fully deterministic expression for the number of loop iterations c. For example, we could define a loop in which the number of iterations is given by the dataset size.

Lists, vectors and hash maps. Eliminating spurious edges in the dependency graph becomes particularly important in programs that make use of data structures. Let us consider the following example, which defines a 3-state Markov chain

```
(let [A [[0.9 0.1]
         [0.1 0.9]]
      x1 (sample (discrete [1. 1.]))
      x2 (sample (discrete (get A x1)))
      x3 (sample (discrete (get A x2)))]
```

```
  [x1 x2 x3])
```

Compilation to a Bayesian network will yield three variable nodes. If we refer to these nodes as v_1, v_2 and v_3, then there will be arcs from v_1 to v_2 and from v_2 to v_3. Suppose we now rewrite this program using the `loop` syntactic sugar that we introduced in Chapter 2

```
(defn markov-step
  [n xs A]
  (let [k (last xs)
        Ak (get A k)]
    (append xs (sample (discrete Ak)))))

(let [A [[0.9 0.1]
         [0.1 0.9]]
      x1 (sample (discrete [1. 1.]))]
  (loop 2 markov-step [x1] A))
```

In this version of the program, each call to `markov-step` accepts a vector of states `xs` and appends the next state in the Markov chain by calling `(append xs (sample (discrete Ak)))`. In order to sample the next element, we need the row `Ak` for the transition matrix that corresponds to the current state `k`, which is retrieved by calling `(last xs)` to extract the last element of the vector.

The program above generates the same sequence of random variables as the previous one, and has the advantage of allowing us to generalize to sequences of arbitrary length by changing the constant `2` in the loop to a different value. However, under the partial evaluation rules that we have specified so far, we would obtain a different set of edges. As in the previous version of the program, this version of the program evaluates three sample statements. For the first statement, `(sample (discrete [1. 1.]))` there will be no arcs. Translation of the second sample statement `(sample (discrete Ak))`, which is evaluated in the body of `markov-step`, results in an arc from v_1 to v_2, since the expression for `Ak` expands to

```
(get [[0.9 0.1]
      [0.1 0.9]]
      (last [v1]))
```

However, for the third sample statement there will be arcs from both v_1 and v_2 to v_3, since `Ak` expands to

```
(get [[0.9 0.1]
      [0.1 0.9]]
      (last (append [v1] v2)))
```

The extra arc from v_1 to v_3 is of course not necessary here, since the expression `(last (append [`v_1`] `v_2`))` will always evaluate to v_2. What's more, if we run this program to generate more than 3 states, the node v_n for the n-th state will have incoming arcs from all preceding variables $v_1, \ldots, v_{n-1}$, whereas the only real arc in the Bayesian network is the one from v_{n-1}.

We can eliminate these spurious arcs by implementing an additional set of partial evaluation rules for data structures,

$$\begin{aligned}
&\textsc{eval}(\texttt{(vector}\ E_1\ \ldots\ E_n\texttt{)}) = \texttt{[}E_1\ \ldots\ E_n\texttt{]},\\
&\textsc{eval}(\texttt{(hash-map}\ c_1\ E_1\ \ldots\ c_n\ E_n\texttt{)}) = \texttt{\{}c_1\ E_1\ \ldots\ c_n\ E_n\texttt{\}}.
\end{aligned}$$

These rules ensure that expressions which construct data structures are partially evaluated to data structures containing expressions. We can similarly partially evaluate functions that add or replace entries. For example, we can define the following rules for the `conj` primitive, which appends an element to a data structure,

$$\begin{aligned}
&\textsc{eval}(\texttt{(append [}E_1\ \ldots\ E_n\texttt{]}\ E_{n+1}\texttt{)}) = \texttt{[}E_1\ \ldots\ E_n\ E_{n+1}\texttt{]},\\
&\textsc{eval}(\texttt{(put \{}c_1\ E_1\ \ldots\ c_n\ E_n\texttt{\}}\ c_k\ E'_k\texttt{)}) = \texttt{\{}c_1\ E_1\ \ldots\ c_k\ E'_k\ \ldots\ c_n\ E_n\texttt{\}}.
\end{aligned}$$

In the Markov chain example, the expression for `Ak` in the third sample statement then simplifies to

```
(get [[0.9 0.1]
      [0.1 0.9]]
      (last [v1 v2]))
```

Now that partial evaluation constructs data structures containing expressions, we can use partial evaluation of accessor functions to extract

the expression corresponding to an entry

$$\text{EVAL}(\texttt{(last [}E_1 \ \ldots \ E_n\texttt{])}) = E_n,$$

$$\text{EVAL}(\texttt{(get [}E_1 \ \ldots \ E_n\texttt{] }k\texttt{)}) = E_k,$$
$$\text{EVAL}(\texttt{(get \{}c_1 \ E_1 \ \ldots \ c_n \ E_n\texttt{\}} \ c_k\texttt{)}) = E_k.$$

With these rules in place, the expression for `Ak` simplifies to

```
(get [[0.9 0.1]
      [0.1 0.9]] v2)
```

This yields the correct dependency structure for the Bayesian network.

3.3 Gibbs Sampling

So far, we have just defined a way to translate probabilistic programs into a data structure for finite graphical models. One important reason for doing so is that many existing inference algorithms are defined explicitly in terms of finite graphical models, and can now be applied directly to probabilistic programs written in the FOPPL. We will consider such algorithms now, starting with a general family of Markov chain Monte Carlo (MCMC) algorithms.

MCMC algorithms perform Bayesian inference by drawing samples from the posterior distribution; that is, the conditional distribution of the latent variables $X \subseteq V$ given the observed variables $Y \subset V$. This is accomplished by simulating from a Markov chain whose transition operator is defined such that the stationary distribution is the target posterior $p(X \mid Y)$. These samples are then used to characterize the distribution of the return value $r(X)$.

Procedurally, MCMC algorithms begin by initializing latent variables to some value $X^{(0)}$, and repeatedly sampling from a Markov transition density to produce a dependent sequence of samples $X^{(1)}, \ldots, X^{(S)}$. For purposes of this tutorial, we will not delve deeply into *why* MCMC produces posterior samples; rather, we will simply describe how these algorithms can be applied in the context of inference in graphs produced by FOPPL compilation in the previous sections. For a review of MCMC methods, see e.g. Neal (1993), Gelman et al. (2013), or Bishop (2006).

The Metropolis-Hastings (MH) algorithm provides a general recipe for producing appropriate MCMC transition operators, by combining a proposal step with an accept / reject step. Given some appropriate proposal distribution $q(X' \mid V)$, the MH algorithm simulates a candidate X' from $q(X' \mid V)$ conditioned on the value of the current sample X, and then evaluates the acceptance probability

$$\alpha(X', X) = \min\left\{1, \frac{p(Y, X')q(X \mid V')}{p(Y, X)q(X' \mid V)}\right\}. \tag{3.19}$$

With probability $\alpha(X', X)$, we "accept" the transition $X \to X'$ and with probability $1 - \alpha(X', X)$ we "reject" the transition and retain the current sample $X \to X$. When we repeatedly apply this transition operator we obtain a Markov process

$$\begin{matrix} X' \sim q(X' \mid V^{(s-1)}), \\ u \sim \text{Uniform}(0, 1) \end{matrix} \qquad X^{(s)} = \begin{cases} X' & u \leq \alpha(X', X^{(s-1)}), \\ X^{(s-1)} & u > \alpha(X', X^{(s-1)}). \end{cases}$$

Gibbs sampling algorithms (Geman and Geman, 1984) are an important special case of MH, which cycle through all the latent variables in the model and iteratively sample from the so-called full conditional distributions

$$p(x \mid Y, X \setminus \{x\}) = p(x \mid V \setminus \{x\}). \tag{3.20}$$

In some (important) special cases of models, these conditional distributions can be derived analytically and sampled from exactly. However, this is not possible in general, and so as a general-purpose solution one turns to *Metropolis-within-Gibbs* algorithms, which instead apply a Metropolis-Hastings transition targeting $p(x \mid V \setminus \{x\})$.

From an implementation point of view, given our compiled graph $(V, A, \mathcal{P}, \mathcal{Y})$ we can compute the acceptance probability in Equation (3.19) by evaluating the expressions $\mathcal{P}(v)$ for each $v \in V$, substituting the values for the current sample X and the proposal X'. More precisely, if we use X to refer to the set of unobserved variables and $\mathcal{X}$ to refer to the map from variables to their values,

$$X = (x_1, \ldots, x_N), \qquad \mathcal{X} = [x_1 \mapsto c_1, \ldots, x_N \mapsto c_N], \tag{3.21}$$

then we can use $\mathcal{V} = \mathcal{X} \oplus \mathcal{Y}$ to refer to the values of all variables and express the joint probability over the variables V as

$$p(V = \mathcal{V}) = \prod_{v \in V} \textsc{eval}(\mathcal{P}(v)[V := \mathcal{V}]). \tag{3.22}$$

When we update a single variable x using a kernel $q(x \mid V)$, we are proposing a new mapping $\mathcal{V}' = \mathcal{V}[x \mapsto c']$, where c' is the candidate value proposed for x. The acceptance probability for changing the value of x from c to c' then takes the form

$$\alpha(\mathcal{V}', \mathcal{V}) = \min\left\{1, \frac{p(V = \mathcal{V}')q(x = c \mid V = \mathcal{V}')}{p(V = \mathcal{V})q(x = c' \mid V = \mathcal{V})}\right\}. \tag{3.23}$$

From a computational point of view, the important thing to note is that many terms in this ratio will actually cancel out. The joint probabilities $p(V = \mathcal{V})$ are composed of a product of conditional density terms $\prod_{v \in V} p(v \mid \textsc{pa}(v))$; the individual expressions $p(v \mid \textsc{pa}(v)) \equiv \mathcal{P}(v)$ depend on the value c or its proposed alternative c' of the node x only if $v = x$, or $x \in \textsc{pa}(v)$, which equates to the condition

$$x \in \textsc{free-vars}(\mathcal{P}(v)). \tag{3.24}$$

If we define V_x to be the set of variables whose densities depend on x,

$$\begin{aligned} V_x &:= \{v : x \in \textsc{free-vars}(\mathcal{P}(v))\}, \\ &= \{x\} \cup \{v : x \in \textsc{pa}(v)\}, \end{aligned} \tag{3.25}$$

then we can decompose the joint $p(V)$ into terms that depend on x and terms that do not

$$p(V) = \left(\prod_{w \in V \setminus V_x} p(w \mid \textsc{pa}(w))\right)\left(\prod_{v \in V_x} p(v \mid \textsc{pa}(v))\right).$$

We now note that all terms $w \in V \setminus V_x$ in the acceptance ratio cancel, with the same value in both numerator and denominator. Denoting the values of a variable v as c_v, c'_v for the maps $\mathcal{V}, \mathcal{V}'$ respectively, we can simplify the acceptance probability α to

$$\alpha(\mathcal{V}', \mathcal{V}) = \min\left\{1, \frac{\prod_{v \in V_x} p(v = c'_v | \textsc{pa}(v))}{\prod_{v \in V_x} p(v = c_v | \textsc{pa}(v))} \frac{q(x = c \mid V = \mathcal{V}')}{q(x = c' \mid V = \mathcal{V})}\right\}. \tag{3.26}$$

This restriction means that we can compute the acceptance ratio in $O(|V_x|)$ time rather than $O(|V|)$, which is advantageous when $|V|$ grows with the size of the dataset, whereas $|V_x|$ does not.

In order to implement a Gibbs sampler, we additionally need to specify some form of proposal. We will here assume a map $\mathcal{Q}$ from unobserved variables to expressions in the target language

$$\mathcal{Q} := [x_1 \mapsto E_1, \ldots, x_N \mapsto E_N]. \tag{3.27}$$

For each variable x, the expression E defines a distribution, which can in principle depend on other unobserved variables X. We can then use this distribution to both generate samples and evaluate the forward and reverse proposal densities $q(x = c' \mid V = \mathcal{V})$ and $q(x = c \mid V = \mathcal{V}')$. To do so, we first evaluate the expression to a distribution

$$d = \textsc{eval}(\mathcal{Q}(x)[V := \mathcal{V}]). \tag{3.28}$$

We then assume that we have an implementation for functions SAMPLE and LOG-PROB which allow us to generate samples and evaluate the density function for the distribution

$$c' = \textsc{sample}(d), \qquad q(x = c' \mid V) = \textsc{log-prob}(d, c'). \tag{3.29}$$

Algorithm 1 shows pseudo-code for a Gibbs sampler with this type of proposal. In this algorithm we have several choices for the type of proposals that we define in the map $\mathcal{Q}$. A straightforward option is to use the prior as the proposal distribution. In other words, when compiling an expression `(sample` e`)` we first compile e to a target language expression E, then pick a fresh variable v, define $\mathcal{P}(v) = \textsc{score}(E, v)$, and finally define $\mathcal{Q}(v) = E$. In this type of proposal $q(x = c' \mid X) = p(x = c' \mid \textsc{pa}(x))$, which means that the acceptance ratio simplifies further to

$$\alpha(\mathcal{V}', \mathcal{V}) = \min\left\{1, \frac{\prod_{v \in V_x \setminus \{x\}} p(v = c'_v | \textsc{pa}(v))}{\prod_{v \in V_x \setminus \{x\}} p(v = c_v | \textsc{pa}(v))}\right\}. \tag{3.30}$$

Instead of proposing from the prior, we can also consider a broader class of proposal distributions. For example, a common choice for continuous random variables is to propose from a Gaussian distribution with small standard deviation, centered at the current value of x; there exist

Algorithm 1 Gibbs Sampling with Metropolis-Hastings Updates

global $V, X, Y, A, \mathcal{P}, \mathcal{Y}$ ▷ A directed graphical model
global $\mathcal{Q}$ ▷ A map of proposal expressions
function ACCEPT($x, \mathcal{X}', \mathcal{X}$)
    $d \leftarrow \text{EVAL}(\mathcal{Q}(x)[X := \mathcal{X}])$
    $d' \leftarrow \text{EVAL}(\mathcal{Q}(x)[X := \mathcal{X}'])$
    $\log \alpha \leftarrow \text{LOG-PROB}(d', \mathcal{X}(x)) - \text{LOG-PROB}(d, \mathcal{X}'(x))$
    $V_x \leftarrow \{v : x \in \text{FREE-VARS}(\mathcal{P}(v))\}$
    for v **in** V_x **do**
        $\log \alpha \leftarrow \log \alpha + \text{EVAL}(\mathcal{P}(v)[Y := \mathcal{Y}, X := \mathcal{X}'])$
        $\log \alpha \leftarrow \log \alpha - \text{EVAL}(\mathcal{P}(v)[Y := \mathcal{Y}, X := \mathcal{X}])$
    return α
function GIBBS-STEP($\mathcal{X}$)
    for x **in** X **do**
        $d \leftarrow \text{EVAL}(\mathcal{Q}(x)[X := \mathcal{X}])$
        $\mathcal{X}' \leftarrow \mathcal{X}$
        $\mathcal{X}'(x) \leftarrow \text{SAMPLE}(d)$
        $\alpha \leftarrow \text{ACCEPT}(x, \mathcal{X}', \mathcal{X})$
        $u \sim \text{Uniform}(0, 1)$
        if $u < \alpha$ **then**
            $\mathcal{X} \leftarrow \mathcal{X}'$
    return $\mathcal{X}$
function GIBBS($\mathcal{X}^{(0)}, S$)
    for s **in** $1, \ldots, S$ **do**
        $\mathcal{X}^{(s)} \leftarrow \text{GIBBS-STEP}(\mathcal{X}^{(s-1)})$
return $\mathcal{X}^{(1)}, \ldots \mathcal{X}^{(S)}$

schemes to tune the standard deviation of this proposal online during sampling (Łatuszyński et al., 2013). In this case, the proposal is symmetric, which is to say that $q(x' \mid x) = q(x \mid x')$, which means that the acceptance probability simplifies to the same form as in Equation (3.30).

A second extension involves "block sampling", in which multiple random variables nodes are sampled jointly, rather than cycling through and updating only one at a time. This can be very advantageous in cases

where two latent variables are highly correlated: when updating one conditioned on a fixed value of the other, it is only possible to make very small changes at a time. wIn contrast, a block proposal which updates both these random variables at once can move them larger distances, in sync. As a pathological example, consider the FOPPL program

```
(let [x0 (sample (normal 0 1))
      x1 (sample (normal 0 1))]
  (observe (normal (+ x0 x1) 0.01) 2.0))
```

in which we observe the sum of two standard normal random variates is very close to 2.0. If initialized at any particular pair of values (x_0, x_1) for which $x_0 + x_1 \approx 2.0$, a Gibbs sampler which updates one random choice at a time will quickly become "stuck".

Consider instead a proposal which updates a subset of latent variables $S \subseteq X$, according to a proposal $q(S \mid V \setminus S)$. The "trivial" choice of proposal distribution — proposing values of each random variable x in S by simulating from the prior $p(x \mid \text{PA}(x))$ — would, for $S = \{x_0, x_1\}$ in this example, sample both values from their independent normal priors. While this is capable of making larger moves (unlike the previous one-choice-at-a-time proposal, it would be possible for this proposal to go in a single step from e.g. $(2.0, 0.0)$ to $(0.0, 2.0)$), with this naïve choice of block proposal overall performance may actually be worse than that with independent proposals: now instead of sampling a single value which needs to be "near" the previous value to be accepted, we are sampling two values, where the second value x_1 needs to be "near" the sampled $x_0 - 2.0$, something quite unlikely for negative values of x_0. Constructing block proposals which have high acceptance rates require taking account of the structure of the model itself. One way of doing this adaptively, analogous to estimating posterior standard deviations to be used as scale parameters in univariate proposals, is to estimate posterior covariance matrices and using these for jointly proposing multiple latent variables (Haario et al., 2001).

As noted already, it is sometimes possible to analytically derive the complete conditional distribution of a single variable in a graphical model. Such cases include all random variables whose value is discrete from a finite set, many settings in which all the densities in V_x are in the

exponential family, and so forth. Programming languages techniques can be used to identify such opportunities by performing pattern matching analyses of the source code of the link functions in V_x. If, as is the case in the simplest example, x itself is determined to be governed by a discrete distribution then, instead of using Metropolis within Gibbs, one would merely enumerate all possible values of x under the support of its distribution, score each, normalize, then sample from this exact conditional.

Inference algorithms vary in their performance, sometimes dramatically. Metropolis Hastings within Gibbs is sometimes efficient but even more often is not, unless utilizing intelligent block proposals (often, ones customized to the particular model). This has led to a proliferation of inference algorithms and methods, some but not all of which are directly applicable to probabilistic programming. In the next section, we consider Hamiltonian Monte Carlo, which incorporates gradient information to provide efficient high-dimensional block proposals.

3.4 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC) is a MCMC algorithm that makes use of gradients to construct an efficient MCMC kernel for high dimensional distributions. The HMC algorithm applies to a target density $\pi(X) = \gamma(X)/Z$ of the general form defined in Equation (3.14), in which each variable $x \in X$ is continuous. This unnormalized density is traditionally re-expressed by defining a *potential energy* function

$$U(X) := -\log \gamma(X). \tag{3.31}$$

With this definition we can write

$$\pi(X) = \frac{1}{Z} \exp\{-U(X)\}. \tag{3.32}$$

Next, we introduce a set of auxiliary "momentum" variables R, one for each variable in $x \in X$, together with a function $K(R)$ representing the *kinetic energy* of the system. These variables are typically defined as samples from a zero-mean Gaussian with covariance M. This choice of

K then yields a joint target distribution $\pi'(X, R)$ defined as follows:

$$\begin{aligned}\pi'(X, R) &= \frac{1}{Z'} \exp\{-U(X) - K(R)\} \\ &= \frac{1}{Z'} \exp\left\{-U(X) + \frac{1}{2} R^\top M^{-1} R\right\}.\end{aligned}$$

Since marginalizing over R in π' recovers the original density π, we can jointly sample X and R from π' to generate samples X from π. The central idea in HMC is to construct an MCMC kernel that changes (X, R) in a way that preserves the Hamiltonian $H(X, R)$, which describes the total energy of the system

$$H(X, R) = U(X) + K(R). \tag{3.33}$$

By way of physical analogy, an HMC sampler constructs samples by simulating the trajectory of a "marble" with position X and a momentum R as it rolls through an energy "landscape" defined by $U(X)$. When moving "uphill" in the direction of the gradient $\nabla U(X)$, the marble loses momentum. Conversely when moving "downhill" (i.e. away from $\nabla U(X)$), the marble gains momentum. By requiring that the total energy H is constant, we can derive the equations of motion

$$\begin{aligned}\frac{dX}{dt} &= \nabla_R H(X, R) = M^{-1} R, \\ \frac{dR}{dt} &= -\nabla_X H(X, R) = -\nabla_X U(X).\end{aligned} \tag{3.34}$$

That is, paths of X and R that solve the above differential equations preserve the total energy $H(X, R)$. The HMC sampler proceeds by alternately sampling the momentum variables R, and then simulating (X, R) forward according to a discretized version of the above differential equations. Since $\pi'(X, R)$ factorizes into a product of independent distributions on X and R, the momentum variables R are simply sampled in a Gibbs step according to their full conditional distribution (i.e. the marginal distribution) of Normal$(0, M)$. The forward simulation (called Hamiltonian dynamics) generates a new proposal (X', R'), which then is accepted with probability

$$\begin{aligned}\alpha &= \min\left\{1, \frac{\pi'(X', R')}{\pi'(X, R)}\right\} \\ &= \min\left\{1, \exp\left\{-H(X', R') + H(X, R)\right\}\right\}.\end{aligned}$$

Note here that if when were able to perfectly integrate the equations of motion in (3.34), then the sample is accepted with probability 1. In other words, the acceptance ratio purely is purely due to numerical errors that arise from the discretization of the equations of motion.

3.4.1 Automatic Differentiation

The essential operation that we need to implement in order to perform HMC for either a suitably restricted block proposal in the Metropolis-within-Gibbs FOPPL inference algorithm from Section 3.3 of this chapter or another suitably restricted FOPPL-like language (Gram-Hansen et al., 2018; Stan Development Team, 2014) is the computation of the gradient

$$\nabla U(X) = -\nabla \log \gamma(X). \tag{3.35}$$

When $\gamma(X)$ is the density associated with a probabilistic program, we must take steps to ensure that this density is *differentiable* at all points $X = \mathcal{X}$ in the support of the distribution, noting that the class of all FOPPL programs includes conditional branching statements which renders HMC incompatible with whole FOPPL program inference. We will further discuss what implications this has for the structure of a program in Section 3.4.2. For now we will assume that $\gamma(X)$ is indeed differentiable everywhere, and either refers to a joint $p(Y, X)$ over continuous variables, or that we are using HMC as a block Gibbs update to sample a subset of continuous variables $X_{\mathrm{b}} \subset X$ from the conditional $p(X_{\mathrm{b}} \mid Y, X \setminus X_{\mathrm{b}})$.

Given that $\gamma(X)$ is differentiable, how can we compute the gradient? For a program with graph $(V, A, \mathcal{P}, \mathcal{Y})$ and variables $V = Y \cup X$, the component of the gradient for a variable $x \in X$ is

$$\begin{aligned} \nabla_x U(X) &= -\frac{\partial \log p(Y, X)}{\partial x} \\ &= -\sum_{x' \in X} \frac{\partial \log p(x' | \mathrm{PA}(x'))}{\partial x} - \sum_{y \in Y} \frac{\partial \log p(y | \mathrm{PA}(y))}{\partial x}. \end{aligned} \tag{3.36}$$

In our graph compilation procedure, we have constructed expressions $p(x|\mathrm{PA}(x)) \equiv \mathcal{P}(x)$ and $p(y|\mathrm{PA}(y)) \equiv \mathcal{P}(y)[y := \mathcal{Y}(y)]$ for each random

variable. In order to calculate $\nabla U(X)$ we will first construct a single expression E_U that represents the potential $U(X) \equiv E_U$ as an explicit sum over the variables $X = \{x_1, \ldots, x_N\}$ and $Y = \{y_1, \ldots, y_M\}$

$E_X :=$`(+ (log` $\mathcal{P}(x_1)$`) ... (log` $\mathcal{P}(x_N)$`))`

$E_Y :=$`(+ (log` $\mathcal{P}(y_1)[y_1 := \mathcal{Y}(y_1)]$`) ... (log` $\mathcal{P}(y_M)[y_M := \mathcal{Y}(y_M)]$`))`

$E_U :=$`(* -1.0 (+` E_X E_Y`))`

We can then define point-wise evaluation of the potential function by means of the partial evaluation operation EVAL after substitution of a map $\mathcal{X}$ of values

$$U(X = \mathcal{X}) := \text{EVAL}(E_U[X := \mathcal{X}]). \tag{3.37}$$

In order to compute the gradients of the potential function, we will use reverse-mode automatic differentiation (AD) (Griewank and Walther, 2008; Baydin et al., 2015), which is the technique that forms the basis for modern deep learning systems such as TensorFlow (Abadi et al., 2015), PyTorch (Paszke et al., 2017), MxNET (Chen et al., 2016), and CNTK (Seide and Agarwal, 2016).

To perform reverse-mode AD, we augment all real-valued primitive procedures in a program with primitives for computing the partial derivatives with respect to each of the inputs. We additionally construct a data structure that represents the computation as a graph. This graph contains a node for each primitive procedure application and edges for each of its inputs. There are a number of ways to construct such a computation graph. In Section 3.5 we will show how to compile a Bayesian network to a factor graph. This graph will also contain a node for each primitive procedure application, and edges for each of its inputs. In this section, we will compute this graph dynamically as a side-effect of evaluation of an expression E in the target language.

Suppose that E contains the free variables $V = \{v_1, \ldots, v_D\}$. We can think of this expression as a function $E \equiv F(v_1, \ldots, v_D)$. Suppose that we wish to compute the gradient of F at values

$$\mathcal{V} = [v_1 \mapsto c_1, \ldots, v_D \mapsto c_D], \tag{3.38}$$

Our goal is now to to define a gradient operator

$$\text{GRAD}\left(\text{EVAL}\left(F[V := \mathcal{V}]\right)\right), \tag{3.39}$$

which computes the map of partial derivatives

$$\mathcal{G} := \left[v_1 \mapsto \left.\frac{\partial F(V)}{\partial v_1}\right|_{V=\mathcal{V}}, \ldots, v_D \mapsto \left.\frac{\partial F(V)}{\partial v_D}\right|_{V=\mathcal{V}} \right]. \tag{3.40}$$

Given that F is an expression in the target language, we can solve the problem of differentiating F by defining the derivative of each expression type E recursively in terms of the derivatives of its sub-expressions. To do so, we only need to consider 4 cases:

1. Constants $E = c$ have zero derivatives
$$\frac{\partial E}{\partial v_i} = 0. \tag{3.41}$$

2. Variables $E = v$ have derivatives
$$\frac{\partial E}{\partial v_i} = \begin{cases} 1 & v = v_i, \\ 0 & v \neq v_i. \end{cases} \tag{3.42}$$

3. For if expressions $E =$ `(if` E_1 E_2 E_3`)` we can define the derivative recursively in terms of the value c_1' of E_1,
$$\frac{\partial E}{\partial v_i} = \begin{cases} \partial E_2 \,/\, \partial v_i & c_1' = \texttt{true} \\ \partial E_3 \,/\, \partial v_i & c_1' = \texttt{false} \end{cases} \tag{3.43}$$

4. For primitive procedure applications $E =$`(`f E_1 $\ldots$ E_n`)` we apply the chain rule
$$\frac{\partial E}{\partial v_i} = \sum_{j=1}^{n} \frac{\partial f(v_1', \ldots, v_n')}{v_j'} \frac{\partial E_j}{\partial v_i}. \tag{3.44}$$

The first 3 base cases are trivial. This means that we can compute the gradient of any target language expression E with respect to the values of its free variables as long as we are able calculate the partial derivatives of values returned by primitive procedure applications with respect to the values of the inputs.

Let us discuss this last case in more detail. Suppose that f is a primitive that accepts n real-valued inputs, and returns a real-valued output $c = f(c_1, \ldots, c_n)$. In order to perform reverse-mode AD, we will

Algorithm 2 Primitive function lifting for reverse-mode AD.

function UNBOX($\tilde{c}$)
  if $\tilde{c} = (c, \_)$ **then**
    return c ▷ Unpack value from $\tilde{c}$
  else
    return $\tilde{c}$ ▷ Return value as is
function LIFT-AD(f, ∇f, n)
  function $\tilde{f}(\tilde{c}_1, \ldots, \tilde{c}_n)$
    $c_1, \ldots, c_n \leftarrow \text{UNBOX}(\tilde{c}_1), \ldots, \text{UNBOX}(\tilde{c}_n)$
    $c \leftarrow f(c_1, \ldots, c_n)$
    $\dot{c}_1, \ldots, \dot{c}_n \leftarrow \nabla f(c_1, \ldots, c_n)$
    return $(c, ((\tilde{c}_1, \ldots, \tilde{c}_n), (\dot{c}_1, \ldots, \dot{c}_n)))$
  return $\tilde{f}$

replace this primitive with a "lifted" variant $\tilde{f}$ such that $\tilde{c} = \tilde{f}(\tilde{c}_1, \ldots, \tilde{c}_n)$ will return a boxed value

$$\tilde{c} = (c, ((\tilde{c}_1, \ldots, \tilde{c}_n), (\dot{c}_1, \ldots, \dot{c}_n))), \tag{3.45}$$

which contains the return value c of f, the input values $\tilde{c}_i$, and the values of the partial derivatives $\dot{c}_i = \partial f(v_1, \ldots, v_n)/\partial v_i|_{v_i=c_i}$ of the output with respect to the inputs. Algorithm 2 shows pseudo-code for an operation that constructs an AD-compatible primitive $\tilde{f}$ from a primitive f and a second primitive ∇f that computes the partial derivatives of f with respect to its inputs.

The boxed value $\tilde{c}$ is a recursive data structure that we can use to walk the computation graph. Each of the input values $\tilde{c}_i$ corresponds the value of a sub-expression that is either a constant, a variable, or the return value of another primitive procedure application. The first two cases correspond leaf nodes in the computation graph. In the case of a variable v (Equation 3.42), we are at an input where the gradient is 1 for the component associated with v, and 0 for all other components. We represent this sparse vector as a map $G = [v \mapsto 1]$. When we reach a constant value (Equation 3.41), we do don't need to do anything, since the gradient of a constant is 0. We represent this zero gradient as an empty map $G = [\,]$. In the third case (Equation 3.44), we can recursively

Algorithm 3 Reverse-mode automatic differentiation.

```
1:  function GRAD(c̃)
2:      match c̃                                  ▷ Pattern match against value type
3:          case (c, v)                                              ▷ Input value
4:              return [v ↦ 1]
5:          case (c, ((c̃_1, ..., c̃_n), (ċ_1, ..., ċ_n)))          ▷ Intermediate value
6:              G ← [ ]
7:              for i in 1, ..., n do
8:                  G_i ← GRAD(c̃_i)
9:                  for v in dom(G_i) do
10:                     if v ∈ dom(G) then
11:                         G(v) ← G(v) + ċ_i · G_i(v)
12:                     else
13:                         G(v) ← ċ_i · G_i(v)
14:             return G
15:     return [ ]                              ▷ Base case, return zero gradient
```

unpack the boxed values $\tilde{c}_i$ to compute gradients with respect to input values, multiply the resulting gradient terms partial derivatives $\dot{c}_i$, and finally sum over i.

Algorithm 3 shows pseudo-code for an algorithm that performs the reverse-mode gradient computation according to this recursive strategy. In this algorithm, we need to know the variable v that is associated with each of the inputs. In order to ensure that we can track these correspondences we will box an input value c associated with variable v into a pair $\tilde{c} = (c, v)$. The gradient computation in Algorithm 3 now pattern matches against values $\tilde{c}$ to determine whether the value is an input, and intermediate value that was returned from a primitive procedure call, or any other value (which has a 0 gradient).

Given this implementation of reverse-mode AD, we can now compute the gradient of the potential function in two steps

$$\tilde{U} = \textsc{eval}(E_U[V := \mathcal{V}]), \tag{3.46}$$

$$\mathcal{G} = \textsc{grad}(\tilde{U}). \tag{3.47}$$

Algorithm 4 Hamiltonian Monte Carlo

global X, E_U
function GRADIENT($\tilde{c}, \dot{c}$)
    ... ▷ As in Algorithm 3
function $\nabla U(\mathcal{X})$
    $\tilde{U} \leftarrow \text{EVAL}(E_U[X := \mathcal{X}])$
    return GRAD($\tilde{U}$)
function LEAPFROG($\mathcal{X}_0$, $\mathcal{R}_0$, T, ϵ)
    $\mathcal{R}_{1/2} \leftarrow \mathcal{R}_0 - \frac{1}{2}\epsilon\nabla U(\mathcal{X}_0)$
    for t **in** $1, \ldots, T-1$ **do**
        $\mathcal{X}_t \leftarrow \mathcal{X}_{t-1} + \epsilon\mathcal{R}_{t-1/2}$
        $\mathcal{R}_{t+1/2} \leftarrow \mathcal{R}_{t-1/2} - \epsilon\nabla U(\mathcal{X}_t)$
    $\mathcal{X}_T \leftarrow \mathcal{X}_{T-1} + \epsilon\mathcal{R}_{T-1/2}$
    $\mathcal{R}_T \leftarrow \mathcal{R}_0 - \frac{1}{2}\epsilon\nabla U(\mathcal{X}_{T-1/2})$
    return $\mathcal{X}_T, \mathcal{R}_T$
function HMC($\mathcal{X}^{(0)}$, S, T, ϵ, M)
    for s **in** $1, \ldots, S$ **do**
        $\mathcal{R}^{(s-1)} \sim \text{Normal}(0, M)$
        $\mathcal{X}', \mathcal{R}' \leftarrow \text{LEAPFROG}(\mathcal{X}^{(s-1)}, \mathcal{R}^{(s-1)}, T, \epsilon)$
        $u \sim \text{Uniform}(0, 1)$
        if $u < \exp\big(-H(\mathcal{X}', \mathcal{R}') + H(\mathcal{X}^{(s-1)}, \mathcal{R}^{(s-1)})\big)$ **then**
            $\mathcal{X}^{(s)} \leftarrow \mathcal{X}'$
        else
            $\mathcal{X}^{(s)} \leftarrow \mathcal{X}$
    return $\mathcal{X}^{(1)}, \ldots, \mathcal{X}^{(S)}$

3.4.2 Implementation Considerations

Algorithm 4 shows pseudocode for an HMC algorithm that makes use of automatic differentiation to compute the gradient $\nabla U(X)$. There are a number of implementation considerations to this algorithm that we have thus far not discussed. Some of these considerations are common to all HMC implementations. Algorithm 4 performs numerical integration using a leapfrog scheme, which discretizes the trajectory for the position X to time points at an interval ϵ and computes a correponding trajectory

for the momentum R at time points that are shifted by $\epsilon/2$ relative to those at which we compute the position. There is a trade-off between the choice of step size ϵ and the numerical stability of the integration scheme, which affects the acceptance rate. Moreover, this step size should also appropriately account for the choice of mass matrix M, which is generally chosen to match the covariance in the posterior expectation $M_{ij}^{-1} \simeq \mathbb{E}_{\pi(X)}[x_i x_j] - \mathbb{E}_{\pi(X)}[x_i]\mathbb{E}_{\pi(X)}[x_j]$. Finally, modern implemenations of HMC typically employ a No-Uturn sampling (NUTS) scheme to ensure that the number of time steps T is chosen in a way that minimizes the degree of correlation between samples.

An implementation consideration unique to probabilistic programming is that not all FOPPL programs define densities $\gamma(X) = p(Y, X)$ that are differentiable at *all* points in the space. The same is true for systems like Stan (Stan Development Team, 2014) and PyMC3 (Salvatier et al., 2016), which opt to provide users with a relatively expressive modeling language that includes if expressions, loops, and even recursion. While these systems enforce the requirement that a program defines a density over set of continuous variables that is known at compile time, they do not enforce the requirement that the density is differentiable. For example, the following FOPPL program would be perfectly valid when expressed as a Stan or PyMC3 model

```
(let
  [x (sample (normal 0.0 1.0))
   y 0.5]
  (if (> x 0.0)
    (observe (normal 1.0 0.1) y)
    (observe (normal -1.0 0.1) y)))
```

This program corresponds to an unnormalized density

$$\gamma(x) = \text{Norm}(0.5; 1, 0)^{I[x>0]}\text{Norm}(0.5; -1, 0)^{I[x\leq 0]}\text{Norm}(x; 0, 1),$$

for which the derivative is clearly undefined at $x = 0$, since $\gamma(x)$ is discontinuous contains at this point. This means that HMC will not sample from the correct distribution if we were to naively compute the derivatives at $x \neq 0$. Even in cases where the density is continuous, the derivative may not be defined at every point.

In other words, it is easy to define a program that may not satisfy the requirements necessary for HMC. So what are these requirements? Precisely characterizing them is complex although early attempts are being made (Gram-Hansen et al., 2018). In practice, to be safe, a program should not contain if expressions that cannot be partially evaluated, all primitive functions must be differentiable everywhere, and cannot contain unobserved discrete random variables.

3.5 Compilation to a Factor Graph

In Section 3.2, we showed that a Bayesian network is a representation of a joint probability $p(Y, X)$ of observed random variables Y, each of which corresponds to an `observe` expression, and unobserved random variables X, each of which corresponds to a `sample` expression. Given this representation, we can now reason about a posterior probability $p(X \mid Y)$ of the sampled values, conditioned on the observed values. In Section 3.2.1, we showed that we can generalize this representation to an unnormalized density $\gamma(X) = \psi(X)p(X)$ consisting of a directed network that defines a prior probability $p(X)$ and a potential term (or factor) $\psi(X)$. In this section, we will represent a probabistic program in the FOPPL as a factor graph, which is a fully undirected network. We will use this representation in Section 3.6 to define an expectation propagation algorithm.

A factor graph defines an unnormalized density on a set of variables X in terms of a product over an index set F

$$\gamma(X) := \prod_{f \in F} \psi_f(X_f), \tag{3.48}$$

in which each function ψ_f, which we refer to as a factor, is itself an unnormalized density over some subset of variables $X_f \subseteq X$. We can think of this model as a bipartite graph with variable nodes X, factor nodes F and a set of undirected edges $A \subseteq X \times F$ that indicate which variables are associated with each factor

$$X_f := \{x : (x, f) \in A\}. \tag{3.49}$$

Any directed graphical model $(V, A, \mathcal{P}, \mathcal{Y})$ can be interpreted as a factor graph in which there is one factor $f \in F$ for each variable $v \in V$. In

$$\frac{}{\rho, c \Downarrow_f c} \qquad \frac{}{\rho, v \Downarrow_f v} \qquad \frac{e_1 \Downarrow_f e_1' \qquad e_2 \Downarrow_f e_2'}{\rho, (\texttt{let}\ [v\ e_1]\ e_2) \Downarrow_f (\texttt{let}\ [v\ e_1']\ e_2')}$$

$$\frac{e \Downarrow_f e'}{\rho, (\texttt{sample}\ e) \Downarrow_f (\texttt{sample}\ e')} \qquad \frac{e_1 \Downarrow_f e_1' \qquad e_2 \Downarrow_f e_2'}{\rho, (\texttt{observe}\ e_1\ e_2) \Downarrow_f (\texttt{observe}\ e_1'\ e_2')}$$

$$\frac{\begin{array}{ll}\rho, e_i \Downarrow_f e_i' \text{ for } i = 0, \ldots, n & \rho(f) = (\texttt{defn}\ [v_1 \ldots v_n]\ e_0) \\ \rho, e_0 \Downarrow_f e_0' & \rho(f') = (\texttt{defn}\ [v_1 \ldots v_n]\ e_0')\end{array}}{\rho, (f\ e_1\ \ldots\ e_n) \Downarrow_f (f'\ e_1'\ \ldots\ e_n')}$$

$$\frac{\rho, e_i \Downarrow_f e_i' \text{ for } i = 1, \ldots, n \qquad op = \texttt{if} \quad \text{or} \quad op = c}{\rho, (op\ e_1\ \ldots\ e_n) \Downarrow_f (\texttt{sample}\ (\texttt{dirac}\ (op\ e_1'\ \ldots\ e_n')))}$$

Figure 3.2: Inference rules for the transformation $\rho, e \Downarrow_f e'$, which replaces if forms and primitive procedure calls with expressions of the form `(sample (dirac e))`.

other words we could define

$$\gamma(X) := \prod_{v \in V} \psi_v(X_v), \tag{3.50}$$

where the factors $\psi_v(X_v)$ equivalent to the expressions $\mathcal{P}(v)$ that evaluate the probability density for each variable v, which can be either observed or unobserved,

$$\psi_v(X_v) \equiv \begin{cases} \mathcal{P}(v)[v := \mathcal{Y}(v)], & v \in \text{dom}(\mathcal{Y}), \\ \mathcal{P}(v), & v \notin \text{dom}(\mathcal{Y}). \end{cases} \tag{3.51}$$

A factor graph representation of a density is not unique. For any factorization, we can merge two factors f and g into a new factor h

$$\psi_h(X_h) := \psi_f(X_f)\psi_g(X_g), \qquad X_h := X_f \cup X_g. \tag{3.52}$$

A graph in which we replace the factors f and g with the merged factor h is then an equivalent representation of the density. The implication of this is that there is a choice in the level of granularity at which we wish to represent a factor graph. The representation above has a comparatively low level of granularity. We will here consider a more

fine-grained representation, analogous to the one used in Infer.NET (Minka et al., 2010b). In this representation, we will still have one factor for every variable, but we will augment the set of nodes X to contain an entry x for every deterministic expression in a FOPPL program. We will do this by defining a source code transformation $\rho, e \Downarrow_f e'$ that replaces each deterministic sub-expressions (i.e. if expressions and primitive procedure calls) with expressions of the form

$$\texttt{(sample (dirac } e'\texttt{))}$$

Where `(dirac` e'`)` refers to the Dirac delta distribution with density

$$p_{\mathsf{dirac}}(x\,;\,c) = I[x = c]$$

After this source code transformation, we can use the rules from Section 3.1 to compile the transformed program into a directed graphical model $(V, A, \mathcal{P}, \mathcal{Y})$. This model will be equivalent to the directed graphical model of the untransformed program, but contains an additional node for each Dirac-distributed deterministic variable.

The inference rules for the source code transformation $\rho, e \Downarrow_f e'$ are much simpler than the rules that we wrote down in Section 3.1. We show these rules in Figure 3.2. The first two rules state that constants c and variables v are unaffected. The next rules state that let, sample, and observe forms are transformed by transforming each of the sub-expressions, inserting deterministic variables where needed. User-defined procedure calls are similarly transformed by transforming each of the arguments $e_1, \ldots, e_n$, and transforming the procedure body e_0. So far, none of these rules have done anything other than state that we transform an expression by transforming each of its sub-expressions. The two cases where we insert Dirac-distributed variables are if forms and primitive procedure applications. For these expression forms e, we transform the sub-expressions to obtain a transformed expression e' and then return the wrapped expression `(sample (dirac` e'`))`.

As noted above, a directed graphical model can always be interpreted as a factor graph that contains single factor for each random variable. To aid discussion in the next section, we will explicitly define the mapping from the directed graph $(V^{\mathrm{dg}}, A^{\mathrm{dg}}, \mathcal{P}^{\mathrm{dg}}, \mathcal{Y}^{\mathrm{dg}})$ of a transformed program

onto a factor graph (X, F, A, Ψ) that defines a density of the form in Equation 3.48.

A factor graph (X, F, A, Ψ) is a bipartite graph in which X is the set of variable nodes, F is the set of factor nodes, A is a set of undirected edges between variables and factors, and Ψ is a map of factors that will be described shortly. The set of variables is identical to the set of unobserved variables (i.e. the set of sample forms) in the corresponding directed graph

$$X := X^{\mathrm{dg}} = V^{\mathrm{dg}} \setminus \mathrm{dom}(\mathcal{Y}^{\mathrm{dg}}). \tag{3.53}$$

We have one factor $f \in F$ for every variable $v \in V^{\mathrm{dg}}$, which includes *both* unobserved variables $x \in X^{\mathrm{dg}}$, corresponding to sample expressions, and observed variables $y \in Y^{\mathrm{dg}}$. We write $F \overset{1-1}{=} V^{\mathrm{dg}}$ to signify that there is a bijective relation between these two sets and use $v_f \in V$ to refer to the variable node that corresponds to the factor f. Conversely we use $f_v \in F$ to refer to the factor that corresponds to the variable node v. We can then define the set of edges $A \overset{1-1}{=} A^{\mathrm{dg}}$ as

$$A := \{(v, f) : (v, v_f) \in A^{\mathrm{dg}}\}. \tag{3.54}$$

The map Ψ contains an expression $\Psi(f)$ for each factor, which evaluates the potential function of the factor $\psi_f(X_f)$. We define $\Psi(f)$ in terms of the of the corresponding expression for the conditional density $\mathcal{P}^{\mathrm{dg}}(v_f)$,

$$\Psi(f) := \begin{cases} \mathcal{P}^{\mathrm{dg}}(v_f)[v_f := \mathcal{Y}^{\mathrm{dg}}(v_f)], & v_f \in \mathrm{dom}(\mathcal{Y}^{\mathrm{dg}}), \\ \mathcal{P}^{\mathrm{dg}}(v_f), & v_f \notin \mathrm{dom}(\mathcal{Y}^{\mathrm{dg}}). \end{cases} \tag{3.55}$$

This defines the usual correspondence between $\psi_f(X_f)$ and $\Psi(f)$, where we note that the set of variables X_f associated with each factor f is equal to the set of variables in $\Psi(f)$,

$$\psi_f(X_f) \equiv \Psi(f), \qquad X_f = \textsc{free-vars}(\Psi(f)). \tag{3.56}$$

For purposes of readability, we have omitted one technical detail in this discussion. In Section 3.2.2, we spent considerable time on techniques for partial evaluation, which proved necessary to avoid graphs that contain spurious edges for between variable that are in fact conditionally independent. In the context of factor graphs, we can similarly eliminate

unnecessary factors and variables. Factors that can be eliminated are those in which the expression $\Psi(f)$ either takes the form $(p_{\text{dirac}}\ v\ c)$ or $(p_{\text{dirac}}\ c\ v)$. In such cases we remove the factor f, the node v, and substitute $v := c$ in the expressions of all other potential functions. Similarly, we can eliminate all variables with factors of the form $(p_{\text{dirac}}\ v_1\ v_2)$ by substituting $v_1 := v_2$ everywhere.

To get a sense of how a factor graph differs from a directed graph, let us look at a simple example, inspired by the TrueSkill model (Herbrich et al., 2007). Suppose we consider a match between two players who each have a skill variable s_1 and s_2. We will assume that the player 1 beats player 2 when $(s_1 - s_2) > \epsilon$, which is to say that the skill of player 1 exceeds the skill of player 2 by some margin ϵ. Now suppose that we define a prior over the skill of each player and observe that player 1 beats player 2. Can we reason about the posterior on the skills s_1 and s_2? We can translate this problem to the following FOPPL program

```
(let [s1 (normal 0 1.0)
      s2 (normal 0 1.0)
      delta (- s1 s2)
      epsilon 0.1
      w (> delta epsilon)
      y true]
  (observe (dirac w) y)
  [s1 s2])
```

This program differs from the ones we have considered so far in that we are using a Dirac delta to enforce a *hard* constraint on observations, which means that this program defines an unnormalized density

$$\gamma(s_1, s_2) = \left(p_{\text{norm}}(s_1; 0, 1)\, p_{\text{norm}}(s_2; 0, 1)\right)^{I[(s_1 - s_2) > \epsilon]}. \tag{3.57}$$

This type of hard constraint poses problems for many inference algorithms for directed graphical models. For example, in HMC this introduces a discontinuity in the density function. However, as we will see in the next section, inference methods based on message passing are much better equipped to deal with this form of condition.

When we compile the program above to a factor graph we obtain a

set of variables $X = (s_1, s_2, \delta, w)$ and the map of potentials

$$\Psi = \begin{bmatrix} f_{s_1} \mapsto (p_{\text{norm}}\ s_1\ \texttt{0.0}\ \texttt{1.0}), \\ f_{s_2} \mapsto (p_{\text{norm}}\ s_2\ \texttt{0.0}\ \texttt{1.0}), \\ f_{\delta} \mapsto (p_{\text{dirac}}\ \delta\ (\texttt{-}\ s_1\ s_2)), \\ f_{w} \mapsto (p_{\text{dirac}}\ w\ (\texttt{>}\ \delta\ \texttt{0.1})), \\ f_{y} \mapsto (p_{\text{dirac}}\ \texttt{true}\ w) \end{bmatrix}. \tag{3.58}$$

Note here that the variables s_1 and s_2 would also be present in the directed graph corresponding to the unstransformed program. The deterministic variables δ and w have been added as a result of the transformation in Figure 3.2. Since the factor f_y restricts w to the value `true`, we can eliminate f_y from the set of factors and w from the set of varables. This results in a simplified graph where $X = (s_1, s_2, \delta)$ and the potentials

$$\Psi = \begin{bmatrix} f_{s_1} \mapsto (p_{\text{norm}}\ \texttt{0.0}\ \texttt{1.0}), \\ f_{s_2} \mapsto (p_{\text{norm}}\ \texttt{0.0}\ \texttt{1.0}), \\ f_{\delta} \mapsto (p_{\text{dirac}}\ \delta\ (\texttt{-}\ s_1\ s_2)), \\ f_{w} \mapsto (p_{\text{dirac}}\ \texttt{true}\ (\texttt{>}\ \delta\ \texttt{0.1})) \end{bmatrix}. \tag{3.59}$$

In summary, we have now created an undirected graphical model, in which there is deterministic variable node $x \in X$ for all primitive operations such as $(\texttt{>}\ v_1\ v_2)$ or $(\texttt{-}\ v_1\ v_2)$. In the next section, we will see how this representation helps us when performing inference.

3.6 Expectation Propagation

One of the main advantages in representing a probabilistic program as a factor graph is that we can perform inference with message passing algorithms. As an example of this we will consider expectation propagation (EP), which forms the basis of the runtime of Infer.NET (Minka et al., 2010b), a popular probabilistic programming language.

EP considers an unnormalized density $\gamma(X)$ that is defined in terms of a factor graph (X, F, A, Ψ). As noted in the preceding section, a factor graph defines a density as a product over an index set F

$$\pi(X) := \gamma(X)/Z^{\pi}, \qquad \gamma(X) := \prod_{f \in F} \psi_f(X_f). \tag{3.60}$$

We approximate $\pi(X)$ with a distribution $q(X)$ that is similarly defined as a product over factors

$$q(X) := \frac{1}{Z^q} \prod_{f \in F} \phi_f(X_f). \tag{3.61}$$

The objective in EP is to make $q(X)$ as similar as possible to $\pi(X)$ by minimizing the Kullback-Leibler divergence

$$\operatorname*{argmin}_q D_{\mathrm{KL}}\left(\pi(X) \,||\, q(X)\right) = \operatorname*{argmin}_q \int \pi(X) \log \frac{\pi(X)}{q(X)} dX, \tag{3.62}$$

EP algorithms minimize the KL divergence iteratively by updating one factor ϕ_f at a time

- Define a tilted distribution

$$\pi_f(X) := \gamma_f(X)/Z_f, \qquad \gamma_f(X) := \frac{\psi_f(X_f)}{\phi_f(X_f)} q(X). \tag{3.63}$$

- Update the factor by minimizing the KL divergence

$$\phi_f = \operatorname*{argmin}_{\phi_f} D_{\mathrm{KL}}\left(\pi_f(X) \,||\, q(X)\right). \tag{3.64}$$

In order to ensure that the KL minimization step is tractable, EP methods rely on the properties of exponential family distributions. We will here consider the variant of EP that is implemented in Infer.NET, which assumes a fully-factorized form for each of the factors in $q(X)$

$$\phi_f(X_f) := \prod_{x \in X_f} \phi_{f \to x}(x). \tag{3.65}$$

We refer to the potential $\phi_{f \to x}(x)$ as the message from factor f to the variable x. We assume that messages have an exponential form

$$\phi_{f \to x}(x) = \exp[\lambda_{f \to x}^\top t(x)], \tag{3.66}$$

in which $\lambda_{f \to x}$ is the vector of natural parameters and $t(x)$ is the vector of sufficient statistics of an exponential family distribution. We can then express the marginal $q(x)$ as an exponential family distribution

$$\begin{aligned} q(x) &= \frac{1}{Z_x^q} \prod_{f : x \in V_f} \phi_{f \to x}(x), \\ &= h(x) \exp\left(\lambda_x t(x) - a(\lambda_x)\right), \end{aligned} \tag{3.67}$$

where $a(\lambda_x)$ is the log normalizer of the exponential family and λ_x is the sum over the parameters for individual messages

$$\lambda_x = \sum_{f:x\in X_f} \lambda_{f\to x}. \tag{3.68}$$

Note that we can express the normalizing constant Z^q as a product over per-variable normalizing constants Z_x^q,

$$Z^q := \prod_{x\in X} Z_x^q, \qquad Z_x^q := \int dx \prod_{f:x\in X_f} \phi_{f\to x}(x), \tag{3.69}$$

where we can compute Z_x^q in terms of λ_x using

$$Z_x^q = \exp\left(a(\lambda_x)\right) = \exp\left(a\left(\textstyle\sum_{f\,:\,x\in X_f} \lambda_{f\to x}\right)\right). \tag{3.70}$$

Exponential family distributions have many useful properties. One such property is that expected values of the sufficient statistics $t(x)$ can be computed from the gradient of the log normalizer

$$\nabla_{\lambda_x} a(\lambda_x) = \mathbb{E}_{q(x)}[t(x)]. \tag{3.71}$$

In the context of EP, this property allows us to express KL minimization as a so-called moment-matching condition. To explain what we mean by this, we will expand the KL divergence

$$D_{\mathrm{KL}}\left(\pi_f(X) \,||\, q(X)\right) = \log \frac{Z^q}{Z_f} + \mathbb{E}_{\pi_f(X)}\left[\log \frac{\psi_f(X_f)}{\phi_f(X_f)}\right]. \tag{3.72}$$

We now want to minimize this KL divergence with respect the parameters $\lambda_{f\to v}$. When we ignore all terms that do not depend on these parameters, we obtain

$$\begin{aligned}&\nabla_{\lambda_{f\to x}} D_{\mathrm{KL}}\left(\pi_f(X) \,||\, q(X)\right) = \\ &\qquad \nabla_{\lambda_{f\to x}}\left(\log Z_x^q - \mathbb{E}_{\pi_f(X)}[\log \phi_{f\to x}(x)]\right) = 0.\end{aligned}$$

When we substitute the message $\phi_{f\to x}(x)$ from Equation 3.66, the normalizing constant $Z_x^q(\lambda_x)$ from Equation 3.70, and apply Equation 3.71, then we obtain the moment matching condition

$$\begin{aligned}\mathbb{E}_{q(x)}[t(x)] &= \nabla_{\lambda_{f\to x}} \mathbb{E}_{\pi_f(X)}[\log \phi_{f\to x}(x)], \\ &= \nabla_{\lambda_{f\to x}} \mathbb{E}_{\pi_f(X)}[\lambda_{f\to x}^\top t(x)], \\ &= \mathbb{E}_{\pi_f(X)}[t(x)].\end{aligned} \tag{3.73}$$

Algorithm 5 Fully-factorized Expectation Propagation

function PROJ(G, λ, f)
  $X, F, A, \Psi \leftarrow G$
  $\gamma_f(X) \leftarrow \psi_f(X) q(X) / \phi_f(X)$ ▷ Equation (3.63)
  $Z_f \leftarrow \int dX\, \gamma_f(X)$ ▷ Equation (3.75)
  for x in X_f **do**
    $\bar{t} \leftarrow 1/Z_f \int dX\, \gamma_f(X) t(x)$ ▷ Equation (3.77)
    $\lambda^*_x \leftarrow$ MOMENT-MATCH($\bar{t}$) ▷ Equation (3.73)
    $\lambda_{f\to x} \leftarrow \lambda^*_x - \sum_{f' \neq f\,:\, x \in X_{f'}} \lambda_{f'\to x}$ ▷ Equation (3.74)
  return $\lambda, \log Z_f$
function EP(G)
  $X, F, A, \Psi \leftarrow G$
  $\lambda \leftarrow$ INITIALIZE-PARAMETERS(G)
  for f in SCHEDULE(G) **do**
    $\lambda, \log Z_f \leftarrow$ PROJ(G, λ, f)
  for x in X **do**
    $\log Z^q_x \leftarrow a(\lambda_x)$ ▷ Equation (3.70)
  $\log Z^\pi \leftarrow \sum_f \log Z_f + \sum_x \log Z^q_x$ ▷ Equation (3.78)
  return $\lambda, \log Z^\pi$

If we assume that the parameters λ^*_x satisfy the condition above, then we can use Equation 3.68 to define the update for the message $\phi_{f\to x}$

$$\lambda_{f\to x} \leftarrow \lambda^*_x - \sum_{f' \neq f\,:\, x \in X_{f'}} \lambda_{f'\to x}. \tag{3.74}$$

In order to implement the moment matching step, we have to solve two integrals. The first computes the normalizing constant Z_f. We can express this integral, which is nominally an integral over all variables X, as an integral over the variables X_f associated with the factor f,

$$\begin{aligned} Z_f = \int dX \frac{\psi_f(X_f)}{\phi_f(X_f)} q(X) &= \int dX_f\, \frac{\psi_f(X_f)}{\phi_f(X_f)} \prod_{x \in X_f} \frac{1}{Z^q_x} \prod_{f'\,:\, x \in V_{f'}} \phi_{f'\to x}(x), \\ &= \int dX_f\, \psi_f(X_f) \prod_{x \in X_f} \frac{1}{Z^q_x} \phi_{x\to f}(x). \end{aligned} \tag{3.75}$$

Here, the function $\phi_{x\to f}(x)$ is known as the message from the variable v to the factor f, which is defined as

$$\phi_{x\to f}(x) := \prod_{x\in X_f} \prod_{f'\neq f\,:\,x\in X_{f'}} \phi_{f'\to x}(x). \tag{3.76}$$

These messages can also be used to compute the second set of integrals for the sufficient statistics

$$\bar{t} = \mathbb{E}_{\pi_f(V)}[t(v)] = \frac{1}{Z_f}\int dV_f\; t(x)\psi_f(X_f) \prod_{x\in X_f} \frac{1}{Z_x^q}\phi_{x\to f}(x). \tag{3.77}$$

Algorithm 5 summarizes these computations. We begin by initializing parameter values for each of the messages. We then pick factors f to update according to some schedule. For each update we then compute Z_f. For each each $x \in X_f$ we then compute $\bar{t}$, find the the parameters λ_x^* that satisfy the moment-matching condition and then use these parameters to update parameters $\lambda_{f\to x}$. Finally, we note that EP obtains an approximation to the normalizing constant Z^π for the full unnormalized distribution $\pi(X) = \gamma(X)/Z^\pi$. This approximation can be computed from the normalizing constants of the tilted distributions Z_f and the normalizing constants Z_x^q,

$$Z^\pi \simeq \prod_{f\in F} Z_f \prod_{x\in X} Z_x^q. \tag{3.78}$$

3.6.1 Implementation Considerations

There are a number of important considerations when using EP for probabilistic programming in practice. The type of schedule implemented by the function SCHEDULE(G) is perhaps the most important design consideration. In general, EP updates are not guaranteed to converge to a fixed point, and choosing a schedule that is close to optimal is an open problem. In fact, a large proportion of the development effort for Infer.NET (Minka et al., 2010b) has focused on identifying heuristic for choosing this schedule.

As with HMC, there are also restrictions to the types of programs that are amenable to inference with EP. To perform EP, a FOPPL program needs to satisfy the following requirements

1. We need to be able to associate an exponential family distribution with each variable x in the program.

2. For every factor f, we need to be able to compute the integral for Z_f in Equation (3.75).

3. For every message $\phi_{f \to x}(x)$, we need to be able to compute the sufficient statistics $\bar{t}$ in Equation (3.77).

The first requirement is relatively easy to satisfy. The exponential family includes the Gaussian, Gamma, Discrete, Poisson, and Dirichlet distributions, which covers the cases of real-valued, positive-definite, discrete with finite cardinality and discrete with infinite cardinality.

The second and third requirement impose more substantial restrictions on the program. To get a clearer sense of these requirements, let us return to the example that we looked at in Section 3.5

```
(let [s1 (normal 0 1.0)
      s2 (normal 0 1.0)
      delta (- s1 s2)
      epsilon 0.1
      w (> delta epsilon)
      y true]
  (observe (dirac w) y)
  [s1 s2])
```

After elimination of unnecessary factors and variables, this program defines a model with variables $X = (s_1, s_2, \delta)$ and potentials

$$\Psi = \begin{bmatrix} f_1 \mapsto (p_{\text{norm}}\ \texttt{0.0}\ \texttt{1.0}), \\ f_2 \mapsto (p_{\text{norm}}\ \texttt{0.0}\ \texttt{1.0}), \\ f_3 \mapsto (p_{\text{dirac}}\ \delta\ (\texttt{-}\ s_1\ s_2)), \\ f_4 \mapsto (p_{\text{dirac}}\ \texttt{true}\ (\texttt{>}\ \delta\ \texttt{0.1})) \end{bmatrix}. \tag{3.79}$$

In fully-factorized EP, we assume an exponential family form for each of the variables s_1, s_2 and d_{12}. The obvious choice here is to approximate each variable with an unnormalized Gaussian, for which the sufficient statistics are $t(x) = (x^2, x)$. The Gaussian marginals $q(s_1)$, $q(s_2)$ and $q(d_{12})$ will then approximate the the corresponding marginals $\pi(s_1)$, $\pi(s_2)$, and $\pi(d_{12})$ of the target density.

Let us now consider what operations we need to implement to compute the integrals in Equation (3.75) and Equation (3.77). We will start with the case of the integral for Z_f when updating the factor f_3,

$$Z_f = \frac{1}{Z^q_{s_1} Z^q_{s_2} Z^q_\delta} \int ds_1\, ds_2\, d\delta\; I[\delta = s_1 - s_2] \\ \phi_{s_1 \to f_3}(s_1)\phi_{s_2 \to f_3}(s_2)\phi_{\delta \to f_3}(\delta). \tag{3.80}$$

We can the substitute $\delta := s_1 - s_2$ to eliminate δ, which yields an integral over s_1 and s_2

$$Z_f = \frac{1}{Z^q_{s_1} Z^q_{s_2} Z^q_\delta} \int ds_1\, ds_2\; \phi_{s_1 \to f_3}(s_1)\phi_{s_2 \to f_3}(s_2)\phi_{\delta \to f_3}(s_1 - s_2).$$

Each of the messages is an unnormalized Gaussian, so this is an integral over a product of 3 Gaussians, which we can compute in closed form.

Now let us consider the case of the update for factor f_4. For this factor the integral for Z_f takes the form

$$\begin{aligned} Z_f &= \frac{1}{Z^q_\delta} \int_{-\infty}^{\infty} d\delta \;\; I[\delta > 0.1]\; \phi_{\delta \to f_4}(\delta), \\ &= \frac{1}{Z^q_\delta} \int_{0.1}^{\infty} d\delta\; \phi_{\delta \to f_4}(\delta). \end{aligned} \tag{3.81}$$

This is just an integral over a truncated Gaussian, which is also something that we can approximate numerically.

We now also see why it is advantageous to introduce a factor for each primitive operation. In the case above, if we were to combine the factors f_3 and f_4 into a single factor, then we would obtain the integral

$$Z_f = \frac{1}{Z^q_{s_1} Z^q_{s_2}} \int ds_1\, ds_2 I[s_1 - s_2 > 0.1] \\ \phi_{s_1 \to f_{3+4}}(s_1)\phi_{s_2 \to f_{3+4}}(s_2). \tag{3.82}$$

Integrals involving constraints over multiple deterministic operations will be much harder to compute in an automated manner than integrals involving constraints over atomic operations. Representing each deterministic operation as a separate factors avoids this problem.

To provide a full implementation of EP for the FOPPL, we need to be able to solve the integral for Z_f in Equation (3.75) and the integrals

for the sufficient statistics in Equation (3.77) for each potential type. This requirement imposes certain constraints on the programs we can write. The cases that we have to consider are stochastic factors (sample and observe expressions) and deterministic factors (if expressions and primitive procedure calls).

For sample and observe expressions, potentials have the form $\Psi(f) = (p\ v_0\ v_1\ \ldots\ v_n)$ and $\Psi(f) = (p\ c_0\ v_1\ \ldots\ v_n)$ respectively. For these potentials, we have to integrate over products of densities, which can in general be done only for a limited number of cases, such as conjugate prior and likelihood pairs. This means that the exponential family that is chosen for the messages needs to be compatible with the densities in sample and observe expressions.

Deterministic factors take the form $(p_{\mathsf{dirac}}\ v_0\ E)$ where E is an expression in which all sub-expressions are variable references,

$$E ::= (\texttt{if}\ v_1\ v_2\ v_3) \mid (c\ v_1 \ldots v_n)$$

For if expressions $(\texttt{if}\ v_1\ v_2\ v_3)$, it is advantageous to employ contructs known as gates (Minka and Winn, 2009), which treat the if block as a mixture over two distributions and propagate messages by computing expected values of over the indicator variable accordingly.

In the case of primitive procedure calls, we need to provide implementations of the integrals that only depend on the primitive c, but also on the type of exponential family that is used for the messages v_1 through v_n. For example, if we consider the expression $(-\ v_1\ v_2)$, then our implementation for the integrals will be different when v_1 and v_2 are both Gaussian, both Gamma-distributed, or when one variable is Gaussian distributed and the other is Gamma-distributed.

4

Evaluation-Based Inference I

In the previous chapter, our inference algorithms operated on a graph representation of a probabilistic model, which we created through a compilation of a program in our first-order probabilistic programming language. Like any compilation step, the construction of this graph is performed ahead of time, prior to running inference. We refer to graphs that can be constructed at compile time as having static support.

There are many models in which the graph of conditional dependencies is dynamic, in the sense that it cannot be constructed prior to performing inference. One way that such graphs arise is when the number of random variables is itself not known at compile time. For example, in a model that performs object tracking, we may not know how many objects will appear, or for how long they will be in the field of view. We will refer to these types of models as having dynamic support.

There are two basic strategies that we can employ to represent models with dynamic support. One strategy is to introduce an upper bound on the number of random variables. For example, we can specify a maximum number of objects that can be tracked at any one time. When employing this type of modeling strategy, we additionally need to specify which variables are needed at any one time. For example, if

we had random variables corresponding to the position of each possible object, then we would have to introduce auxiliary variables to indicate which objects are in view. This process of "switching" random variables "on" and "off" allows us to approximate what is fundamentally a dynamic problem with a static one.

The second is strategy is to implement inference methods that dynamically instantiate random variables. For example, at each time step an inference algorithm could decide whether there are any new objects have appeared in the field of view, and then create random variables for the position of these objects as needed. A particular strategy for dynamic instantiation of variables is to generate values for variables by simply running a program. We refer to such strategies as evaluation-based inference methods.

Evaluation-based methods differ from their compilation-based counterparts in that they do not require a representation of the dependency graph to be known prior to execution. Rather, the graph is either built up dynamically at run time, or never explicitly constructed at all. This means that many evaluation-based strategies can be applied to models that can in principle instantiate an unbounded number of random variables.

One of the main things we will change in evaluation-based methods is how we deal with if-expressions. In the previous chapter we realized that if-expressions require special consideration in probabilistic programs. The question that we identified was whether lazy or eager evaluation should be used in if expressions that contain sample and/or observe expressions. We showed that lazy evaluation is necessary for observe expressions, since these expressions affect the posterior distribution on the program output. However, for sample expressions, we have a choice between evaluation strategies, since we can always treat variables in unused branches as auxiliary variables. Because lazy evaluation makes it difficult to characterize the support, we adopted an eager evaluation strategy, in which both branches of each if expression are evaluated, but a symbolic flow control predicate determines when observe expressions need to be incorporated into the likelihood.

In practice, this eager evaluation strategy for if expressions has its limitations. The language that we introduced in chapter 2 was carefully

designed to ensure that programs always evaluate a bounded set of sample and observe expressions. Because of this, programs that are written in the FOPPL can be safely eagerly evaluated. It is very easy to create a language in which this is no longer the case. For example, if we simply allow function definitions to be recursive, then we can now write programs such as this one

```
(defn sample-geometric [alpha]
  (if (= (sample (bernoulli alpha)) 1)
    1
    (+ 1 (sample-geometric p))))

(let [alpha (sample (uniform 0 1))
      k (sample-geometric alpha)]
  (observe (poisson  k) 15)
  alpha)
```

In this program, the recursive function `sample-geometric` defines the functional programming equivalent of a while loop. At each iteration, the function samples from a Bernoulli distribution, returning 1 when the sampled value is 1 and recursively calling itself when the value is 0. Eager evaluation of if expressions would result in an infinite recursion for this program, so the compilation strategy that we developed in the previous chapter would clearly fail here. This makes sense, since the expression `(sample (bernoulli p))` can in principle be evaluated an unbounded number of times, implying that the number of random variables in the graph is unbounded as well.

Even though we can no longer compile the program above to a static graph, it turns out that we can still perform inference in order to characterize the posterior on the program output. To do so, we rely on the fact that we can always simply run a program (using lazy evaluation for if expressions) to generate a sample from the prior. In other words, even though we might not be able to characterize the support of a probabilistic program, we can still generate a sample that, by construction, is guaranteed to be part of the support. If we additionally keep track of the probabilities associated with each of the observe expressions that is evaluated in a program, then we can implement sampling algorithms that either evaluate an Metropolis-

Hastings acceptance ratio, or assuming an importance weight to each sample.

While many evaluation-based methods in principle apply to models with unbounded numbers of variables, there are in practice some subtleties that arise when reasoning about such inference methods. In this chaper, we will therefore assume that programs are defined using the first order language form Chapter 2, but that a lazy evaluation strategy is used for if expressions. Evaluation-based methods for these programs are still easier to reason about, since we know that there is some finite set of sample and observe expressions that can be evaluated. In the next chapter, we will discuss implementation issues that arise when probabilistic programs can have unbounded numbers of variables.

4.1 Likelihood Weighting

Arguably the simplest evaluation-based method is likelihood weighting, which is a form of importance sampling in which the proposal is the prior. In order to see how importance sampling methods can be implemented using evaluation-based strategies, we will first discuss what operations need to be performed in importance sampling. We then briefly discuss how we could implement likelihood weighting for a program that has been compiled to a graphical model. We will then move on to discussing how we can implement importance sampling by repeatedly running the program.

4.1.1 Background: Importance Sampling

Like any Monte Carlo technique, importance sampling methods approximates the posterior distribution $p(X|Y)$ with a set of (weighted) samples. The trick that importance sampling methods rely upon is that we can replace an expectation over $p(X|Y)$, which is generally hard to sample from, with an expectation over a proposal distribution $q(X)$,

which is chosen to be easy to sample from

$$\begin{aligned}\mathbb{E}_{p(X|Y)}[r(X)] &= \int dX\, p(X|Y) r(X), \\ &= \int dX\, q(X) \frac{p(X|Y)}{q(X)} r(X) = \mathbb{E}_{q(X)}\left[\frac{p(X|Y)}{q(X)} r(X)\right].\end{aligned}$$

The above equality holds as long as $p(X|Y)$ is absolutely continuous with respect to $q(X)$, which informally means that if according to $p(X|Y)$, the random variable X has a non-zero probability of being in some set A, then $q(X)$ assigns a non-zero probability to X being in the same set. If we draw samples $X^l \sim q(X)$ and define importance weights $w^l := p(X^l|Y)/q(X^l)$ then we can express our Monte Carlo estimate as an average over weighted samples $\{(w^l, X^l)\}_{l=1}^L$,

$$\mathbb{E}_{q(X)}\left[\frac{p(X|Y)}{q(X)} r(X)\right] \simeq \frac{1}{L}\sum_{l=1}^{L} w^l r(X^l).$$

Unfortunately, we cannot calculate the importance ratio $p(X|Y)/q(X)$. This requires evaluating the posterior $p(X|Y)$, which is what we did not know how to do in the first place. However, we are able to evaluate the joint $p(Y, X)$, which allows us to define an unnormalized weight,

$$W^l := \frac{p(Y, X^l)}{q(X^l)} = p(Y)\, w^l. \tag{4.1}$$

If we substitute $p(X|Y) = p(Y, X)/p(Y)$ then we can re-express the expectation over $q(X)$ in terms of the unnormalized weights,

$$\mathbb{E}_{q(X)}\left[\frac{p(X|Y)}{q(X)} r(X)\right] = \frac{1}{p(Y)} \mathbb{E}_{q(X)}\left[\frac{p(Y, X)}{q(X)} r(X)\right], \tag{4.2}$$

$$\simeq \frac{1}{p(Y)} \frac{1}{L} \sum_{l=1}^{L} W^l r(X^l), \tag{4.3}$$

This solves one problem, since the unnormalized weights W^l are quantities that we can calculate directly, unlike the normalized weights w^l. However, we now have a new problem: We also don't know how to calculate the normalization constant $p(Y)$. Thankfully, we can derive an approximation to $p(Y)$ using the same unnormalized weights W^l by

considering the special case $r(X) = 1$,

$$p(Y) = \mathbb{E}_{q(X)}\left[\frac{p(Y,X)}{q(X)}1\right] \simeq \frac{1}{L}\sum_{l=1}^{L} W^l. \tag{4.4}$$

In other words, if we define $\hat{Z} := \frac{1}{L}\sum_{l=1}^{L} W^l$ as the average of the unnormalized weights, then $\hat{Z}$ is an unbiased estimate of the marginal likelihood $p(Y) = \mathbb{E}[\hat{Z}]$. We can now use this estimate to approximate the normalization term in Equation (4.3),

$$\mathbb{E}_{q(X)}\left[\frac{p(X|Y)}{q(X)}r(X)\right] \simeq \frac{1}{p(Y)}\frac{1}{L}\sum_{l=1}^{L} W^l r(X^l), \tag{4.5}$$

$$\simeq \frac{1}{\hat{Z}}\frac{1}{L}\sum_{l=1}^{L} W^l r(X^l) = \sum_{l=1}^{L}\frac{W^l}{\sum_k W^k} r(X^l). \tag{4.6}$$

To summarize, as long as we can evaluate the joint $p(Y, X^l)$ for a sample $X^l \sim q(X)$, then we can perform importance sampling using unnormalized weights W^l. As a bonus, we obtain an estimate $\hat{Z} \simeq p(Y)$ of the marginal likelihood as a by-product of this computation, a number which turns out to be of practical importance for many reasons, not least because it allows for Bayesian model comparison (Rasmussen and Ghahramani, 2001).

We have played a little fast and loose with notation here with the aim of greater readability. Throughout we have focused on the fact that a FOPPL program represents a marginal projection of the posterior distribution, but in the above we temporarily pretended that a FOPPL program represented the full posterior distribution on X. It is entirely correct and acceptable to reread the above with $r(X)$ being the return value projection of X. The most important fact that we have skipped in this entire work up until now is that this posterior marginal will almost always be used in an outer host program to compute an expectation, say of a test function f applied to the posterior distribution of the return value $r(X)$. Note that no matter what the test function is, $\mathbb{E}_{p(X|Y)}[f(r(X))] \approx \sum_{l=1}^{L} w_k f(r(X^l))$ meaning that $\{(w^l, r(X^l))\}_{l=1}^{L}$ is the a weighted sample-based posterior marginal representation that can be used to approximate any expectation.

Likelihood weighting is a special case of importance sampling, in which we use the prior as the proposal distribtion, i.e. $q(X) = p(X)$. The reason this strategy is known as likelihood weighting is that unnormalized weight evaluates to the likelihood when $X^l \sim p(X)$,

$$W^l = \frac{p(Y, X^l)}{q(X^l)} = \frac{p(Y|X^l)p(X^l)}{p(X^l)} = p(Y|X^l). \tag{4.7}$$

4.1.2 Graph-based Implementation

Suppose that we compiled our program to a graphical model as described in Section 3.1. We could then implement likelihood weighting using the following steps:

1. For each $x \in X$: sample from the prior $x^l \sim p(x \mid \text{PA}(x))$.
2. For each $y \in Y$: calculate the weights $W_y^l = p(y \mid \text{PA}(y))$.
3. Return the weighted set of return values $r(X^l)$

$$\sum_{l=1}^{L} \frac{W^l}{\sum_{k=1}^{L} W^k} \delta_{r(X^l), W^l} := \prod_{y \in Y} W_y^l.$$

where δ_x denotes an atomic mass centered on x.

Sampling from the prior for each $x \in X$ is more or less trivial. The only thing we need to make sure of is that we sample all parents $\text{PA}(x)$ before sampling x, which is to say that we need to loop over nodes $x \in X$ according to their topological order. As described in Section 3.2, the terms W_y^l can be calculated by simply evaluating the target language expression $\mathcal{P}(y)[y := \mathcal{Y}(y)]$, substituting the sampled value x^l for each $x \in \text{PA}(y)$.

4.1.3 Evaluation-based Implementation

So how can we implement this same algorithm using an evaluation-based strategy? The basic idea in this implementation will be that we can generate samples by simply running the program. More precisely, we will sample a value $x \sim d$ whenever we encounter an expression `(sample d)`. By definition, this will generate samples from the prior. We can then

Algorithm 6 Base cases for evaluation of a FOPPL program.

global ρ ▷ Procedure definitions
function EVAL(e, σ, ℓ)
  match e
    case (`sample` d)
      ... ▷ Algorithm-specific
    case (`observe` d y)
      ... ▷ Algorithm-specific
    case c
      return c, σ
    case v
      return $\ell(v)$, σ
    case (`let` [v_1 e_1] e_0)
      $c_1, \sigma \leftarrow$ EVAL(e_1, σ, ℓ)
      return EVAL(e_0, σ $\ell[v_1 \mapsto c_1]$)
    case (`if` e_1 e_2 e_3)
      $e_1', \sigma \leftarrow$ EVAL(e_1, σ, ℓ)
      if e_1' **then**
        return EVAL(e_2, σ, ℓ)
      else
        return EVAL(e_3, σ, ℓ)
    case (e_0 e_1 ... e_n)
      for i **in** $1, \ldots, n$ **do**
        $c_i, \sigma \leftarrow$ EVAL(e_i, σ, ℓ)
      match e_0
        case f
          $(v_1, \ldots, v_n), e_0' \leftarrow \rho(f)$
          return EVAL(e_0', σ, $\ell[v_1 \mapsto c_1, \ldots, v_n \mapsto c_n]$)
        case c
          return $c(c_1, \ldots, c_n)$, σ

Constants c are returned as is. Symbols v return a constant from the local environment ℓ. When evaluating the body e_0 of a `let` form or a procedure f, free variables are bound in ℓ. Evaluation of `if` expressions is lazy. The `sample` and `observe` cases are algorithm-specific.

calculate the likelihood as a side-effect of running the program. To do so, we initalize a state variable σ with a single entry $\log W = 0$, which tracks the log of the unnormalized importance weight. Each time we encounter an expression `(observe` d y`)`, we calculate the log likelihood $\log p_d(y)$ and update the log weight to $\log W \leftarrow \log W + \log p_d(y)$, ensuring that $\log W = \log p(Y|X)$ at the end of the execution.

In order to define this method more formally, let us specify what we mean by "running" the program. In Chapter 2, we defined a program q in the FOPPL as

$$q ::= e \mid \texttt{(defn}\ f\ \texttt{[}x_1 \ldots x_n\texttt{]}\ e\texttt{)}\ q$$

In this definition, a program is a single expression e, which evaluates to a return value r, which is optionally preceded by one or more definitions for procedures that may be used in the program. Our language contained eight expression types

$$\begin{aligned} e ::= {} & c \mid v \mid \texttt{(let [}v\ e_1\texttt{]}\ e_2\texttt{)} \mid \texttt{(if}\ e_1\ e_2\ e_3\texttt{)} \\ & \mid \texttt{(}f\ e_1\ \ldots\ e_n\texttt{)} \mid \texttt{(}c\ e_1\ \ldots\ e_n\texttt{)} \\ & \mid \texttt{(sample}\ e\texttt{)} \mid \texttt{(observe}\ e_1\ e_2\texttt{)} \end{aligned}$$

Here we used c to refer to a constant or primitive operation in the language, v to refer to a program variable, and f to refer to a user-defined procedure.

In order to "run" a FOPPL program, we will define a function that evaluates an expression e to a value c. We can define this function recursively; if we want to evaluate the expression `(+ (* 2 3) (* 4 5))` then we would first recursively evaluate the sub-expressions `(* 2 3)` and `(* 4 5)`. We then obtain values `6` and `20` that can be used to perform the function call `(+ 6 20)`. As long as our evaluation function knows how to recursively evaluate each of the eight expression forms above, then we can use this function to evaluate any program written in the FOPPL.

Algorithm 6 shows pseudo-code for a function $\textsc{eval}(e, \sigma, \ell)$ that implements evaluation of each of the non-probabilistic expression forms in the FOPPL (that is, all forms except sample and observe). The arguments to this function are an expression e, a mapping of inference state variables σ and a mapping of local variables ℓ, which we refer to as the local environment. The map σ allows us to store variables needed

for inference, which are computed as a side-effect of the computation. The map ℓ holds the local variables that are bound in let forms and procedure calls. As in Section 3.1, we also assume a mapping ρ, which we refer to as the global environment. For each procedure f the global environment holds a pair $\rho(f) = ([v_1, \ldots, v_n], e_0)$ consisting of the argument variables and the body of the procedure.

In the function $\textsc{eval}(e, \sigma, \ell)$, we use the **match** statement to pattern-match (Wikipedia contributors, 2018) the expression e against each of the 6 non-probabilistic expression forms. These forms are then evaluated as follows:

- Constant values c are returned as is.
- For program variables v, the evaluator returns the value $\ell(v)$ that is stored in the local environment.
- For let forms `(let [`v_1 e_1`]` e_0`)`, we first evaluate e_1 to obtain a value c_1. We then evaluate the body e_0 relative to the extended environment $\ell[v_1 \mapsto c_1]$. This ensures that every reference to v_1 in e_0 will evaluate to c_1.
- For if forms `(if` e_1 e_2 e_3`)`, we first evaluate the predicate e_1 to a value c_1. If c_1 is logically true, then we evaluate the expression for the consequent branch e_2; otherwise we evaluate the alternative branch e_3. Since we only evaluate one of the two branches, this implements a *lazy evaluation strategy* for if expressions.
- For procedure calls `(`f e_1 ... e_n`)`, we first evaluate each of the arguments to values $c_1, \ldots, c_n$. We then retrieve the argument list $[v_1, \ldots, v_n]$ and the procedure body e_0 from the global environment ρ. As with let forms, we then evaluate the body e_0 relative to an extended environment $\ell[v_1 \mapsto c_1, \ldots, v_n \mapsto c_n]$.
- For primitive calls `(`c_0 e_1 ... e_n`)`, we similarly evaluate each of the arguments to values $c_1, \ldots, c_n$. We assume that the primitive c_0 is a function that can be called in the language that implements EVAL. The value of the expression is therefore simply the value of the function call $c_0(c_1, \ldots, c_n)$.

The pseudo-code in Algorithm 6 is remarkably succinct given that this function can evaluate any non-probabilistic program in our first order language. Of course, we are hiding a little bit of complexity. Each of the cases in matches against a particular expression template. Implementing these matching operations can require a bit of extra code. That said, you can write your own LISP interpreter, inclusive of the parser, in about 100 lines of Python (Norvig, 2010).

Now that we have formalized how to evaluate non-probabilistic expressions, it remains to define evaluation for sample and observe forms. As we described at a high level, these evaluation rules are algorithm-dependent. For likelihood weighting, we want to draw from the prior when evaluating sample expressions and update the importance weight when evaluting observe expressions. In Algorithm 7 we show pseudo-code for an implementation of these operations. We assume a variable $\log W$, that holds the log importance weight.

Sample and observe are now implemented as follows:

- For sample forms `(sample` e`)`, we first evaluate the distribution argument e to obtain a distribution value d. We then call SAMPLE(d) to generative a sample from this distribution. Here SAMPLE is a function in the language that implements the evaluator, which needs to be able to generate samples of each distribution type in the FOPPL (in other words, we can think of SAMPLE as a required method for each type of distribution object).
- For observe forms `(observe` e_1 e_2`)` we first evaluate the argument e_1 to a distribution d_1 and the argument e_2 to a value c_2. We then update a variable $\sigma(\log W)$, which is stored in the inference state, by adding LOG-PROB(d_1, c_2), which is the log likelihood of c_2 under the distribution d_1. Finally we return c_2. The function LOG-PROB similarly needs to be able to compute log probability densities for each distribution type in the FOPPL.

Given a program with procedure definitions ρ and body e, the likelihood weighting algorithm repeatedly evaluates the program, starting from an initial state $\sigma \leftarrow [\log W \rightarrow 0]$. It returns the value r^l and the final log weight $\sigma(\log W^l)$ for each execution.

Algorithm 7 Evaluation-based likelihood weighting

global ρ, e ▷ Program procedures, body
function EVAL(e, σ, ℓ)
  match e
    case (`sample` e)
      $d, \sigma \leftarrow$ EVAL(e, σ, ℓ)
      return SAMPLE(d), σ
    case (`observe` e_1 e_2)
      $d_1, \sigma \leftarrow$ EVAL(e_1, σ, ℓ)
      $c_2, \sigma \leftarrow$ EVAL(e_2, σ, ℓ)
      $\sigma(\log W) \leftarrow \sigma(\log W) +$ LOG-PROB(d_1, c_2)
      return c_2, σ
    ... ▷ Base cases (as in Algorithm 6)
function LIKELIHOOD-WEIGHTING(L)
  $\sigma \leftarrow [\log W \mapsto 0]$
  for l in $1, \ldots, L$ **do** ▷ Initialize state
    $r^l, \sigma^l \leftarrow$ EVAL(e, σ, $[\,]$) ▷ Run program
    $\log W^l \leftarrow \sigma(\log W)$ ▷ Store log weight
  return $((r^1, \log W^1), \ldots, (r^L, \log W^L))$

To summarize, we have now defined an evaluated-based inference algorithm that applies generally to probabilistic programs written in the FOPPL. This algorithm generates a sequence of weighted samples by simply running the program repeatedly. Unlike the algorithms that we defined in the previous chapter, this algorithm does not require any explicit representation of the graph of conditional dependencies between variables. In fact, this implementation of likelihood weighting does not even track how many sample and observe statements a program evaluates. Instead, it draws from the prior as needed and accumulates log probabilities when evaluating observe expressions.

Aside 1: Relationship between Evaluation and Inference Rules

In order to evaluate an expression e, we first evaluate its sub-expressions and then compute the value of the expression from the values of the

sub-expressions. In Section 3.1 we implicitly followed the same pattern when defining inference rules for our translation. For example, the rule for translation of a primitive call was

$$\frac{\rho,\, \phi,\, e_i \Downarrow G_i,\, E_i \text{ for all } 1 \leq i \leq n}{\rho,\, \phi,\, (f\ e_1 \ldots e_n) \Downarrow G_1 \oplus \ldots \oplus G_n,\, (c\ E_1 \ldots E_n)}$$

This rule states that if we were implementing a function TRANSLATE then TRANSLATE(ρ, ϕ, e) should perform the following steps when e is of the form $(f\ e_1\ \ldots\ e_n)$:

1. Recursively call TRANSLATE(ρ, ϕ, e_i) to obtain a pair G_i, E_i for each of the sub-expresions $e_1, \ldots, e_n$.
2. Merge the graphs $G \leftarrow G_1 \oplus \ldots \oplus G_n$
3. Construct an expression $E \leftarrow (c\ E_1 \ldots E_n)$
4. Return the pair G, E

In other words, inference rules do not only formally specify how our translation should behave, but also give us a recipe for how to implement a recursive TRANSLATE operation for each expression type.

This similarity is not an accident. In fact, inference rules are commonly used to specify the big-step semantics of a programming language, which defines the value of each expression in terms of the values of its sub-expressions. We can similarly use inference rules to define our evaluation-based likelihood weighting method. We show these inference rules in Figure 4.1.

Aside 2: Side Effects and Referential Transparency

The implementation in Algorithm 7 highlights a fundamental distinction between sample and observe forms relative to the non-probabilistic expression types in the FOPPL. If we do not include sample and observe in our syntax, then our first order language is not only deterministic, but it is also pure in a functional sense. In a purely functional language, there are no side effects. This means that every expression e will always evaluate to the same value. An implication of this is that any expression

$$\frac{}{\rho, \ell, c \Downarrow c, 0} \quad \frac{\ell(v) = c}{\rho, \ell, v \Downarrow c} \quad \frac{\rho, \ell, e_1 \Downarrow c_1, l_1 \quad \rho, \ell \oplus [v_1 \mapsto c_1], e_0 \Downarrow c_0, l_0}{\rho, \ell, (\texttt{let}\ \ [v_1\ \ e_1]\ \ e_0) \Downarrow c_0, l_0 + l_1}$$

$$\frac{\begin{array}{c}\rho, \ell, e_1 \Downarrow \texttt{true}, l_1 \\ \rho, \ell, e_2 \Downarrow c_2, l_2\end{array}}{\rho, \ell, (\texttt{if}\ \ e_1\ \ e_2\ \ e_3) \Downarrow c_2, l_1 + l_2} \quad \frac{\begin{array}{c}\rho, \ell, e_1 \Downarrow \texttt{false}, l_1 \\ \rho, \ell, e_3 \Downarrow c_3, l_3\end{array}}{\rho, \ell, (\texttt{if}\ \ e_1\ \ e_2\ \ e_3) \Downarrow c_3, l_1 + l_3}$$

$$\frac{\begin{array}{c}\rho(f) = [v_1, \ldots, v_n], e_0 \quad \rho, \ell, e_i \Downarrow c_i, l_i \text{ for } i = 1, \ldots, n \\ \rho, \ell \oplus [v_1 \mapsto c_1, \ldots, v_n \mapsto c_n], e_0 \Downarrow c_0, l_0\end{array}}{\rho, \ell, (f\ \ e_1\ \ \ldots\ \ e_n) \Downarrow c_0, l_0 + l_1 + \ldots + l_n}$$

$$\frac{\rho, \ell, e_i \Downarrow c_i, l_i \text{ for } i = 1, \ldots, n \quad c(c_1, \ldots, c_n) = c_0}{\rho, \ell, (c\ \ e_1\ \ \ldots\ \ e_n) \Downarrow c_0, l_1 + \ldots + l_n}$$

$$\frac{\rho, \ell, e \Downarrow d, l \quad c \sim d}{\rho, \ell, (\texttt{sample}\ \ e) \Downarrow c, l} \quad \frac{\rho, \ell, e_1 \Downarrow d_1, l_1 \quad \rho, \ell, e_2 \Downarrow c_2, l_2 \quad \log p_{d_1}(c_2) = l_0}{\rho, \ell, (\texttt{observe}\ \ e_1\ \ e_2) \Downarrow c_2, l_0 + l_1 + l_2}$$

Figure 4.1: Big-step semantics for likelihood weighting. These rules define an evaluation operation $\rho, \ell, e \Downarrow c, l$, in which ρ and ℓ refers to the global and local environment, refers to the local environment, e is an expression, c is the value of the expression and l is its log likelihood.

in a program can be replaced with its corresponding value without affecting the behavior of the rest of the program. We refer to expressions with this property as referentially transparent, and expressions that lack this property as referentially opaque.

Once we incorporate sample and observe into our language, our language is no longer functionally pure, in the sense that not all expressions are referentially transparent. In our implementation in Algorithm 7, a sample expression does not always evaluate to the same value and is therefore referentially opaque. By extension, any expression containing a sample form as a sub-expression is also opaque. An observe expression `(observe` e_1 e_2`)` always evaluates to the same value as long as e_2 is referentially transparent. However observe expressions have a side effect, which is that they increment the log weight stored in the inference state $\sigma(\log W)$. If we replaced an observe form `(observe` e_1 e_2`)` with

the expression for its observed value e_2, then the program would still produce the same distribution on return values when sampling from the prior, but the log weight $\sigma(\log W)$ would be 0 after every execution.

The distinction between referentially transparent and opaque expressions also implicitly showed up in our compilation procedure in Section 3.1. Here we translated an opaque program into a set of target-language expressions for conditional probabilities, which were referentially transparent. In these target-language expressions, each sub-expression corresponding to sample or observe was replaced with a free variable v. If a translated expression has no free variables, then the original untranslated expression is referentially transparent. In Section 3.2.2, we implicitly exploited this property to replace all target-language expressions without free variables with their values. We also relied on this property in Section 3.1 to ensure that observe forms (`observe` e_1 e_2) always contained a referentially transparent expression for the observed value e_2.

4.2 Metropolis-Hastings

In the previous section, we used evaluation to generate samples from the program prior while calculating the likelihood associated with these samples as a side-effect of the computation. We can use this same strategy to define Markov-chain Monte Carlo (MCMC) algorithms. We already discussed two such algorithms, Gibbs Sampling and Hamiltonian Monte Carlo in Sections 3.3 and 3.4 respectively. Both these methods implicitly relied on the fact that we were able to represent a probabilistic program as a static graphical model. In Gibbs sampling, we explicitly made use of the conditional dependency graph in order to identify the minimal set of variables needed to compute the acceptance ratio. In Hamiltonian Monte Carlo, we relied on being able to calculate the gradient $\nabla_X \log p(X)$, which relies on the fact that there is some well-defined set of unobserved random variables X, corresponding to sample expressions that will be evaluated in every execution.

Metropolis-Hastings (MH) methods, which we also mentioned in Section 3.3 generate a Markov chain of program return values $r(X)^1, \ldots, r(X)^S$ by accepting or rejecting a newly proposed sample according to the

following pseudo-algorithm.

- Initialize the current sample X. Return $X^1 \leftarrow X$.
- For each subsequent sample $s = 2, \ldots, S$
  - Generate a proposal $X' \sim q(X' \mid X)$
  - Calculate the acceptance ratio

$$\alpha = \frac{p(Y', X')q(X \mid X')}{p(Y, X)q(X' \mid X)} \tag{4.8}$$

  - Update the current sample $X \leftarrow X'$ with probability $\max(1, \alpha)$, otherwise keep $X \leftarrow X$. Return $X^s \leftarrow X$.

An evaluation-based implementation of a MH sampler needs to do two things. It needs to be able to run the program to generate a proposal, conditioned on the values $\mathcal{X}$ of sample expressions that were evaluated previously. The second is that we need to be able to compute the acceptance ratio α as a side effect.

Let us begin by considering a simplified version of this algorithm. Suppose that we defined $q(X'|X) = p(X')$. In other words, at each step we generate a sample $X' \sim p(X)$ from the program prior, which is independent of the previous sample X. We already know that we can generate these samples simply by running the program. The acceptance ratio now simplifies to:

$$\alpha = \frac{p(Y', X')q(X \mid X')}{p(Y, X)q(X' \mid X)} = \frac{p(Y' \mid X')p(X')p(X)}{p(Y \mid X)p(X)p(X')} = \frac{p(Y' \mid X')}{p(Y \mid X)} \tag{4.9}$$

In other words, when we propose from the prior, the acceptance ratio is simply the ratio of the likelihoods. Since our likelihood weighting algorithm computes $\sigma(\log W) = \log p(Y \mid X)$ as a side effect, we can re-use the evaluator from Algorithm 7 and simply evaluate the acceptance ratio as W'/W, where $W' = p(Y'|X')$ is the likelihood of the proposal and $W = p(Y|X)$ is the likelihood associated with the previous sample. Pseudo-code for this implementation is shown in Algorithm 8.

4.2.1 Single-Site Proposals

Algorithm 8 is so simple because we have side-stepped the difficult operations in the more general MH algorithm: In order to generate a

Algorithm 8 Evaluation-based Metropolis-Hastings with independent proposals from the prior.

global ρ, e
function EVAL(e, σ, ℓ)
    ... ▷ As in Algorithm 7
function INDEPENDENT-MH(S)
    $\sigma \leftarrow [\log W \mapsto 0]$
    $r \leftarrow \text{EVAL}(e, \sigma, [\,])$
    $\log W \leftarrow \log W$
    for s **in** $1, \ldots, S$ **do**
        $r', \sigma' \leftarrow \text{EVAL}(e, \sigma, [\,])$
        $\log W' \leftarrow \sigma'(\log W)$
        $\alpha \leftarrow W'/W$
        $u \sim \text{UNIFORM-CONTINUOUS}(0, 1)$
        if $u < \alpha$ **then**
            $r, \log W \leftarrow r', \log W'$
        $r^s \leftarrow r$
    return $(r^1, \ldots, r^S)$

proposal, we have to run our program in a manner that generates a sample $X' \sim q(X'|X)$ which is conditioned on the values associated with our previous sample. In order to evaluate the acceptance ratio, we have to calculate the probability of the reverse proposal $q(X|X')$. Both these operations are complicated by the fact that X and X' potentially refer to different subsets of sample expressions in the program. To see what we mean by this, Let us take another look at Example 3.5, which we introduced in Section 3.1

```
(let [z (sample (bernoulli 0.5))
      mu (if (= z 0)
           (sample (normal -1.0 1.0))
           (sample (normal 1.0 1.0)))
      d (normal mu 1.0)
      y 0.5]
  (observe d y)
  z)
```

In Section 3.1, we would compile this model to a Bayesian network with three latent variables $X = \{\mu_0, \mu_1, z\}$ and one observed variable $Y = \{y\}$. In this section, we evaluate if-expressions lazily, which means that we will either sample μ_1 (when $z = 1$) or μ_0 (when $z = 0$), but not both. This introduces a complication: What happens when we update $z = 0$ to $z = 1$ in the proposal? This now implies that X contains a variable μ_0, which is not defined for X'. Conversely, X' needs to instantiate a value for the variable μ_1 which was not defined in X.

In order to define an evaluation-based algorithm for constructing a proposal, we will construct a map $\sigma(\mathcal{X})$, such that $\mathcal{X}(x)$ refers to the value of a variable x. In order to calculate the acceptance ratio, we will similarly construct a map $\sigma(\log \mathcal{P})$. Section 3.1 contained a target-language expression $\log \mathcal{P}(v)$ that evaluates to the density for each variable $v \in X \cup Y$. In our evaluation-based algorithm, we will store the log density

$$\sigma(\log \mathcal{P}(x)) = \text{LOG-PROB}(d, \mathcal{X}(x)). \tag{4.10}$$

for each sample expression `(sample d)`, as well as the log density

$$\sigma(\log \mathcal{P}(y)) = \text{LOG-PROB}(d, c) \tag{4.11}$$

for each observe expression `(observe d c)`.

With this notation in place, let us define the most commonly used evaluation-based proposal for probabilistic programming systems: the single-site Metropolis-Hastings update. In this algorithm we change the value for one variable x_0, keeping the values of other variables fixed whenever possible. To do so, we sample x_0 from the program prior, as well as any variables $x \notin \text{dom}(\mathcal{X})$. For all other variables, we reuse the values $\mathcal{X}(x)$. This strategy can be summarized in the following pseudo-algorithm:

- Pick a variable $x_0 \in \text{dom}(\mathcal{X})$ at random from the current sample.
- Construct a proposal $\mathcal{X}', \mathcal{P}'$ by re-running the program:
  - For expressions `(sample d)` with variable x:
    - If $x = x_0$, or $x \notin \text{dom}(\mathcal{X})$, then sample $\mathcal{X}'(x) \sim d$. Otherwise, reuse the value $\mathcal{X}'(x) \leftarrow \mathcal{X}(x)$.

Algorithm 9 Acceptance ratio for single-site proposals

```
1: function ACCEPT(x_0, X', X, log P', log P)
2:     X'^sampled ← {x_0} ∪ (dom(X') \ dom(X))
3:     X^sampled ← {x_0} ∪ (dom(X) \ dom(X'))
4:     log α ← log |dom(X)| − log |dom(X')|
5:     for v in dom(log P') \ X'^sampled do
6:         log α ← log α + log P'(v)
7:     for v in dom(log P) \ X^sampled do
8:         log α ← log α − log P(v)
9:     return α
```

- Calculate the probability $\mathcal{P}'(x) \leftarrow \text{PROB}(d, \mathcal{X}'(x))$.

- For expressions (`observe` d c) with variable y:
  - Calculate the probability $\mathcal{P}'(y) \leftarrow \text{PROB}(d, c)$

What is convenient about this proposal strategy is that it becomes comparatively easy to evaluate the acceptance ratio α. In order to evaluate this ratio, we will rearrange the terms in Equation (4.8) into a ratio of probabilities for X' and a ratio of probabilities for X:

$$\alpha = \frac{p(Y', X')q(X|X')}{p(Y, X)q(X'|X)} \tag{4.12}$$

$$= \frac{p(Y', X')}{q(X'|X, x_0)} \frac{q(X|X', x_0)}{p(Y, X)} \frac{q(x_0|X')}{q(x_0|X)}. \tag{4.13}$$

Here the ratio $q(x_0|X')/q(x_0|X)$ accounts for the relative probability of selecting the initial site. Since x_0 is chosen at random, this is

$$\frac{q(x_0|X')}{q(x_0|X)} = \frac{|X|}{|X'|}. \tag{4.14}$$

We can now express the ratio $p(Y', X')/q(X'|X, x_0)$ in terms of the probabilities $\mathcal{P}'$. The joint probability is simply the product

$$p(Y', X') = p(Y'|X')p(X') = \prod_{y \in Y'} \mathcal{P}'(y) \prod_{x \in X'} \mathcal{P}'(x), \tag{4.15}$$

where $X' = \text{dom}(\mathcal{X}')$ and $Y' = \text{dom}(\mathcal{P}') \setminus X'$.

To calculate the probability $q(X'|X, x_0)$ we decompose the set of variables $X' = X'^{\text{sampled}} \cup X'^{\text{reused}}$ into the set of sampled variables X'^{sampled} and the set of reused variables X'^{reused}. Based on the rules above, the set of sampled variables is given by

$$X'^{\text{sampled}} = \{x_0\} \cup (\text{dom}(\mathcal{X}') \setminus \text{dom}(\mathcal{X})). \tag{4.16}$$

Since all variables in X'^{sampled} were sampled from the program prior, the proposal probability is

$$q(X'|X, x_0) = \prod_{x \in X'^{\text{sampled}}} \mathcal{P}'(x). \tag{4.17}$$

Since some of the terms in the prior and the proposal cancel, the ratio $p(Y', X')/q(X'|X, x_0)$ simplifies to

$$\frac{p(Y', X')}{q(X'|X, x_0)} = \prod_{y \in Y'} \mathcal{P}'(y) \prod_{x \in X'^{\text{reused}}} \mathcal{P}'(x) \tag{4.18}$$

We can define the ratio $p(Y, X)/q(X|X', x_0)$ for the reverse transition by noting that this transition would require sampling a set of variables X^{sampled} from the prior whilst reusing a set of variables X^{reused}

$$\frac{p(Y, X)}{q(X|X, x_0)} = \prod_{y \in \mathcal{Y}} \mathcal{P}(y) \prod_{x \in X^{\text{reused}}} \mathcal{P}(x). \tag{4.19}$$

Here the set of reused variable X^{reused} for the reverse transition is, by definition, identical that of the forward transition X'^{reused},

$$X'^{\text{reused}} = (\text{dom}(\mathcal{X}') \cap \text{dom}(\mathcal{X})) \setminus \{x_0\} = X^{\text{reused}}. \tag{4.20}$$

Putting all the terms together, the acceptance ratio becomes:

$$\alpha = \frac{|\text{dom}(\mathcal{X})|}{|\text{dom}(\mathcal{X}')|} \frac{\prod_{y \in \mathcal{Y}} \mathcal{P}'(y) \prod_{x \in X'^{\text{reused}}} \mathcal{P}'(x)}{\prod_{y \in \mathcal{Y}} \mathcal{P}(y) \prod_{x \in X^{\text{reused}}} \mathcal{P}(x)}. \tag{4.21}$$

If we look at the terms above, then we see that the acceptance ratio for single-site proposals is a generalization of the acceptance ratio that we obtained for independent proposals. When using independent proposals, we could express the acceptance ratio $\alpha = W'/W$ in terms of the likelihood weights $W' = p(Y', X')/q(X') = p(Y' \,|\, X')$. In the

$$\frac{}{\rho, c \Downarrow_\alpha c} \qquad \frac{}{\rho, v \Downarrow_\alpha v} \qquad \frac{\rho, e_1 \Downarrow_\alpha e_1' \quad \rho, e_0 \Downarrow_\alpha e_0'}{\rho, (\texttt{let}\ [v_1\ e_1]\ e_0) \Downarrow_\alpha (\texttt{let}\ [v_1\ e_1']\ e_0')}$$

$$\frac{\rho, e_i \Downarrow_\alpha e_i' \text{ for } i = 1, \ldots, n \quad op = \texttt{if} \text{ or } op = c}{\rho, (op\ e_1\ \ldots\ e_n) \Downarrow_\alpha (op\ e_1'\ \ldots\ e_n')}$$

$$\frac{\begin{array}{c}\rho, e_i \Downarrow_\alpha e_i' \text{ for } i = 0, \ldots, n \qquad \rho(f) = (\texttt{defn}\ [v_1 \ldots v_n]\ e_0) \\ \rho, (\texttt{let}\ [v_n\ e_n']\ e_0') \Downarrow_\alpha e_n'' \quad \rho, (\texttt{let}\ [v_{i-1}\ e_{i-1}']\ e_i'') \Downarrow_\alpha e_{i-1}'' \text{ for } i = n, \ldots, 2\end{array}}{\rho, (f\ e_1\ \ldots\ e_n) \Downarrow_\alpha e_1''}$$

$$\frac{\rho, e \Downarrow_\alpha e' \quad \text{fresh } v}{\rho, (\texttt{sample}\ e) \Downarrow_\alpha (\texttt{sample}\ v\ e')} \qquad \frac{\rho, e_1 \Downarrow_\alpha e_1' \quad \rho, e_2 \Downarrow_\alpha e_2' \quad \text{fresh } v}{\rho, (\texttt{observe}\ e_1\ e_2) \Downarrow_\alpha (\texttt{observe}\ v\ e_1'\ e_2')}$$

Figure 4.2: Addressing transformation for FOPPL programs.

single-site proposal, we treat retained variables $X'^{\text{reused}} = X^{\text{reused}}$ as if they were observed variables. In other words, we could define

$$W' = \frac{p(Y', X')}{q(X'|X, x_0)}. \tag{4.22}$$

Addressing Transformation

In defining the acceptance ratio in Equation (4.21), we have tacitly assumed that we can associate a variable x or y with each sample or observe expression. This is in itself not such a strange assumption, since we did just that in Section 3.1, where we assigned a unique variable v to every sample and observe expression as part of our compilation of a graphical model. In the context of evaluation-based methods, this type of unique identifier for a sample or observe expression is commonly referred to as an address.

If needed, unique addresses can be constructed dynamically at run time. We will get back to this in Chapter 6, Section 6.2. For programs in the FOPPL, we can create addresses using a source code transformation that is similar to the one we defined in Section 3.1, albeit a much simpler one. In this transformation we replace all expressions of the

Algorithm 10 Evaluator for single-site proposals

```
1:  global ρ
2:  function EVAL(e, σ, ℓ)
3:      match e
4:          case (sample v e)
5:              d, σ ← EVAL(e, σ, ℓ)
6:              if v ∈ dom(σ(C)) \ {σ(x_0)} then
7:                  c, ← σ(C(v))                    ▷ Retain previous value
8:              else
9:                  c ← SAMPLE(d)                   ▷ Sample new value
10:             σ(X(v)) ← c                         ▷ Store value
11:             σ(log P(v)) ← LOG-PROB(d, c)        ▷ Store log density
12:             return c, σ
13:         case (observe v e_1 e_2)
14:             d, σ ← EVAL(e_1, σ, ℓ)
15:             c, σ ← EVAL(e_2, σ, ℓ)
16:             σ(log P(v)) ← LOG-PROB(d, c)        ▷ Store log density
17:             return c, σ
18:         ...                                     ▷ Base cases (as in Algorithm 6)
```

form `(sample` e`)` with expressions of the form `(sample` v e`)` in which v is a newly created variable. Similarly, we replace `(observe` e_1 e_2`)` with `(observe` v e_1 e_2`)`. Figure 4.2 defines this translation $\rho, e \Downarrow_\alpha e'$. As in Section 3.1, this translation accepts a map of function definitions ρ, e and returns a transformed expression e' in which addresses have been inserted into all sample and observe expressions.

Evaluating Proposals

Now that we have incorporated addresses that uniquely identify each sample and observe expression, we are in a position to formally define the pseudo-algorithm for single-site Metropolis Hastings that we oulined in Section 4.2.1.

In Algorithm 10, we define the evaluation rules for sample and observe expressions. We assume that the inference state σ holds a value

Algorithm 11 Single-site Metropolis Hastings

```
1:  global ρ, e
2:  function EVAL(e, σ, ℓ)
3:      ...                                              ▷ As in Algorithm 10
4:  function ACCEPT(x_0, X', X, log P', log P)
5:      ...                                              ▷ As in Algorithm 9
6:  function SINGLE-SITE-MH(S)
7:      σ_0 ← [x_0 ← nil, C ↦, [], X ↦ [], log P ↦ []]
8:      r, σ ← EVAL(e, σ_0, [])
9:      for s in 1, ..., S do
10:         v ~ UNIFORM(dom(σ(X)))
11:         σ' ← σ_0[x_0 ↦ v, C ↦ σ(X)]
12:         r', σ' ← EVAL(e, σ', [])
13:         u ~ UNIFORM-CONTINUOUS(0, 1)
14:         α ← ACCEPT(x_0, σ'(X), σ(X), σ'(log P), σ(log P))
15:         if u < α then
16:             r, σ ← r', σ'
17:         r^s ← r
18:     return (r^1, ..., r^S)
```

$\sigma(x_0)$, which is the site of the proposal, a map $\sigma(\mathcal{X})$ map $\sigma(\log \mathcal{P})$, which holds the log density for each variable, and finally a "cache" $\sigma(\mathcal{C})$ of values that we would like to condition the execution on.

For a sample expression with address v, we reuse the value $\mathcal{X}(v) \leftarrow \mathcal{C}(v)$ when possible, unless we are evaluating the proposal site $v = x_0$. In all other cases, we sample $\mathcal{X}(v)$ from the prior. For both sample and observe expressions we calculate the log probability $\log \mathcal{P}(v)$.

The Metropolis Hastings implementation is shown in Algorithm 11. This algorithm initializes the state σ sample by evaluating the program, storing the values $\sigma(\mathcal{X})$ and log probabilities $\sigma(\log \mathcal{P})$ for the current sample. For each subsequent sample the algorithm then selects the initial site x_0 at random from the domain of the current sample $\sigma(\mathcal{X})$. We then rerun the program accordingly to construct a proposal and either accept or reject according to the ratio defined in Algorithm 9.

4.3 Sequential Monte Carlo

One of the limitations of the likelihood weighting algorithm that we introduced in Section 4.1 is that it is essentially a "guess and check" algorithm; we *guess* by sampling a proposal X^l from the program prior and then *check* whether this is in fact a good proposal by calculating a weight $W^l = p(Y|X^l)$ according to the probabilities of observe expressions in the program. The great thing about this algorithm is that it is both simple and general. Unfortunately it is not necessarily *efficient*. In order to get a high weight sample, we have to generate reasonable values for all random variables X. This means that likelihood weighting will work well in programs with a small number of sample expressions, where we can expect to "get lucky" for all sample expressions with reasonable frequency. However, the frequency with which we generate good proposals decreases exponentially with the number of sample expressions in the program.

Sequential Monte Carlo (SMC) methods solve this problem by turning a sampling problem for a high dimensional distribution into a sequence of sampling problems for lower dimensional distributions. In their most general form, SMC methods consider a sequence of unnormalized densities $\gamma_1(X_1), \ldots, \gamma_N(X_N)$, where each $\gamma_n(X_n)$ has the form that we discussed in Section 3.2.1. Here $\gamma_1(X_1)$ is typically a low dimensional distribution, for which it is easy to perform importance sampling, whereas $\gamma_N(X_N)$ is a high dimensional distribution, for which want to generate samples. For each $\gamma_n(X_n)$ in between increases in dimensionality to interpolate between these two distributions. For a FOPPL program, we can define $\gamma_N(X_N) = \gamma(X) = p(Y, X)$ as the joint density associated with the program.

Given a set of unnormalized densities $\gamma_n(X_n)$, SMC sequentially generates weighted samples $\{(X_n^l, W_n^l)\}_{l=1}^L$ by performing importance sampling for each of the normalized densities $\pi_n(X_n) = \gamma_n(X_n)/Z_n$ according to the following rules

- Initialize a weighted set $\{(X_1^l, W_1^l)\}_{l=1}^L$ using importance sampling

$$X_1^l \sim q_1(X_1), \qquad W_1^l := \frac{\gamma_1(X_1^l)}{q_1(X_1^l)}. \tag{4.23}$$

- For each subsequent generation $n = 2, \ldots, N$:

  1. Select a value X_{n-1}^k from the preceding set by sampling an ancestor index $a_{n-1}^l = k$ with probability proportional to W_{n-1}^k

$$a_{n-1}^l \sim \text{Discrete}\left(\frac{W_{n-1}^1}{\sum_l W_{n-1}^l}, \ldots, \frac{W_{n-1}^L}{\sum_l W_{n-1}^l},\right), \tag{4.24}$$

  2. Generate a proposal conditioned on the selected particle

$$X_n^l \sim q_n(X_n \mid X_{n-1}^{a_{n-1}^l}), \tag{4.25}$$

  and define the importance weights

$$W_n^l := W_{n\backslash n-1}^l \hat{Z}_{n-1} \tag{4.26}$$

  where $W_{n\backslash n-1}^l$ is the incremental weight

$$W_{n\backslash n-1}^l := \frac{\gamma_n(X_n^l)}{\gamma_{n-1}(X_{n-1}^{a_{n-1}^l}) q_n(X_n^l \mid X_{n-1}^{a_{n-1}^l})}, \tag{4.27}$$

  and $\hat{Z}_{n-1}$ is defined as the average weight

$$\hat{Z}_{n-1} = \frac{1}{L} \sum_{l=1}^{L} W_{n-1}^l. \tag{4.28}$$

The defining operation in this algorithm is in Equation (4.24), which is known as the resampling step. We can think of this operation as performing "natural selection" on the sample set; samples X_{n-1}^k with a high weight W_{n-1}^k will be used more often to construct proposals equation in (4.25), whereas samples with a low weight will with high probability not be used at all. In other words, SMC uses the weight of a sample at generation $n-1$ as a heuristic for the weight that it will have at generation n, which is a good strategy whenever weights in subsequent densities are strongly correlated.

4.3.1 Defining Intermediate Densities with Breakpoints

As we discussed in Section 3.2.1, a FOPPL program defines an unnormalized distribution $\gamma(X) = p(Y, X)$. When inference is performed with

SMC we define the final density as $\gamma_N(X_N) = \gamma(X)$. In order to define intermediate densities $\gamma_n(X_n) = p(Y_n, X_n)$ we consider a sequence of *truncated* programs that evaluate successively larger subsets of the sample and observe expressions

$$X_1 \subseteq X_2 \subseteq \ldots \subseteq X_N = X, \tag{4.29}$$

$$Y_1 \subseteq Y_2 \subseteq \ldots \subseteq Y_N = Y. \tag{4.30}$$

The definition of a *truncated* program that we employ here is programs that halt at a breakpoint. Breakpoints can be specified explicitly by the user, constructed using program analysis, or even dynamically defined at run time. The sequence of breakpoints needs to satisfy the following two properties in order.

1. The breakpoint for generation n must always occur after the breakpoint for generation $n-1$.

2. Each breakpoint needs to occur at an expression that is evaluated in every execution of a program. In particular, this means that breakpoints should not be associated with expressions inside branches of if expressions.

In this section we will assume that we first apply the addressing transformation from Section 4.2.1 to a FOPPL program. We then assume that the user identifies a sequence of symbols $y_1, \ldots, y_{N-1}$ for observe expressions that satisfy the two properties above. An alternative design, which is often used in practice, is to simply break at every observe and assert that each sample has halted at the same point at run time.

4.3.2 Calculating the Importance Weight

Now that we have defined a notion of intermediate densities $\gamma_n(X_n)$ for FOPPL programs, we need to specify a mechanism for generating proposals from a distribution $q_n(X_n|X_{n-1})$. The SMC analogue of likelihood weighting is to simply sample from the program prior $p(X_n|X_{n-1})$, which is sometimes known as a bootstrapped proposal. For this proposal,

we can express $\gamma_n(X_n)$ in terms of $\gamma_{n-1}(X_{n-1})$ as

$$\begin{aligned}\gamma_n(X_n) &= p(Y_n, X_n) \\ &= p(Y_n|Y_{n-1}, X_n)p(X_n|X_{n-1})p(Y_{n-1}, X_{n-1}) \\ &= p(Y_n|Y_{n-1}, X_n)p(X_n|X_{n-1})\gamma_{n-1}(X_{n-1}).\end{aligned}$$

If we substitute this expression back into Equation (4.27), then the incremental weight $W^l_{n\backslash n-1}$ simplifies to

$$W^l_{n\backslash n-1} = \frac{p(Y^l_n \mid X^l_n)}{p(Y^{a^l_{n-1}}_{n-1} \mid X^{a^l_{n-1}}_{n-1})} = \prod_{y \in Y^l_{n\backslash n-1}} p(y \mid X^l_n), \tag{4.31}$$

where $Y^l_{n\backslash n-1}$ is the set difference between the observed variables at generation n and the observed variables at generation $n-1$.

$$Y^l_{n\backslash n-1} = \operatorname{dom}(\mathcal{Y}^l_n) \setminus \operatorname{dom}(\mathcal{Y}^{a^l_{n-1}}_{n-1}).$$

In other words, for a bootstrapped proposal, the importance weight at each generation is defined in terms of the joint probability of observes that have been evaluated at breakpoint n but not at $n-1$.

4.3.3 Evaluating Proposals

To implement SMC, we will introduce a function PROPOSE$(\mathcal{X}_{n-1}, y_n)$. This function evaluates the program that truncates at the observe expression with address y_n, conditioned on previously sampled values $\mathcal{X}_{n-1}$, and returns a pair $(\mathcal{X}_n, \log \Lambda_n)$ containing a map $\mathcal{X}_n$ of values associated with each sample expression and the log likelihood $\log \Lambda_n = \log p(Y_n|X_n)$. To construct the proposal for the final generation we will call PROPOSE$(\mathcal{X}_{N-1}, \texttt{nil}, y_{N-1})$, which returns a pair $(r, \log \Lambda)$ in which the return value r replaces the values $\mathcal{X}$.

In Algorithm 12 we define this function and its evaluator. When evaluating sample expressions, we reuse previously sampled values $\mathcal{X}(v)$ for previously sampled variables v and sample from the prior for new variables v. When evaluating observe expressions, we accomulate log probability into a state variable $\log \Lambda$ as we have done with likelihood weighting. When we reach the observe expression with a specified symbol

Algorithm 12 Evaluator for bootstrapped sequential Monte Carlo

```
1:  global ρ, e
2:  function EVAL(e, σ, ℓ)
3:      match e
4:          case (sample v e)
5:              d, σ ← EVAL(e, σ, ℓ)
6:              if v ∉ dom(σ(X)) then
7:                  σ(X(v)) ← SAMPLE(d)
8:              return σ(X(v)), σ
9:          case (observe v e1 e2)
10:             d, σ ← EVAL(e1, σ, ℓ)
11:             c, σ ← EVAL(e2, σ, ℓ)
12:             σ(log Λ) ← σ(log Λ) + LOG-PROB(d, c)
13:             if v = σ(y_r) then
14:                 error RESAMPLE-BREAKPOINT( )
15:             return c, σ
16:         ...                          ▷ Base cases (as in Algorithm 6)
17: function PROPOSE(X, y)
18:     σ ← [y_r ↦ y, X ↦ X, log Λ ↦ 0]
19:     try
20:         r, σ ← EVAL(e, σ, [ ])
21:         return r, σ(log Λ)
22:     catch RESAMPLE-BREAKPOINT( )
23:         return σ(X), σ(log Λ)
```

y_r, we terminate the program by throwing a special-purpose RESAMPLE-BREAKPOINT error. In the function PROPOSE, we initialize $\mathcal{X} \leftarrow \mathcal{X}_{n-1}$ and $y \leftarrow y_n$. The evaluator will then reuse all the previously sampled values $\mathcal{X}_{n-1}$ and run the program until the observe with address y_n, which samples $\mathcal{X}_n | \mathcal{X}_{n-1}$ from the program prior. We then catch the RESAMPE-BREAKPOINT error to return $(\mathcal{X}_n, \log \Lambda_n)$ for a program that truncates at y_n, and return $(r, \log \Lambda)$ when no such error occurs.

Algorithm 13 Sequential Monte Carlo with bootstrapped proposals

global ρ, e
function EVAL(e, σ, ℓ)
    ... ▷ As in Algorithm 12
function PROPOSE($\mathcal{X}$, y)
    ... ▷ As in Algorithm 12
function SMC($L, y_1, \ldots, y_{N-1}$)
    $\log \hat{Z}_0 \leftarrow 0$
    for l **in** $1, \ldots, L$ **do**
        $\mathcal{X}_1^l, \log \Lambda_1^l \leftarrow$ PROPOSE($[], y_1$)
        $\log W_1^l \leftarrow \log \Lambda_1^l$
    for n **in** $2, \ldots, N$ **do**
        $\log \hat{Z}_{n-1} \leftarrow$ LOG-MEAN-EXP($\log W_{n-1}^{1:L}$)
        for l **in** $1, \ldots, L$ **do**
            $a_{n-1}^l \sim$ DISCRETE($W_{n-1}^{1:L} / \sum_l W_{n-1}^l$)
            if $n < N$ **then**
                $(\mathcal{X}_n^l, \log \Lambda_n^l) \leftarrow$ PROPOSE($\mathcal{X}_{n-1}^{a_{n-1}^l}, y_n$)
            else
                $(r^l, \log \Lambda_N^l) \leftarrow$ PROPOSE($\mathcal{X}_{N-1}^{a_{N-1}^l}$, `nil`)
            $\log W_n^l \leftarrow \log \Lambda_n^l - \log \Lambda_{n-1}^{a_{n-1}^l} + \log \hat{Z}_{n-1}$
    return $((r^1, \log W_N^1), \ldots, (r^L, \log W_N^L))$

4.3.4 Algorithm Implementation

In Algorithm 13 we use this proposal mechanism to calculate the importance weight at each generation as according to Equation (4.31)

$$\log W_n = \log \Lambda_n - \log \Lambda_{n-1} + \hat{Z}_{n-1} \tag{4.32}$$

We calculate $\log \hat{Z}_{n-1}$ at each iteration by evaluating the function

$$\text{LOG-MEAN-EXP}(\log W_{n-1}^{1:L}) = \log \left(\frac{1}{L} \sum_{l=1}^{L} W_{n-1}^l \right). \tag{4.33}$$

4.3.5 Computational Complexity

The proposal generation mechanism in Algorithm 12 has a lot in common with the mechanism for single-site Metropolis Hastings proposals in Algorithm 10. In both evaluators, we rerun a program conditioned on previously sampled values $\mathcal{X}$. The advantage of this type of proposal strategy is that it is relatively easy to define and understand; a program in which all sample expressions evaluate to their previously sampled values is fully deterministic, so it is intuitive that we can condition on values of random variables in this manner.

Unfortunately this implementation is not particularly efficient. SMC is most commonly used in settings where we evaluate one additional observe expression for each generation, which means that the cardinality of the set of variables $|Y^l_{n\setminus n-1}|$ that determines the incremental weight in Equation (4.31) is either 1 or $\mathcal{O}(1)$. Generally this implies that we can also generate proposals and evaluate the incremental weight in constant time, which means that a full SMC sweep with L samples and N generations requires $\mathcal{O}(LN)$ computation. For this particular proposal strategy, each proposal step will require $\mathcal{O}(n)$ time, since we must rerun the program for the first n steps, which means that the full SMC sweep will require $\mathcal{O}(LN^2)$ computation.

For this reason, the SMC implementation in this section is more a proof-of-concept implementation than an implementation that one would use in practice. We will define a more realistic implementation of SMC in Section 6.7, once we have introduced an execution model based on continuations, which eliminates the need to rerun the first $n-1$ steps at each stage of the algorithm.

4.4 Black Box Variational Inference

In the sequential Monte Carlo method that we developed in the last section, we performed resampling at observes in order to obtain high quality importance sampling proposals. A different strategy for importance sampling is to learn a parameterized proposal distribution $q(X;\lambda)$ in order to maximize some notion of sample quality. In this section we will learn proposals by performing variational inference, which optimizes

the evidence lower bound (ELBO)

$$\begin{aligned}\mathcal{L}(\lambda) &:= \mathbb{E}_{q(X;\lambda)}\left[\log \frac{p(Y,X)}{q(X;\lambda)}\right], \\ &= \log p(Y) - D_{\mathrm{KL}}\left(q(X;\lambda) \,||\, p(X|Y)\right) \leq \log p(Y).\end{aligned} \tag{4.34}$$

In this definition, $D_{\mathrm{KL}}\left(q(X;\lambda) \,||\, p(X|Y)\right)$ is the KL divergence between the distribution $q(X;\lambda)$ and the posterior $p(X|Y)$,

$$D_{\mathrm{KL}}\left(q(X;\lambda) \,||\, p(X)\right) := \mathbb{E}_{q(X;\lambda)}\left[\log \frac{q(X;\lambda)}{p(X|Y)}\right]. \tag{4.35}$$

The KL divergence is a positive definite measure of dissimilarity between two distributions; it is 0 when $q(X;\lambda)$ and $p(X|Y)$ are identical and greater than 0 otherwise, which implies $\mathcal{L}(\lambda) \leq \log p(Y)$. We can therefore maximize $\mathcal{L}(\lambda)$ with respect to λ to minimize the KL term, which yields a distribution $q(X;\lambda)$ that approximates $p(X|Y)$.

In this section we will use variational inference to learn a distribution $q(X;\lambda)$ that we will then use as an importance sampling proposal. We will assume an approximation $q(X;\lambda)$ in which all variables x are independent, which in the context of variational inference is known as a mean field assumption

$$q(X;\lambda) = \prod_{x \in X} q(x;\lambda_x). \tag{4.36}$$

4.4.1 Likelihood-ratio Gradient Estimators

Black-box variational inference (BBVI) (Wingate and Weber, 2013; Ranganath et al., 2014) optimizes $\mathcal{L}(\lambda)$ by performing gradient updates using a noisy estimate of the gradient $\hat{\nabla}\mathcal{L}(\lambda)$

$$\lambda_t = \lambda_{t-1} + \eta_t \hat{\nabla}_\lambda \mathcal{L}(\lambda)\big|_{\lambda=\lambda_{t-1}}, \quad \sum_{t=1}^{\infty} \eta_t = \infty, \quad \sum_{t=1}^{\infty} \eta_t^2 < \infty. \tag{4.37}$$

BBVI uses a particular type of estimator for the gradient, which is alternately referred to as a likelihood-ratio estimator or a REINFORCE-style estimator. In general, likelihood-ratio estimators compute a Monte

Carlo approximation to an expectation of the form

$$\begin{aligned}\nabla_\lambda \mathbb{E}_{q(X;\lambda)}[r(X;\lambda)] &= \int dX\ \nabla_\lambda q(X;\lambda) r(X;\lambda) + q(X;\lambda)\nabla_\lambda r(X;\lambda) \\ &= \int dX\ \nabla_\lambda q(X;\lambda) r(X;\lambda) + \mathbb{E}_{q(X;\lambda)}[\nabla r(X;\lambda)].\end{aligned} \tag{4.38}$$

Clearly, this expression is equal to the ELBO in Equation (4.34) when we substitute $r(X;\lambda) := \log\left(p(Y,X)/q(X;\lambda)\right)$. For this particular choice of $r(X;\lambda)$, the second term in the equation above is 0,

$$\begin{aligned}\mathbb{E}_{q(X;\lambda)}\left[\nabla_\lambda \log \frac{p(Y,X)}{q(X;\lambda)}\right] &= -\mathbb{E}_{q(X;\lambda)}\left[\nabla_\lambda \log q(X;\lambda)\right] \\ &= -\int dX\ q(X;\lambda)\nabla_\lambda \log q(X;\lambda) \\ &= -\int dX\ \nabla_\lambda q(X;\lambda) = -\nabla_\lambda 1 = 0,\end{aligned} \tag{4.39}$$

where the final equalities make use of the fact that, by definition, $\int dX\ q(X;\lambda) = 1$ since a probability distribution is normalized.

If we additionally substitute $\nabla_\lambda q(X;\lambda) := q(X;\lambda)\nabla_\lambda \log q(X;\lambda)$ in Equation (4.38), then we can express the gradient of the ELBO as

$$\nabla_\lambda \mathcal{L}(\lambda) = \mathbb{E}_{q(X;\lambda)}\left[\nabla_\lambda \log q(X;\lambda)\left(\log \frac{p(Y,X)}{q(X;\lambda)} - b\right)\right], \tag{4.40}$$

where b is arbitrary constant vector, which does not change the expected value since $\mathbb{E}_{q(X;\lambda)}[\nabla_\lambda \log q(X;\lambda)] = 0$.

The likelihood-ratio estimator for the gradient of the ELBO approximates the expectation with a set of samples $X^l \sim q(X;\lambda)$. If we define the standard importance weight $W^l = p(Y^l, X^l)/q(X^l;\lambda)$, the the likelihood-ratio estimator is defined as

$$\hat{\nabla}_\lambda \mathcal{L}(\lambda) := \frac{1}{L}\sum_{l=1}^{L} \nabla_\lambda \log q(X^l;\lambda)\left(\log W^l - \hat{b}\right). \tag{4.41}$$

Here we set $\hat{b}$ to a value that minimizes the variance of the estimator. If we use $(\lambda_{v,1}, \ldots, \lambda_{v,D_v})$ to refer to the components of the parameter

Algorithm 14 Evaluator for Black Box Variational Inference

```
1:  global ρ
2:  function EVAL(e, σ, ℓ)
3:      match e
4:          case (sample v e)
5:              p, σ ← EVAL(e, σ, ℓ)
6:              if v ∉ dom(σ(Q)) then
7:                  σ(Q(v)) ← p                ▷ Initialize proposal using prior
8:              c ∼ SAMPLE(σ(Q(v)))
9:              σ(G(v)) ← GRAD-LOG-PROB(σ(Q(v)), c)
10:             log W_v ← LOG-PROB(p, c) − LOG-PROB(σ(Q(v)), c)
11:             σ(log W) ← σ(log W) + log W_v
12:             return c, σ
13:         case (observe v e_1 e_2)
14:             p, σ ← EVAL(e_1, σ, ℓ)
15:             c, σ ← EVAL(e_2, σ, ℓ)
16:             σ(log W) ← σ(log W) + LOG-PROB(p, c)
17:             return c, σ
18:         ...                                ▷ Base cases (as in Algorithm 6)
```

vector λ_v, then the variance reduction constant $\hat{b}_{v,d}$ for the component $\lambda_{v,d}$ is defined as

$$\hat{b}_{v,d} := \frac{\operatorname{covar}(F_{v,d}^{1:L}, G_{v,d}^{1:L})}{\operatorname{var}(G_{v,d}^{1:L})}, \tag{4.42}$$

$$F_{v,d}^{l} := \nabla_{\lambda_{v,d}} \log q(X_v^l; \lambda_v) \log W^l, \tag{4.43}$$

$$G_{v,d}^{l} := \nabla_{\lambda_{v,d}} \log q(X_v^{1:L}; \lambda_v). \tag{4.44}$$

4.4.2 Evaluator for Gradient Estimation

From the equations above, we see that we need to calculate two sets of quantities in order to estimate the gradient of the ELBO. The first consists of the importance weights $\log W^l$. The second consists of the gradients of the log proposal density for each variable $G_v^l =$

Algorithm 15 Black Box Variational Inference

global ρ, e
function EVAL(e, σ, ℓ)
    ... ▷ As in Algorithm 14
function OPTIMIZER-STEP(q, $\hat{g}$)
    for v **in** dom($\hat{g}$) **do**
        $\lambda(v) \leftarrow$ GET-PARAMETERS($\mathcal{Q}(v)$)
        $\lambda'(v) \leftarrow \lambda(v) + \ldots$ ▷ SGD/Adagrad/Adam update
        $\mathcal{Q}'(v) \leftarrow$ SET-PARAMETERS($\mathcal{Q}(v), \lambda'$)
    return $\mathcal{Q}'$
function ELBO-GRADIENTS($G^{1:L}$, $\log W^{1:L}$)
    for v **in** dom(G^1) $\cup \ldots \cup$ dom(G^L) **do**
        for l **in** $1, \ldots, L$ **do**
            if $v \in$ dom(G^l) **then**
                $F^l(v) \leftarrow G^l(v) \log W^{1:L}$
            else
                $F^l(v), G^l(v) \leftarrow 0, 0$
        $\hat{b} \leftarrow$ SUM(COVAR($F^{1:L}(v), G^{1:L}(v)$))/SUM(VAR($G^{1:L}(v)$))
        $\hat{g}(v) \leftarrow$ SUM($F^{1:L}(v) - \hat{b}\, G^{1:L}(v)$)/$L$
    return $\hat{g}$
function BBVI(S, L)
    $\sigma \leftarrow [\log W \mapsto 0, q \mapsto [\,], G \mapsto [\,]]$
    for t **in** $1, \ldots, T$ **do**
        for l **in** $1, \ldots, L$ **do**
            $r^{t,l}, \sigma^{t,l} \leftarrow$ EVAL($e, \sigma, [\,]$)
            $G^{t,l}, \log W^{t,l} \leftarrow \sigma^{t,l}(G), \sigma^{t,l}(\log W)$
        $\hat{g} \leftarrow$ ELBO-GRADIENTS($G^{s,1:L}, \log W^{s,1:L}$)
        $\sigma(\mathcal{Q}]) \leftarrow$ OPTIMIZER-STEP($\sigma(\mathcal{Q}), \hat{g}$)
    return $((r^{1,1}, \log W^{1,1}), \ldots, (r^{1,L}, \log W^{1,L}), \ldots, (r^{T,L}, \log W^{T,L}))$

$\nabla_{\lambda_v} \log q(X_v^l | \lambda_v)$.

In Algorithm 14 we define an evaluator that extends the likelihood-ratio evaluator from Algorithm 7 in two ways:

1. Instead of sampling proposals from the program prior, we now

propose from a distribution $\mathcal{Q}(v)$ for each variable v and update the importance weight $\log W$ accordingly.

2. When evaluating a sample expression, we additionally calculate the gradient of the log proposal density $G(v) = \nabla_{\lambda_v} \log q(X_v|\lambda_v)$. For this we assume an implementation of a function GRAD-LOG-PROB(d, c) for each primitive distribution type supported by the language.

Algorithm 15 defines a BBVI algorithm based on this evaluator. The function ELBO-GRADIENTS returns a map $\hat{g}$ in which each entry $\hat{g}(v) := \hat{\nabla}_{\lambda_v} \mathcal{L}(\lambda)$ contains the gradient components for the variable v as defined in Equations (4.41)-(4.44). The main algorithm BBVI then simply runs the evaluator L times at each iteration and then passes the computed gradient estimates $\hat{g}$ to a function OPTIMIZER-STEP, which can either implement the vanilla stochastic gradient updates defined in Equation (4.37), or more commonly updates for an extension of stochastic gradient descent such as Adam (Kingma and Ba, 2015) or Adagrad (Duchi et al., 2011).

4.4.3 Computational Complexity and Statistical Efficiency

From an implementation point of view, BBVI is a relatively simple algorithm. The main reason for this is the mean field approximation for $q(X; \lambda)$ in Equation (4.36). Because of this approximation, calculating the gradients $\nabla_\lambda \log q(X; \lambda)$ is easy, since we can calculate the gradients $\nabla_{\lambda_v} \log q(X_v; \lambda_v)$ for each component independently, which only requires that we implement gradients of the log density for each primitive distribution type.

One of the main limitations of this BBVI implementation is that the gradient estimator tends to be relatively high variance, which means that we will need a relatively large number of samples per gradient step L in order to ensure convergence. Values of L of order 10^2 or 10^3 are not uncommon, depending on the complexity of the model. For comparison, methods for variational autoencoders that compute the gradient of a reparameterized objective (Kingma and Welling, 2014; Rezende et al., 2014) can be evaluated with $L = 1$ samples for many models. In addition to this, the number of iterations T that is needed to achieve convergence

can easily be order 10^3 to 10^4. This means that BBVI we may need order 10^6 or more samples before BBVI starts generating high quality proposals.

When we compile a program to a graph $(V, A, \mathcal{P}, \mathcal{Y})$ we can perform an additional optimization to reduce the variance. To do so, we replace the term $\log W$ in the objective with a vector in which each component $\log W_v$ contains a weight that is restricted to the variables in the Markov blanket,

$$\log W_v = \sum_{w \in \text{MB}(v)\}} \frac{p(w|\text{PA}(w))}{q(w|\lambda_w)}, \tag{4.45}$$

where the Markov blanket $\text{MB}(v)$ of a variable v is

$$\begin{aligned} \text{MB}(v) = \text{PA}(v) &\cup \{w : w \in \text{PA}(v)\} \\ &\cup \left\{w : \exists u \Big(v \in \text{PA}(u) \wedge w \in \text{PA}(u) \Big) \right\}. \end{aligned} \tag{4.46}$$

This can be interpreted as a form of Rao-Blackwellization (Ranganath et al., 2014), which reduces the variance by ignoring the components of the weight that are not directly associated with the sampled value X_v. In a graph-based implementation of BBVI, one can easily construct this Markov blanket, which we rely upon in the implementation of Gibbs sampling 3.3.

5

A Probabilistic Programming Language With Recursion

In the three preceding chapters we have introduced a first-order probabilistic programming language and described graph- and evaluation-based inference methods. The defining characteristic of the FOPPL is that it is suitably restricted to ensure that there can only ever be a finite number of random variables in any model denoted by a program.

In this chapter we relax this restriction by introducing a higher-order probabilistic programming language (HOPPL) that supports programming language features, such as higher-order procedures and general recursion. HOPPL programs can denote models with an unbounded number of random variables. This rules out graph-based evaluation strategies immediately, since an infinite graph cannot be represented on a finite-capacity computer. However, it turns out that evaluation-based inference strategies can still be made to work by considering only a finite number of random variables at any particular time, and this is what will be discussed in the subsequent chapter.

In the FOPPL, we ensured that programs could be compiled to a finite graph by placing a number of restrictions on the language:

- The `defn` forms disallow recursion;
- Functions are not first class objects, which means that it is not

possible to write higher-order functions that accept functions as arguments;

- The first argument to the `loop` form, the loop depth, has to be a constant, as `loop` was syntactic sugar unrolled to nested let expressions at compile time.

Say that we wish to remove this last restriction, and would like to be able to loop over the range determined by the runtime value of a program variable.

This means that the looping construct cannot be syntactic sugar, but must instead be a function that takes the loop bound as an argument and repeats the execution of the loop body up to this dynamically-determined bound.

If we wanted to implement a loop function that supports a dynamic number of loop iterations, then we could do so as follows

```
(defn loop-helper [i c v f a_1 ... a_n]
  (if (= i c)
      v
      (let [v' (f i v a_1 ... a_n)]
         (loop-helper (+ i 1) c v' f a_1 ... a_n))))
(defn loop [c v f a_1 ... a_n]
  (loop-helper 0 c v f a_1 ... a_n)).
```

In order to implement this function we have to allow the `defn` form to make recursive calls, a large departure from the FOPPL restriction. Doing so gives us the ability to write programs that have loop bounds that are determined at runtime rather than at compile time, a feature that most programmers expect to have at their disposal when writing any program. However, as soon as loop is a function that takes a runtime value as a bound, then we could write programs such as

```
(defn flip-and-sum [i v]
  (+ v (sample (bernoulli 0.5))))
(let [c (sample (poisson 1))]
  (loop c 0 flip-and-sum)).
```

This program, which represents the distribution over the sums of the outcomes of a Poisson distributed number of of fair coin flips, is one of the shortest programs that illustrates concretely what we mean by a

program that denotes an infinite number of random variables. Although this program is not particularly useful, we will soon show many practical reasons to write programs like this. If one were to attempt the loop desugaring approach of the FOPPL here one would need to desugar this loop for all of the possible constant values c could take. As the support of the Poisson distribution is unbounded above, one would need to desugar a loop indefinitely, leading to an infinite number of random variables (the Bernoulli draws) in the expanded expression. The corresponding graphical model would have an infinite number of nodes, which means that it is no longer possible to compile this model to a graph.

The unboundedness of the number of random variables is the central issue. It arises naturally when one uses stochastic recursion, a common way of implementing certain random variables. Consider the example

```
(defn geometric-helper [n dist]
  (if (sample dist)
      n
      (geometric-helper (+ n 1))))
(defn geometric [p]
 (let [dist (flip p)]
    (geometric-helper 0 dist))).
```

This is a well-known sampler for geometrically distributed random variables. Although a primitive for the geometric distribution would definitely be provided by a probabilistic programming language (e.g. in the FOPPL), the point of this example is to demonstrate that the use of infinitely many random variables arises with the introduction of stochastic recursion. Notably, here, it could be that this particular computation never terminates, as at each stage of the recursion `(sample dist)` could return `false`, with probability `p`. Leveraging referential transparency, one could attempt to inline the helper function above as

```
(defn geometric [p]
 (let [dist (flip p)]
    (if (sample dist)
      0
      (if (sample dist)
        1
        (if (sample dist)
          2
          ⋮
            (if (sample dist)
               ∞
               (geometric-helper (+ ∞ 1))))))))
```

but the problem in attempting to do so quickly becomes apparent. Without a deterministic loop bound, the inlining cannot be terminated, showing that the denoted model has an infinite number of random variables. No inference approach which requires eager evaluation of if statements, such as the graph compilation techniques in the previous chapter, can be applied in general.

While expanding the class of denotable models is important, the primary reason to introduce the complications of a higher-order modeling language is that ultimately we would like simply to be able to do probabilistic programming using any *existing* programming language as the modeling language. If we make this choice, we need to be able to deal with all of the possible models that could be written in said language and, in general, we will not be able to syntactically prohibit stochastic loop bounds or conditioning on data whose size is known only at runtime. Furthermore, in the following chapter we will show how to do probabilistic programming using not just an existing language syntax but also an existing compiler and runtime infrastructure. Then, we may not even have access to the source code of the model. A probabilistic programming approach that extends an existing language in this manner will typically target a family of models that are, roughly speaking, in the same class as models that can be defined using the HOPPL.

5.1 Syntax

Relative to the first-order language in Chapter 2, the higher-order language that we introduce here has two additional features. The first is that functions can be recursive. The second is that functions are first-class values in the language, which means that we can define higher-order functions (i.e. functions that accept other functions as arguments). The syntax for the HOPPL is shown in Language 5.4.

v ::= variable
c ::= constant value or primitive operation
f ::= procedure
e ::= c | v | f | `(if` e e e`)` | `(`e $e_1 \ldots e_n$`)` | `(sample` e`)`
| `(observe` e e`)` | `(fn [`$v_1 \ldots v_n$`]` e`)`
q ::= e | `(defn` f `[`$v_1 \ldots v_n$`]` e`)` q.

Language 5.4: Higher-order probabilistic programming language (HOPPL)

While a procedure had to be declared globally in the FOPPL, functions in the HOPPL can be created locally using an expression `(fn` $[v_1 \ldots v_n]$ e`)`. Also, the HOPPL lifts the restriction of the FOPPL that the operators in procedure calls are limited to globally declared procedures f or primitive operations c; as the case `(`e $e_1 \ldots e_n$`)` in the grammar indicates, a general expression e may appear as an operator in a procedure call in the HOPPL. Finally, the HOPPL drops the constraint that all procedures are non-recursive. When defining a procedure f using `(defn` f $[v_1 \ldots v_n]$ e`)` in the HOPPL, we are no longer forbidden to call f in the body e of the procedure.

These features are present in Church, Venture, Anglican, and WebPPL and are required to reason about languages like Probabilistic-C, Turing, and CPProb. In the following we illustrate the benefits of having these features by short evocative source code examples of some kinds of advanced probabilistic models that can now be expressed. In the next chapter we describe a class of inference algorithms suitable for performing inference in the models that are denotable in such an expressive higher-order probabilistic programming language.

5.2 Syntactic sugar

We will define syntactic sugar that re-establishes some of the convenient syntactic features of the HOPPL. Note that the syntax of the HOPPL omits the `let` expression. This is because it can be defined in terms of nested functions as

```
(let [x e1] e2) = ((fn [x] e2) e1).
```

For instance,

```
(let [a (+ k 2)
      b (* a 6)]
  (print (+ a b))
  (* a b))
```

gets first desugared to the following expression

```
(let [a (+ k 2)]
  (let [b (* a 6)]
    (let [c (print (+ a b))]
      (* a b))))
```

where `c` is a fresh variable. This can then be desugared to the expression without `let` as follows

```
((fn [a]
   ((fn [b]
     ((fn [c] (* a b))
      (print (+ a b))))
    (* a 6)))
 (+ k 2)).
```

While we already described a HOPPL `loop` implementation in the preceding text, we have elided the fact that the FOPPL `loop` accepts a variable number of arguments, a language feature we have not explicitly introduced here. An exact replica of the FOPPL loop can be implemented as HOPPL sugar, with loop desugaring occurring prior to the let desugaring. If we define the helper function

```
(defn loop-helper [i c v g]
  (if (= i c)
      v
      (let [v' (g i v)]
         (loop-helper (+ i 1) c v' g))))
```

the expression (loop c e f e_1 $\cdots$ e_n) can be desugared to

```
(let [bound c
      initial-value e
      a_1 e_1
          ⋮
      a_n e_n
      g (fn [i w] (f i w a_1 ... a_n))]
  (loop-helper 0 bound initial-value g)).
```

With this loop and let sugar defined, and the implementation of foreach straightforward, any valid FOPPL program is also valid in the HOPPL.

5.3 Examples

In the HOPPL, we will employ a number of design patterns from functional programming, which allow us to write more conventional code than was necessary to work around limitations of the FOPPL. Here we give some examples of higher-order function implementations and usage in the HOPPL before revisiting models previously discussed in chapter 2 and introducing new examples which depend on new language features.

Examples of higher-order functions

We will frequently rely on higher-order functions `map` and `reduce`. We can write these explicitly as HOPPL functions which take functions as arguments, and do so here by way of introduction to HOPPL usage before considering generative model code.

Map. The higher-order function `map` takes two arguments: a function and a sequence. It then returns a new sequence, constructed by applying the function to every individual element of the sequence.

```
(defn map [f values]
  (if (empty? values)
    values
    (prepend (map f (rest values))
             (f (first values)))))
```

Here `prepend` is a primitive that prepends a value to the beginning of a sequence. This "loop" works by applying `f` to the first element of the collection `values`, and then recursively calling `map` with the same function on the rest of the sequence. At the base case, for an empty input `values`, we return the empty sequence of values.

Reduce. The `reduce` operation, also known as "fold", takes a function and a sequence as input, along with an initial state; unlike `map`, it returns a single value. The fixed-length `loop` construct we defined as syntactic sugar in the FOPPL can be thought of as a poor-man's `reduce`. The function passed to `reduce` takes a state and a value, and computes a new state. We get the output by repeatedly applying the function to the current state and the first item in the list, recursively processing the rest of the list.

```
(defn reduce [f x values]
  (if (empty? values)
    x
    (reduce f (f x (first values)) (rest values))))
```

Whereas `map` is a function that maps a sequence of values onto a sequence of function outputs, `reduce` is a function that produces a single result. An example of where you might use `reduce` is when writing a function that computes the sum of all entries in a sequence:

```
(defn sum [items]
  (reduce + 0.0 items))
```

Note that the output of `reduce` depends on the return type of the provided function. For example, to return a list with the same entries as the original list, but reversed, we can use a `reduce` with a function that builds up a list from back-to-front:

```
(defn reverse [values]
  (reduce prepend [] values))
```

No need to inline data. A consequence of allowing unbounded numbers of random variables in the model is that we no longer need to "inline" our data. In the FOPPL, each `loop` needed to have an explicit integer literal representing the total number of iterations in order to

desugar to `let` forms. As a result, each program that we wrote had to hard-code the total number of instances in any dataset. Flexible looping structures mean we can read data into the HOPPL in a more natural way; assuming libraries for e.g. file access, we could read data from disk, and use a recursive function to loop through entries until reaching the end of the file.

For example, consider the hidden Markov model in the FOPPL given by Program 2.5. In that implementation, we hard coded the number of loop iterations (there, 16) to the length of the data. In the HOPPL, suppose instead we have a function which can read the data in regardless of its length.

```
(defn read-data []
  (read-data-from-disk "filename.csv"))

;; Sample next HMM latent state and condition
(defn hmm-step [trans-dists obs-dists]
  (fn [states data]
    (let [state (sample (get trans-dists
                             (last states)))]
      (observe (get obs-dists state) data)
      (conj states state))))

(let [trans-dists [(discrete [0.10 0.50 0.40])
                   (discrete [0.20 0.20 0.60])
                   (discrete [0.15 0.15 0.70])]
      obs-dists [(normal -1.0 1.0)
                 (normal 1.0 1.0)
                 (normal 0.0 1.0)]
      state [(sample (discrete [0.33 0.33 0.34]))]]
  ;; Loop through the data, return latent states
  (reduce (hmm-step trans-dists obs-dists)
          [state]
          (read-data)))
```

The `hmm-step` function now takes a vector containing the current states, and a *single* data point, which we `observe`. Rather than using an explicit iteration counter n, we can use `reduce` to traverse the data recursively, building up and returning a vector of visited states.

Open-universe Gaussian Mixtures

The ability to write loops of unknown or random iterations is not just a handy tool for writing more readable code; as hinted by the recursive geometric sampler example, it also increases the expressivity of the model class. Consider the Gaussian mixture model example we implemented in the FOPPL in Program 2.4: there we had two explicit loops, one over the number of data points, but the other over the number of mixture components, which we had to fix at compile time. As an alternative, we can re-write the Gaussian mixture to define a distribution over the number of components. We do this by introducing a prior over the number of mixture components; this prior could be e.g. a Poisson distribution, which places non-zero probability on all positive integers.

To implement this, we can define a higher-order function, `repeatedly`, which takes a number n and a function f, and constructs a sequence of length n where each entry is produced by invoking f.

```
(defn repeatedly [n f]
  (if (<= n 0)
    []
    (append (repeatedly (- n 1) f) (f))))
```

The `repeatedly` function can stand in for the fixed-length loops that we used to sample mixture components from the prior in the FOPPL implementation. An example implementation is in Program 5.5.

```
(defn sample-likelihood []
  (let [sigma (sample (gamma 1.0 1.0))
        mean (sample (normal 0.0 sigma))]
    (normal mean sigma)))

(let [ys [1.1 2.1 2.0 1.9 0.0 -0.1 -0.05]
      K (sample (poisson 3)) ;; random, with mean 3
      ones (repeatedly K (fn [] 1.0))
      z-prior (discrete (sample (dirichlet ones)))
      likes (repeatedly K sample-likelihood)]
  (map (fn [y]
         (let [z (sample z-prior)]
           (observe (nth likes z) y)
           z))
```

```
      ys))
```

Program 5.5: HOPPL: An open-universe Gaussian mixture model with an unknown number of components

Here we still used a fixed, small data set (the `ys` values, same as before, are inlined) but the model code would not change if this were replaced by a larger data set. Models such as this one, where the distribution over the number of mixture components K is unbounded above, are sometimes known as *open-universe* models: given a small amount of data, we may infer there are only a small number of clusters; however, if we were to add more and more entries to `ys` and re-run inference, we do not discount the possibility that there are additional clusters (i.e. a larger value of K) than we had previously considered.

Notice that the way we wrote this model interleaves sampling from z with observing values of y, rather than sampling all values $z_1, z_2, z_3, \ldots$ up front. While this does not change the definition of the model (i.e. does not change the joint distribution over observed and latent variables), writing the model in a formulation which moves `observe` statements as early as possible (or alternatively delays calls to `sample`) yields more efficient SMC inference.

Sampling with constraints

One common design pattern involves simulating from a distribution, subject to constraints. Obvious applications include sampling from truncated variants of known distributions, such as a normal distribution with a positivity constraint; however, such rejection samplers are in fact much more common than this. In fact, samplers for most standard distributions (e.g. Gaussian, gamma, Dirichlet) are implemented under the hood as rejection samplers which propose from some known simpler distribution, and evaluate an acceptance criteria; they continue looping until the criteria evaluates to true.

In a completely general form, we can write this algorithm as a higher-order function which takes two functions as arguments: a `proposal` function which simulates a candidate point, and `is-valid?` which returns true when the value passed satisfies the constraint.

```
(defn rejection-sample [proposal is-valid?]
```

```
(let [value (proposal)]
  (if (is-valid? value)
    value
    (rejection-sample proposal is-valid?))))
```

This sort of accept-reject algorithm can take an unknown number of iterations, and thus cannot be expressed in the FOPPL.

The `rejection-sample` function can be used to implement samplers for distributions which do not otherwise have samplers, for example when sampling from constrained in simulation-based models in the physical sciences.

Program synthesis

As a more involved modeling example which cannot be written without exploiting higher-order language features, we consider writing a generative model for mathematical functions. The representation of functions we will use here is actually literal code written in the HOPPL: that is, our generative model will produce samples of function bodies `(fn [] ...)`. For purposes of illustration, suppose we restrict to simple arithmetic functions of a single variable, which we could generate using the grammar

```
op ::= + | - | * | /
num ::= 0 | 1 | ... | 9
e ::= num | x | (op e e)
f ::= (fn [x] (op e e))
```

We can sample from the space of all functions $f(x)$ generated by composition of digits with `+`, `-`, `*`, and `/`, by starting from the initial rule for expanding f and recursively applying rules to fill in values of *op*, *num*, and e until only terminals remain. To do so, we need to assign a probability for sampling each rule at each stage of the expansion. In the following example, when expanding each e we choose a number with probability 0.4, the symbol x with probability 0.3, and a new function application with probability 0.3; both operations and numbers $0, \ldots, 9$ are chosen uniformly.

```
(defn gen-operation []
  (sample (uniform [+ - * /])))
```

```
(defn gen-expr []
  (let [expr-prior (discrete [0.4 0.3 0.3])
        expr-type (sample expr-prior)]
    (case expr-type
      0 (sample (uniform-discrete 0 10))
      1 (quote x)
      2 (let [operation (gen-operation)]
          (list operation
                (gen-expr)
                (gen-expr))))))

(defn gen-function []
  (list (quote fn) [(quote x)]
    (list (gen-operation)
          (gen-expr)
          (gen-expr))))
```

Program 5.6: generative model for function of a single variable

In this program we make use of two constructs that we have not previously encountered. The first is the `(case` v e_1 $\ldots$ e_n`)` form, which is syntactic sugar that allows us to select between more than two branches, depending on the value of the variable v. The second is the `list` data type. A call `(list 1 2 3)` returns a list of values `(1 2 3)`. We differentiate a list from a vector by using round parentheses `(...)` rather than squared parentheses `[...]`.

In this program we see one of the advantages of a language which inherits from LISP and Scheme: programmatically generating code in the HOPPL is quite straightforward, requiring only standard operations on a basic `list` data type. The function `gen-function` in Program 5.6 returns a list, not a "function". That is, it does not directly produce a HOPPL function which we can call, but rather the source code for a function. In defining the source code, we used the `quote` function to wrap keywords and symbols in the source code, e.g. `(quote x)`. This primitive prevents the source code from being evaluated, which means that the variable name `x` is included into the list, rather than the value of the variable (which does not exist). Repeated invocation of `(gen-function)` produces samples from the grammar, which can be used as a basic diagnostic:

```
(fn [x] (- (/ (- (* 7 0) 2) x) x))
(fn [x] (- x 8))
(fn [x] (* 5 8))
(fn [x] (+ 7 6))
(fn [x] (* x x))
(fn [x] (* 2 (+ 0 1)))
(fn [x] (/ 6 x))
(fn [x] (- 0 (+ 0 (+ x 5))))
(fn [x] (- x 6))
(fn [x] (* 3 x))
(fn [x]
  (+ (+ 2
        (- (/ x x)
           (- x (/ (- (- 4 x) (* 5 4))
                   (* 6 x)))))
     x))
(fn [x] (- x (+ 7 (+ x 4))))
(fn [x] (+ (- (/ (+ x 3) x) x) x))
(fn [x] (- x (* (/ 8 (/ (+ x 5) x)) (- 0 1))))
(fn [x] (/ (/ x 7) 7))
(fn [x] (/ x 2))
(fn [x] (* 8 x))
```

Program 5.7: Unconditioned samples from a generative model for arithmetic expressions, produced by calling `(gen-function)`

Most of the generated expressions are fairly short, with many containing only a single function application. This is because the choice of probabilities in Program 5.6 is biased towards avoiding nested function applications; the probability of producing a number or the variable x is 0.7, a much larger value than the probability 0.3 of producing a function application. However, there is still positive probability of sampling an expression of any arbitrarily large size — there is nothing which explicitly bounds the number of function applications in the model. Such a model could not be written in the FOPPL without introducing a hard bound on the recursion depth. In the HOPPL we can allow functions to grow long if necessary, while still preferring short results, thanks to the eager evaluation of `if` statements and the lack of any need to enumerate possible random choices.

Note that some caution is required when defining models which can generate a countably infinite number of latent random variables:

it is possible to write programs which do not necessarily terminate. In this example, had we assigned a high enough probability to the expansion rule $e \rightarrow$ `(op` e e`)`, then it is possible that, with positive probability, the program *never* terminates. In contrast, it is not possible to inadvertently write an infinite loop in the FOPPL.

If we wish to fit a function to data, it is not enough to merely generate the source code for the function — we also need to actually evaluate it. This step actually requires invoking either a compiler or an interpreter to parse the symbolic representation of the function (i.e., as a list containing symbols) and evaluate it to a user-defined function, just as if we had included the expression `(fn [`x`]...)` in our original program definition. The magic word is `eval`, which we assume to be supplied as a primitive in the HOPPL target language. We use `eval` to evaluate code that has previously been quoted with `quote`. Consider the function `(fn [x] (- x 8))`. Using `quote`, we can define source code (in the form of a list) that could then be evaluated to produce the function itself,

```
;; These two lines are identical:
(eval (quote (fn [x] (- x 8))))
(fn [x] (- x 8))
```

For our purposes, we will want to evaluate the generated functions at particular inputs to see how well they conform to some specific target data, e.g.

```
;; Calling the function at x=10 (outputs: 2)
(let [f (eval (quote (fn [x] (- x 8))))]
  (f 10))
```

Running a single-site Metropolis-Hastings sampler, using an algorithm similar to that in Section 4.2 (which we will describe precisely in Section 6.6), we can draw posterior samples given particular data. Some example functions are shown in Figure 5.1, conditioning on three input-output pairs.

Captcha-breaking

We can also now revisit the Captcha-breaking example we discussed in Chapter 1, and write a generative model for Captcha images in the

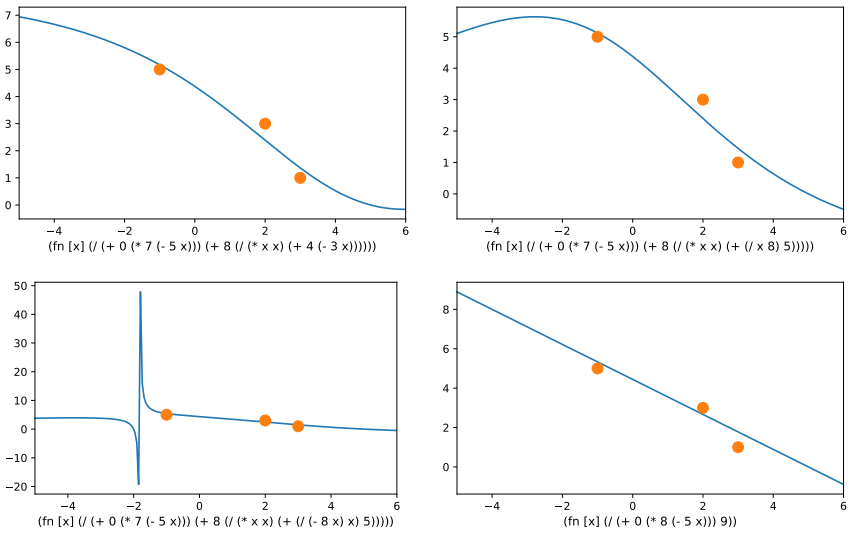

Figure 5.1: Examples of posterior sampled functions, drawn from the same MH chain.

HOPPL. Unlike the FOPPL, the HOPPL is a fully general programming language, and could be used to write functions such as a Captcha renderer which produces images similar to those in Figure 1.1. If we write a `render` function, which takes as input a string of text to encode and a handful of parameters governing the distortion, and returns the a rendered image, it is straightforward to then include this function in a probabilistic program that then can be used for inference. We simply define a distribution (perhaps even uniform) over text and parameters

```
;; Define a function to sample a single character
(defn sample-char []
  (sample (uniform ["a" "b" ... "z"
                    "A" "b" ... "Z"
                    "0" "1" ... "9"])))

;; Define a function to generate a Captcha
(defn generate-captcha [text]
  (let [char-rotation (sample (normal 0 1))
        add-distortion? (sample (flip 0.5))
        add-lines? (sample (flip 0.5))
```

```
        add-background? (sample (flip 0.4))]
    ;; Render a Captcha image
    (render text char-rotation
            add-distortion? add-lines? add-background?)))
```

and then to perform inference on the text

```
(let [image ( ... ) ;; read target Captcha from disk
      num-chars (sample (poisson 4))
      text (repeatedly num-chars sample-char)
      generated (generate-captcha text)]
  ;; score using any image similarity measure
  (factor (image-similarity image generated))
  text)
```

Here we treated the `render` function as a black box, and just assumed it could be called by the HOPPL program. In fact, so long as `render` has purely deterministic behavior and no side-effects it can actually be written in another language entirely, or even be a black-box precompiled binary; it is just necessary that it can be invoked in some manner by the HOPPL code (e.g. through a foreign function interface, or some sort of inter-process communication).

6

Evaluation-Based Inference II

Programs in the HOPPL can represent an unbounded number of random variables. In such programs, the compilation stategies that we developed in Chapter 3 will not terminate, since the program represents a graphical model with an infinite number of nodes and vertices. In Chapter 4, we developed inference methods that generate samples by evaluating a program. In the context of the FOPPL, the defining difference between graph-based methods and evaluation-based methods lies in the semantics of if forms, which are evaluated eagerly in graph-based methods and lazily in evaluation-based methods. In this chapter, we generalize evaluation-based inference to probabilistc programs in general-purpose languages such as the HOPPL. A simple yet important insight behind this strategy is that every terminating execution of an HOPPL program works on only finitely many random variables, so that program evaluation provides a systematic way to select a finite subset of random variables used in the program.

As in Chapter 4, the inference algorithms in this chapter use program evaluation as one of their core subroutines. However, to more clearly illustrate how evaluation-based inference can be implemented by extending existing languages, we abandon the definition of inference

algorithms in terms of evaluators in favor of a more language-agnostic formulation; we define inference methods as non-standard schedulers of HOPPL programs. The guiding intuition in this formulation is that the majority of operations in HOPPL programs are deterministic and referentially transparent, with the exception of `sample` and `observe`, which are stochastic and have side-effects. In the evaluators in Chapter 4, this is reflected in the fact that only `sample` and `observe` expressions are algorithm specific; all other expression forms are always evaluated in the same manner. In other words, a probabilistic program is a computation that is mostly inference-agnostic. The abstraction that we will employ in this chapter is that of a program as a deterministic computation that can be interrupted at `sample` and `observe` expressions. Here, the program cedes control to an inference controller, which implements probabilistic and stochastic operations in an algorithm-specific manner.

Representing a probabilistic program as an interruptible computation can also improve computational efficiency. If we implement an operation that "forks" a computation in order to allow multiple independent evaluations, then we can avoid unnecessary re-evaluation during inference. In the single-site Metropolis-Hastings algorithm in Chapter 4, we re-evaluate a program in its entirety for every update, even when this update only changes the value of a single random variable. In the sequential Monte Carlo algorithm, the situation was even worse; we needed to re-evaluate the program at `observe`, which lead to an overall runtime that is quadratic in the number of observations, rather than linear. As we will see, forking the computation at `sample` and `observe` expressions avoids this re-evaluation, while this forking operation almost comes for free in languages such as the HOPPL, in which there are no side effects outside of `sample` and `observe`.

6.1 Explicit separation of model and inference code

A primary advantage of using a higher-order probabilistic programming language is that we can leverage existing compilers for real-world languages, rather than writing custom evaluators and custom languages. In the interface we consider here, we assume that a probabilistic program is a deterministic computation that is interrupted at every `sample` and

`observe` expression. Inference is carried out using a "controller" process. The controller needs to be able to start executions of a program, receive the return value when an execution terminates, and finally control program execution at each `sample` and `observe` expression.

The inference controller interacts with program executions via a messaging protocol. When a program reaches a `sample` or `observe` expression, it sends a message back to the controller and waits a response. This message will typically include a unique identifier (i.e. an address) for the random variable, and a representation of the fully-evaluated arguments to `sample` and `observe`. The controller then performs any operations that are necessary for inference, and sends back a message to the running program. The message indicates whether the program should continue execution, fork itself and execute muliple times, or halt. In the case of `sample` forms, the inference controller must also provide a value for the random variable (when continuing), or multiple values for the random variable (when forking).

This interface defines an abstraction boundary between program execution and inference. From the perspective of the inference controller, the deterministic steps in the execution of a probabilistic program can be treated as a black box. As long as the program executions implement the messaging interface, inference algorithms can be implemented in a language-agnostic manner. In fact, it is not even necessary that the inference algorithm and the program are implemented in the same language, or execute on the same physical machine. We will make this idea explicit in Section 6.4.

Example: likelihood weighting. To build intuition, we begin by outlining how a controller could implement likelihood weighting using a messaging inteface (a precise specification will be presented in Section 6.5). In the evaluation-based implementation of likelihood weighting in Section 4.1, we evaluate `sample` expressions by drawing from the prior, and increment the log importance weight at every `observe` expresion. The controller for this inference strategy would repeat the following operations:

- The controller starts a new execution of the HOPPL program,

and iniatilizes its log weight $\log W = 0.0$;

- The controller repeatedly receives messages from the running program, and dispatches based on type:
  - At a `(sample` d`)` form, the controller samples x from the distribution d and sends the sampled value x back to the program to continue execution;
  - At an `(observe` d c`)` form, the controller increments $\log W$ with the log probability of c under d, and sends a message to continue execution;
  - If the program has terminated with value c, the controller stores a weighted sample $(c, \log W)$ and exits the loop.

Messaging Interface. In the inference algorithm above, a program pauses at every `sample` and `observe` form, where it sends a message to the inference process and awaits a response. In likelihood weighting, the response is always to continue execution. To support algorithms such as sequential Monte Carlo, the program execution process will additionally need to implement a forking operation, which starts multiple independent processes that each resume from the same point in the execution.

To support these operations, we will define an interface in which an inference process can send three messages to the execution process:

1. (`"start"`, σ): Start a new execution with process id σ.
2. (`"continue"`, σ, c): Continue execution for the process with id σ, using c as the argument value.
3. (`"fork"`, σ, σ', c): Fork the process with id σ into a new process with id σ' and continue execution with argument c.
4. (`"kill"`, σ): Terminate the process with id σ.

Conversely, we will assume that the program execution process can send three types of messages to the inference controller:

1. (`"sample"`, σ, α, d): The execution with id σ has reached a `sample` expression with address α and distribution d.

2. (`"observe"`, σ, α, d, c): The execution with id σ has reached an `observe` expression with address α, distribution d, and value c.

3. (`"return"`, σ, c): The execution with id σ has terminated with return value c.

Implementations of interruption and forking. To implement this interface, program execution needs to support interruption, resuming and forking. Interruption is relatively straightforward. In the case of the HOPPL, we will assume two primitives (`send` μ) and (`receive` σ). At every `sample` and `observe`, we send a message μ to the inference process, and then receive a response with process id σ. The call to `receive` then effectively pauses the execution until a response arrives. We will discuss this implementation in more detail in Section 6.4.

Support for forking can be implemented in a number of ways. In Chapter 4 we wrote evaluators that could be conditioned on a trace of random values to re-execute a program in a deterministic manner. This strategy can also be used to implement forking; we could simply re-execute the program from the start, conditioning on values of `sample` expressions that were already evaluated in the parent execution. As we noted previously, this implementation is not particularly efficient, since it requires that we re-execute the program once for every `observe` in the program, resulting a computational cost that is quadratic in the number of `observe` expressions, rather than linear.

An alternative strategy is to implement an evaluator which keeps track of the current execution state of the machine; that is, it explicitly manages all memory which the program is able to access, and keeps track of the current point of execution. To interrupt a running program, we simply store the memory state. The program can then be forked by making a (deep) copy of the saved memory back into the interpreter, and resuming execution. The difficulty with this implementation is that although the asymptotic performance may be better — since the computational cost of forking now depends on the size of the saved

memory, not the total length of program execution — there is a large fixed overhead cost in running an interpreted rather than compiled language, with its explicit memory model.

In certain cases, it is possible to leverage support for process control in the language, or even the operating system itself. An example of this is probabilistic C (Paige and Wood, 2014), which literally uses the system call `fork` to implement forking. In the case of Turing (Ge et al., 2018), the implementing language (Julia) provides coroutines, which specify computations that may be interrupted and resumed later. Turing provides a copy-on-write implementation for cloning coroutines, which is used to support forking of a process in a manner that avoids eagerly copying the memory state of the process.

As it turns out, forking becomes much more straightforward when we restrict the modeling language to prohibit mutable state. In a probabilistic variant of such a language, we have exactly two stateful operations: `sample` and `observe`. All other operations are guaranteed to have no side effects. In languages without mutable state, there is no need to copy the memory associated with a process during forking, since a variable cannot be modified in place once it has been defined.

In the HOPPL, we will implement support for interruption and forking of program executions by way of a transformation to continuation-passing style (CPS), which is a standard technique for supporting interruption of programs in purely functional languages. This transformation is used by both Anglican, where the underlying language Clojure uses data types which are by default immutable, as well as by WebPPL, where the underlying Javascript language is restricted to a purely-functional subset. Intuitively, this transformation makes every procedure call in a program happen as the last step of its caller, so that the program no longer needs to keep a call stack, which stores information about each procedure call. Such stackless programs are easy to stop and resume, because we can avoid saving and restoring their call stacks, the usual work of any scheduler in an operating system.

In the remainder of this chapter, we will first describe two source code transformations for the HOPPL. The first transformation is an addressing transformation, somewhat analogous to the one that we introduced in Section 4.2, which ensures that we can associate a unique

address with the messages that need to be sent at each `sample` and `observe` expression. The second transformation converts the HOPPL program to continuation passing style. Unlike the graph compiler in Chapter 3 and the custom evaluators in Chapter 4, both these code transformations take HOPPL programs as input and then yield output which are still HOPPL programs — they do not change the language. If the HOPPL has an existing efficient compiler, we can still use that compiler on the addressed and CPS-transformed output code. Once we have our model code transformed into this format, we show how we can implement a thin client-server layer and use this to define HOPPL variants of many of the evaluation-based inference algorithms from Chapter 4; this time, without needing to write an explicit evaluator.

6.2 Addressing Transformation

An addressing transformation modifies the source code of the program to a new program that performs the original computation whilst keeping track of an *address*: a representation of the current execution point of the program. This address should uniquely identify any `sample` and `observe` expression that can be reached in the course of an execution of a program. Since HOPPL programs can evaluate an unbounded number of `sample` and `observe` expressions, the transformation that we used to introduce addresses in Section 4.2 is not applicable here, since this transformation inlines the bodies of all function applications to create an exhaustive list of `sample` and `observe` statements, which may not be possible for HOPPL programs.

The most familiar notion of an address is a stack trace, which is encountered whenever debugging a program that has prematurely terminated: the stack trace shows not just which line of code (i.e. lexical position) is currently being executed, but also the nesting of function calls which brought us to that point of execution. In functional programming languages like the HOPPL, a stack trace effectively provides a unique identifier for the current location in the program execution. In particular, this allows us to associate a unique address with each `sample` and `observe` expresssion at run time, rather than at compile time, which we can then use in our implementations of inference methods.

The addressing transformation that we present here follows the design introduced by Wingate et al. (2011); all function calls, `sample` statements, and `observe` statements are modified to take an additional argument which provides the current address. We will use the symbol α to refer to the address argument, which must be a fresh variable that does not occur anywhere else in the program. As in previous chapters, we will describe the addressing transformation in terms of a $(e, \alpha \Downarrow_{\text{addr}} e')$ relation, which translates a HOPPL expression e and a variable α to a new expression which incorporates addresses. We additionally define a secondary $\downarrow_{\text{addr}}$ relation that operates on the top-level HOPPL program q. This secondary evaluator serves to define the top-level outer address; that is, the base of the stack trace.

Variables, procedure names, constants, and if. Since addresses track the call stack, evaluation of expressions that do not increase the depth of the call stack leave the address unaffected. Variables v and procedure names f are invariant under the addressing transformation:

$$\frac{}{v, \alpha \Downarrow_{\text{addr}} v} \qquad \frac{}{f, \alpha \Downarrow_{\text{addr}} f}$$

Evaluation of constants similarly ignores addressing. Ground types (e.g. booleans or floating point numbers) are invariant, whereas primitive procedures are transformed to accept an address argument. Since we are not able to "step in" primitive procedure calls, these calls do not increase the depth of the call stack. This means that the address argument to primitive procedure calls can be ignored.

$$\frac{c \text{ is a constant value}}{c, \alpha \Downarrow_{\text{addr}} c} \qquad \frac{c \text{ is a primitive function with } n \text{ arguments}}{c, \alpha \Downarrow_{\text{addr}} \texttt{(fn [}\alpha\ v_1\ \ldots\ v_n\texttt{] (}c\ v_1\ \ldots\ v_n\texttt{))}}$$

User-defined functions are similarly updated to take an extra address argument, which may be referenced in the function body:

$$\frac{e, \alpha \Downarrow_{\text{addr}} e'}{\texttt{(fn [}v_1\ \ldots\ v_n\texttt{] }e\texttt{)}, \alpha \Downarrow_{\text{addr}} \texttt{(fn [}\alpha\ v_1\ \ldots\ v_n\texttt{] }e'\texttt{)}}$$

Here, the translated expression e' may contain a free variable α, which (as noted above) must be a unique symbol that cannot occur anywhere in the original expression e.

Evaluation of `if` forms also does not modify the address in our implementation, which means that translation only requires translation of each of the sub-expression forms.

$$\frac{e_1, \alpha \Downarrow_{\text{addr}} e_1' \quad e_2, \alpha \Downarrow_{\text{addr}} e_2' \quad e_3, \alpha \Downarrow_{\text{addr}} e_3'}{\texttt{(if } e_1 \ e_2 \ e_3\texttt{)}, \alpha \Downarrow_{\text{addr}} \texttt{(if } e_1' \ e_2' \ e_3'\texttt{)}}$$

This is not the only choice one could make for this rule, as making an address more complex is completely fine so long as each random variable remains uniquely identifiable. If one were to desire interpretable addresses one might wish to add to the address, in a manner somewhat similar to the rules that immediately follow, a value that indicates the conditional branch. This could be useful for debugging or other forms of graphical model inspection.

Functions, sample, and observe. So far, we have simply threaded an address through the entire program, but this address has not been modified in any of the expression forms above. We increase the depth of the call stack at every function call:

$$\frac{e_i, \alpha \Downarrow_{\text{addr}} e_i' \text{ for } i = 0, \ldots, n \quad \text{Choose a fresh value } c}{\texttt{(}e_0 \ e_1 \ \ldots \ e_n\texttt{)}, \alpha \Downarrow_{\text{addr}} \texttt{(}e_0' \ \texttt{(push-addr } \alpha \ c\texttt{)} \ e_1' \ \ldots \ e_n'\texttt{)}}$$

In this rule, we begin by translating the expression e_0, which returns a transformed function e_0' that now accepts an address argument. This address argument is updated to reflect that we are now nested one level deeper in the call stack. To do so, we assume a primitive `(push-addr` α c`)` which creates a new address by combining the current address α with some unique identifier c which is generated at translation time. The translated expression will contain a new free variable α since this variable is unbound in the expression `(push-addr` α c`)`. We will bind α to a top-level address using the $\downarrow_{\text{addr}}$ relation.

If we take the stack trace metaphor literally, then we can think of α a list-like data structure, and of `push-addr` as an operation that appends a new unique identifier c to the end of this list. Alternatively, `push-addr` could perform some sort of hash on α and c to yield an address of constant size regardless of recursion depth. The identifier c

can be any, such as an integer counter for the number of function calls in the program source code, or a random hash. Alternatively, if we want addresses to be human-readable, then c can be string representation of the expression $(e_0\ e_1\ \ldots\ e_n)$ or its lexical position in the source code.

The translation rules `sample` and `observe` can be thought of as special cases of the rule for general function application.

$$\frac{e, \alpha \Downarrow_{\text{addr}} e' \quad \text{Choose a fresh value } c}{(\texttt{sample}\ e) \Downarrow_{\text{addr}} (\texttt{sample}\ (\texttt{push-addr}\ \alpha\ c)\ e')}$$

$$\frac{e_1, \alpha \Downarrow_{\text{addr}} e_1' \quad e_2, \alpha \Downarrow_{\text{addr}} e_2' \quad \text{Choose a fresh value } c}{(\texttt{observe}\ e_1\ e_2) \Downarrow_{\text{addr}} (\texttt{observe}\ (\texttt{push-addr}\ \alpha\ c)\ e_1'\ e_2')}$$

The result of this translation is that each `sample` and `observe` expression in a program will now have a unique address associated with it. These addresses are constructed dynamically at run time, but are well-defined in the sense that a `sample` or `observe` expression will have an address that is fully determined by its call stack. This means that this address scheme is valid for any HOPPL program, including programs that can instantiate an unbounded number of variables.

Top-level addresses and program translation. Translation of function calls introduces an unbound variable α into the expression. To associate a top-level address to a program execution, we define a relation $\downarrow_{\text{addr}}$ that translates the program body and wraps it in a function.

$$\frac{\text{Choose a fresh variable } \alpha \qquad e, \alpha \Downarrow_{\text{addr}} e'}{e, \alpha \downarrow_{\text{addr}} (\texttt{fn}\ [\alpha]\ e')}$$

For programs which include functions that are user-defined at the top level, this relation also inserts the additional address argument into each of the function definitions.

$$\frac{\text{Choose a fresh variable } \alpha \qquad e, \alpha \Downarrow_{\text{addr}} e' \qquad q \downarrow_{\text{addr}} q'}{(\texttt{defn}\ f\ [v_1\ \ldots\ v_n]\ e)\ q \downarrow_{\text{addr}} (\texttt{defn}\ f\ [\alpha\ v_1\ \ldots\ v_n]\ e')\ q'}$$

These rules translate our program into an address-augmented version which is still in the same language, up to the definitions of `sample` and `observe`, which are redefined to take a single additional argument.

6.3 Continuation-Passing-Style Transformation

Now that each function call in the program has been augmented with an address that tracks the location in the program execution, the next step is to transform the computation in a manner that allows us to pause and resume, potentially forking it multiple times if needed. The continuation-passing-style (CPS) transformation is a standard method from functional programming that achieves these goals.

A CPS transformation linearizes a computation into a sequence of stepwise computations. Each step in this computation evaluates an expression in the program and passes its value to a function, known as a continuation, which carries out the remainder of the computation. We can think of the continuation as a "snapshot" of an intermediate state in the computation, in the sense that it represents both the expressions that have been evaluated so far, and the expressions that need to be evaluated to complete the computation.

In the context of the messaging interface that we define in this chapter, a CPS transformation is precisely what we need to implement pausing, resuming, and forking. Once we transform a HOPPL program into CPS form, we gain access to the continuation at every `sample` and `observe` expression. This continuation can be called once to continue the computation, or multiple times to fork the computation.

There are many ways of translating a program to continuation passing style. We will here describe a relatively simple version of the transformation; for better optimized CPS transformations, see Appel (2006). We define the $\Downarrow_c$ relation

$$e, \kappa, \sigma \Downarrow_c e'.$$

Here e is a HOPPL expression, and κ is the continuation. The last e' is the result of CPS-transforming e under the continuation κ. As with other relations, we define the $\Downarrow_c$ relation by considering each expression form separately and using inference-rules notation. As with the addressing transformation, we then use this relation to define the CPS transformation of program q, which is specified by another relation

$$q, \sigma \downarrow_c q'.$$

For purposes of generality, we will incorporate an argument σ, which is not normally part of a CPS transformation. This variable serves to store mutable state, or any information that needs to be threaded through the computation. For example, if we wanted to implement support for function memoization, then σ would hold the memoization tables.

In Anglican and WebPPL, σ holds any state that needs to be tracked by the inference algorithm, and hereby plays a role analogous to that of the variable σ that we thread through our evaluators in Chapter 6. In the messaging interface that we define in this chapter, all inference state is stored by the controller process. Moreover, there is no mutable state in the HOPPL. As a result, the only state that we need to pass to the execution is the process id, which is needed to allow an execution to communicate its id when messaging the controller process. For notational simplicity, we therefore use σ to refer to both the CPS state and the process id in the sections that follow.

Variables, Procedure Names and Constants

$$\frac{}{v,\ \kappa,\ \sigma \Downarrow_c (\kappa\ \sigma\ v)} \qquad \frac{}{f,\ \kappa,\ \sigma \Downarrow_c (\kappa\ \sigma\ f)} \qquad \frac{\text{CPS}(c) = \overline{c}}{c,\ \kappa,\ \sigma \Downarrow_c (\kappa\ \sigma\ \overline{c})}$$

When e is a variable v or a procedure name f, the CPS transform simply calls the continuation on the value of the variable. The same is true for constant values c of a ground type, such as boolean values, integers and real numbers. The case that requires special treatment is that of constant primitive functions c, which need to be transformed to accept a continuation and a state as arguments. We do so using a subroutine $\text{CPS}(c)$, which leaves constants of ground type invariant and transforms this primitive functions into a procedure

$$\overline{c} = \text{CPS}(c) = (\texttt{fn}\ [v_1\ \ v_2\ \ \kappa\ \ \sigma]\ \ (\kappa\ \ \sigma\ \ (c\ \ v_1\ \ v_2))).$$

The transformed procedure accepts κ and σ as additional arguments. When called, it evaluates the return value $(c\ v_1\ v_2)$ and passes this value to the continuation κ, together with the state σ. For all the usual operators c, such as `+` and `*`, we represent CPS variants with $\overline{c}$, such as $\overline{+}$ and $\overline{*}$.

If Forms. Evaluation of if forms involves two steps. First we evaluate the predicate, and then we either evaluate the consequent or the alternative branch. When transforming an if form to CPS, we turn this order "inside out", which is to say that we first transform the consequent and alternative branches, and then use the transformed branches to define a transformed if expression that evaluates the predicate and selects the correct branch

$$\frac{\begin{array}{ll} e_2,\, \kappa,\, \sigma \Downarrow_c e_2' & e_3,\, \kappa,\, \sigma \Downarrow_c e_3' \\ \text{Choose a fresh variable } v & e_1, \texttt{(fn [}\sigma\ v\texttt{] (if }v\ e_2'\ e_3'\texttt{))},\, \sigma \Downarrow_c e' \end{array}}{\texttt{(if }e_1\ e_2\ e_3\texttt{)},\, \kappa,\, \sigma \Downarrow_c e'}$$

The inference rule begins by transforming both branches e_1 and e_2 under the continuation κ. This yields expressions e_1' and e_2' that pass the value of each branch to the continuation. Given these expressions, we then define a new continuation `(fn [`σ v`] (if` v e_2' e_3'`))` which accepts the value of a predicate and selects the appropriate branch. We then use this continuation to transform the expression for the predicate e_1.

This inference rule illustrates one of the basic mechanics of the CPS transformation, which is to create continuations *dynamically* during evaluation. To see what we mean by this, let us consider the expression `(if true 1 0)`, which translates to

```
((fn [σ v]
   (if v
    (κ σ 1)
    (κ σ 0))) σ true)
```

The CPS transformation accepts a HOPPL program and two variables κ and σ, and returns a HOPPL program in which κ and σ are free variables. When we evaluate this program, we pass the state and the value of the predicate to a newly created anonymous procedure that calls the continuation κ on the value of the appropriate branch. The important point is that the CPS transformation creates the *source code* for a procedure, not a procedure itself. In other words, the top-level continuation is not created until we evaluate the transformed program. This property will prove essential when we define the CPS tranformation for procedure calls.

Procedure Definition To tranform an anonymous procedure, we need to transform the procedure to accept continuation and state arguments, and transform the procedure body to pass the return value to the continuation. We do so using the following rule

$$\frac{\text{Choose a fresh variable } \kappa' \qquad e,\, \kappa',\, \sigma \Downarrow_c e'}{\texttt{(fn [}v_1\ \ldots\ v_n\texttt{] } e\texttt{)},\, \kappa,\, \sigma \Downarrow_c \texttt{(}\kappa\ \sigma\ \texttt{(fn [}v_1\ \ldots\ v_n\ \kappa'\ \sigma\texttt{] } e'\texttt{))}}$$

We introduce a new continuation variable κ', and transform the procedure body e recursively under this new κ'. Then, we use the transformed body e' to define a new procedure, which is passed to the original continuation κ. Note that the original continuation expects a procedure, not the return value of a procedure. For instance,

$$\texttt{(fn [] 1)},\, \kappa,\, \sigma \Downarrow_c \texttt{(}\kappa\ \sigma\ \texttt{(fn [}\kappa'\ \sigma\texttt{] (}\kappa'\ \sigma\ 1\texttt{)))}$$

The continuation parameter κ' takes the result of the original procedure 1 while the current continuation κ takes the CPS-transformed version of the procedure itself.

Procedure Call To evaluate a procedure call, we normally evaluate each of the arguments, bind the argument values to argument variables and then evaluate the body of the procedure. When performing the CPS transformation we once again reverse this order

$$\frac{\begin{array}{l}\text{Choose fresh variables } v_0, \ldots, v_n \\ e_n,\, \texttt{(fn [}\sigma\ v_n\texttt{] (}v_0\ v_1 \ldots v_n\ \kappa\ \sigma\texttt{))},\, \sigma \Downarrow_c e'_n \\ e_i,\, \texttt{(fn [}\sigma\ v_i\texttt{] } e'_{i+1}\texttt{)},\, \sigma \Downarrow_c e'_i \quad \text{for } i = (n-1), \ldots, 0\end{array}}{\texttt{(}e_0\ e_1 \ldots e_n\texttt{)},\, \kappa,\, \sigma \Downarrow_c e'_0}$$

We begin by constructing a continuation `(fn [`$\sigma\ v_n$`] (`$v_0\ v_1\ \ldots\ v_n\ \kappa\ \sigma$`))` that calls a transformed procedure v_0 with continuation κ and state σ. Note that this continuation is "incomplete", in the sense that $v_0, \ldots, v_{n-1}$ are unbound variables that are not passed to the continuation. In order to bind these variables, we transform the expression, and put the result e'_n inside another expression that creates the continuation for variable v_{n-1}. We continue this transformation-then-nesting recursively until we have defined source code that creates a continuation

`(fn [`σ v_0`] ...)`, which accepts the transformed procedure as an argument. It is here where the ability to create continuations dynamically, which we highlighted in our earlier discussion of if expressions, becomes essential.

To better understand what is going on, let us consider the HOPPL expression `(+ 1 2)`. Based on the rules we defined above, we know that 1 and 2 are invariant and that the primitive function `+` will be transformed to a procedure $\bar{+}$ that accepts a continuation and a state as additional arguments. The CPS tranform of `(+ 1 2)` is

```
((fn [σ v0]
   ((fn [σ v1]
      ((fn [σ v2]
         (v0 v1 v2 κ σ)
       ) σ 2)
    ) σ 1)
 ) σ +̄)
```

This expression may on first inspection not be the easiest to read. It is equivalent to the following nested `let` expressions, which are much easier understand (you can check this by desugaring)

```
(let [σ σ
      v0 +̄]
  (let [σ σ
        v1 1]
    (let [σ σ
          v2 2]
      (v0 v1 v2 κ σ))))
```

In order to highlight where continuations are defined, we can equivalently rewrite the expression by assigning each anonymous procedure to a variable name

```
(let [κ0 (fn [σ v0]
           (let [κ1 (fn [σ v1]
                      (let [κ2 (fn [σ v2]
                                 (v0 v1 v2 κ σ))]
                        (κ2 σ 2)))]
             (κ1 σ 1)))
  (κ0 σ +̄))
```

In this form of the expression we see clearly that we define 3 continuations at runtime in a nested manner. The outer continuation κ_0 accepts σ and $\dot{+}$. This continuation κ_0 in turn defines a continuation κ_1, which accepts σ and the first argument. The continuation κ_1 defines a third continuation κ_2, which accepts the σ and the second argument, and calls the CPS-transformed function.

This example illustrates how continuations record both the remainder of the computation and variables that have been defined thus far. κ_2 references v_0 and v_1, which are in scope because κ_2 is defined inside a call to κ_1 (where v_1 is defined), which is in turn defined in a call to κ_0 (where v_0 is defined). In functional programming terms, we say that the continuation κ_2 *closes* over the variables v_0 and v_1. If we want to interrupt the computation, then we can return a tuple $[\kappa_2\ \sigma\ v_2]$, rather than evaluating the continuation call $(\kappa_2\ \sigma\ v_2)$. The continuation κ_2 then effectively contains a snapshot of the variables v_0 and v_1.

Observe and Sample

$$\frac{\begin{array}{l}\text{Choose fresh variables } v_{\text{addr}}, v_1, v_2 \\ e_2, (\texttt{fn}\ [\sigma\ v_2]\ (\overline{\texttt{observe}}\ v_{\text{addr}}\ v_1\ v_2\ \kappa\ \sigma)), \sigma \Downarrow_c e_2' \\ e_1, (\texttt{fn}\ [\sigma\ v_1]\ e_2'), \sigma \Downarrow e_1' \\ e_{\text{addr}}, (\texttt{fn}\ [\sigma\ v_{\text{addr}}]\ e_1'), \sigma \Downarrow e_{\text{addr}}'\end{array}}{(\texttt{observe}\ e_{\text{addr}}\ e_1\ e_2), \kappa, \sigma \Downarrow_c e_{\text{addr}}'}$$

$$\frac{\begin{array}{l}\text{Choose a fresh variable } v_{\text{addr}}, v \\ e, (\texttt{fn}\ [\sigma\ v]\ (\overline{\texttt{sample}}\ v_{\text{addr}}\ v\ \kappa\ \sigma)), \sigma \Downarrow_c e' \\ e_{\text{addr}}, (\texttt{fn}\ [\sigma\ v_{\text{addr}}]\ e'), \sigma \Downarrow e_{\text{addr}}'\end{array}}{(\texttt{sample}\ e_{\text{addr}}\ e), \kappa, \sigma \Downarrow_c e_{\text{addr}}'}$$

These two rules are unique for the CPS transform of probabilistic programming languages. They replace `observe` and `sample` operators with their CPS equivalents $\overline{\texttt{observe}}$ and $\overline{\texttt{sample}}$, which take two additional parameters κ for the current continuation and σ for the current state. In this translation we assume that an addressing tranformation has already been applied to add an address e_{addr} as an argument to sample and observe.

Implementing `observe` and `sample` corresponds to writing an inference algorithm for probabilistic programs. When a program execution hits one of `observe` and `sample` expressions, it suspends the execution, and returns its control to an inference algorithm with information about address α, parameters, current continuation κ and current state σ. In the next section we will discuss how we can implement these operations.

Program translation The CPS tranformation of expression defined so far enables the translation of programs. It is shown in the following inference rules in terms of the relation $q \downarrow_c q'$, which means that the CPS transformation of the program q is q':

$$\frac{\text{Choose fresh variables } v, \sigma, \sigma' \qquad e, \texttt{(fn [}\sigma\ v\texttt{] (return }v\ \sigma\texttt{))}, \sigma' \Downarrow_c e'}{\texttt{(fn [}\alpha\texttt{] }e\texttt{)} \downarrow_c \texttt{(fn [}\alpha\ \sigma\texttt{] }e'\texttt{)}}$$

$$\frac{\text{Choose fresh variables } \kappa, \sigma \qquad e, \kappa, \sigma \Downarrow_c e' \qquad q \downarrow_c q'}{\texttt{(defn }f\ \texttt{[}v_1\ \ldots\ v_n\texttt{] }e\texttt{)}\ q \downarrow_c \texttt{(defn }f\ \texttt{[}v_1\ \ldots\ v_n\ \kappa\ \sigma\texttt{] }e'\texttt{)}\ q'}$$

The main difference between the CPS transformation of programs and that of expressions is the use of the default continuation in the first rule, which returns its inputs v, σ by calling the `return` function.

6.4 Message Interface Implementation

Now that we have inserted addresses into our programs, and transformed them into CPS, we are ready to perform inference. To do so, we will implement the messaging interface that we outlined in Section 6.1. In this interface, an inference controller process starts copies of the probabilistic program, which are interrupted at every `sample` and `observe` expression. Upon interruption, each program execution sends a message to the controller, which then carries out any inference operations that need to be performed. These operations can include sampling a new value, reusing a stored value, computing the log probabilies, and resampling program executions. After carrying out these operations, the controller sensds messages to the program executions to continue or fork the computation.

As we noted previously, the messaging interface creates an abstraction boundary between the controller process and the execution process.

As long as each process can send and receive messages, it need not have knowledge of the internals of the other process. This means that the two processes can be implemented in different languages if need be, and can even be executed on different machines.

In order to clearly highlight this separation between model execution and inference, we will implement our messaging protocol using a client-server architecture. The server carries out program executions, and the client is the inference process, which sends requests to the server to start, continue, and fork processes. We assume the existence of an interface that supports centrally-coordinated asynchronous message passing in the form of request and response. Common networking packages such as ZeroMQ (Powell, 2015) provide abstractions for these patterns. We will also assume a mechanism for defining and serializing messages, e.g. protobuf (Google, 2018). At an operational level, the most important requirement in this interface is that we are able to serialize and deserialize distribution objects, which effectively means that the inference process and the modeling language must implement the same set of distribution primitives.

Messages in the Inference Controller. In the language that implements the inference controller (i.e. the client), we assume the existence of a SEND method with 4 argument signatures, which we previously introduced in Section 6.1

1. SEND(`"start"`, σ): Start a new process with id σ.
2. SEND(`"continue"`, σ, c): Continue process σ with argument c.
3. SEND(`"fork"`, σ, σ', c): Fork process σ into a new process with id σ' and continue execution with argument c.
4. SEND(`"kill"`, σ): Halt execution for process σ.

In addition, we assume a method RECEIVE, which listens for responses from the execution server.

Messages in the Execution Server. The execution server, which runs CPS-transformed HOPPL programs, can itself entirely be implemented

in the HOPPL. The execution server must be able to receive requests from the inference controller and return responses. We will assume that primitive functions `receive` and `send` exist for this purpose. The 3 repsonses that we defined in Section 6.1 were

1. `(send "sample"` σ α d`)`: The process σ has arrived at a `sample` expression with address α and distribution d.
2. `(send "observe"` σ α d c`)`: The process σ has arrived at an `observe` expression with address α, distribution d, and value c.
3. `(send "return"` σ c`)`: Process σ has terminated with value c.

To implement this messaging architecture, we need to change the behavior of `sample` and `observe`. Remember that in the CPS transformation, we make use of CPS analogues $\overline{\texttt{sample}}$ and $\overline{\texttt{observe}}$. To interrupt the computation, we will provide an implementation that returns a tuple, rather than calling the continuation. Similarly, we will also implement `return` to yield a tuple containing the state (i.e. the process id) and the return value

```
(defn sample [α d κ σ]
  ["sample" α d κ σ])

(defn observe [α d c κ σ]
  ["observe" α d c κ σ])

(defn return [c σ]
  ["return" c σ])
```

Now we will assume that execution server reads in some program source from a file, parses the source, applies the address transformation and the cps transformation, and then evaluates the source code to create the program

```
(def program
  (eval (cps-transform
          (address-transform
            (parse "program.hoppl")))))
```

Now that this program is defined, we will implement a request handler that accepts a process table and an incoming message.

```
(defn handler [processes message]
  (let [request-type (first message)]
    (case request-type
      "start" (let [[σ] (rest message)
                    output (program default-addr σ)]
                (respond processes output))
      "continue" (let [[σ c] (rest message)
                       κ (get processes σ)
                       output (κ σ c)]
                   (respond processes output))
      "fork" (let [[σ σ′ c] (rest message)
                   κ (get processes σ)
                   output (κ σ′ c)]
               (respond (put processes σ′ κ) output))
      "kill" (let [[σ] (rest message)]
               (remove processes σ)))))
```

To process a message, the handler dispatches on the request type. For `"start"`, it starts a new process by calling the compiled `program`. For `"kill"`, it simply deletes the continuation from the process table. For `"continue"` and `"fork"`, it retrieves one of continuations from the process table and continues executions. For each request type the program/continuation will return a tuple that is the output from a call to `sample`, `observe`, or `return`. The handler then calls a second function

```
(defn respond [processes output]
  (let [response-type (first output)]
    (case response-type
      "sample" (let [[α d κ σ] (rest output)]
                 (send "sample" σ α d)
                 (put processes σ κ))
      "observe" (let [[α d c κ σ] (rest output)]
                  (send "observe" σ α d c)
                  (put processes σ κ))
      "return" (let [[c σ] (rest output)]
                 (send "return" σ c)
                 (remove processes σ)))))
```

This function dispatches on the response type, sends the appropriate message, and returns a process table that is updated with the continuation if needed. Now that we have all the machinery in place, we can define the execution loop as a recursive function

```
(defn execution-loop [processes]
  (let [processes (handler processes (receive))]
    (execution-loop processes)))
```

6.5 Likelihood Weighting

Setting up the capability to run, interrupt, resume, and fork HOPPL programs, required a fair amount of work. However, the payoff is that we have now implemented an interface which we can use to easily write many different inference algorithms. We illustrate this benefit with a series of inference algorithms, starting with likelihood weighting.

Algorithm 16 shows an explicit definition of the inference controller for likelihood weighting that we described high-level at the beginning of this chapter. In this implementation, we launch L executions of the program. For each execution, we define a unique process id σ using a function NEWID, and intialize the log weight to $\log W_\sigma \leftarrow 0$. We then repeatedly listen for responses. At `"sample"` interrupts, we draw a sample from the prior and continue execution. At `"observe"` interrupts, we update the log weight of the process with id σ and continue execution with the observed value. When an execution completes with a `"return"` interrupt, we output the return value c and the accumulated log weight $\log W_\sigma$ by calling a procedure OUTPUT, which depending on our needs could either save to disk, print to standard output, or store the sample in some database.

Note that this controller process is fully asynchronous. This means that if we were to implement the function `execution-loop` to be multi-threaded, then we can trivially exploit the embarrassingly parallel nature of the likelihood weighting algorithm to speed up execution.

6.6 Metropolis-Hastings

We next implement a single-site Metropolis-Hastings algorithm using this interface. The full algorithm, given in Algorithm 17, has an overall structure which closely follows that of the evaluation-based algorithm for the first-order language given in Section 4.2.

Algorithm 16 Inference controller for Likelihood Weighting

```
1:  repeat
2:      for ℓ = 1, ..., L do                    ▷ Start L copies of the program
3:          σ ← NEWID()
4:          log W_σ ← 0
5:          SEND("start", σ)
6:      l ← 0
7:      while l < L do
8:          μ ← RECEIVE()
9:          switch μ do
10:             case ("sample", σ, α, d)
11:                 x ← SAMPLE(d)
12:                 SEND("continue", σ, x)
13:             case ("observe", σ, α, d, c)
14:                 log W_σ ← log W_σ + LOG-PROB(d, c)
15:                 SEND("continue", σ, c)
16:             case ("return", σ, c)
17:                 l ← l + 1
18:                 OUTPUT(c, log W_σ)
19: until forever
```

The primary difference between this algorithm and that of Section 4.2 is due to the dynamic addressing. In the FOPPL, each function is guaranteed to be called a finite number of times. This means that we can unroll the entire computation, inlining functions, and literally annotate every `sample` and `observe` that can ever be evaluated with a unique identifier. In the HOPPL, programs can have an unbounded number of addresses that can be encountered, which necessitates the addressing transformation that we defined in Section 6.2.

As in the evaluator-based implementation in Section 4.2, the inference controller maintains a trace $\mathcal{X}$ for the current sample and a trace $\mathcal{X}'$ for the current proposal, which track the values for `sample` form that is evaluated. We also maintain maps $\log \mathcal{P}$ and $\log \mathcal{P}'$ that hold the log probability for each `sample` and `observe` form. The acceptance ratio is

Algorithm 17 Inference Controller for Metropolis-Hastings

$\ell \leftarrow 0$ ▷ Iteration counter
$r, \mathcal{X}, \log \mathcal{P} \leftarrow$ `nil`, [], [] ▷ Current trace
$\mathcal{X}', \log \mathcal{P}'$ ▷ Proposal trace
function ACCEPT($\beta, \mathcal{X}', \mathcal{X}, \log \mathcal{P}', \log \mathcal{P}$)
    ... ▷ as in Algorithm 9
repeat
    $\ell \leftarrow \ell + 1$
    $\beta \sim$ UNIFORM(dom($\mathcal{X}$)) ▷ Choose a single address to modify
    SEND("start", NEWID())
    repeat
        $\mu \leftarrow$ RECEIVE()
        switch μ **do**
            case ("sample", σ, α, d)
                if $\alpha \in \text{dom}(\mathcal{X}) \setminus \{\beta\}$ **then**
                    $\mathcal{X}'(\alpha) \leftarrow \mathcal{X}(\alpha)$
                else
                    $\mathcal{X}'(\alpha) \leftarrow$ SAMPLE(d)
                $\log \mathcal{P}'(\alpha) \leftarrow$ LOG-PROB($d, \mathcal{X}'(\alpha)$)
                SEND("continue", $\sigma, \mathcal{X}'(\alpha)$)
            case ("observe", σ, α, d, c)
                $\log \mathcal{P}'(\alpha) \leftarrow$ LOG-PROB(d, c)
                SEND("continue", σ, c)
            case ("return", σ, c)
                if $\ell = 1$ **then**
                    $u \leftarrow 1$ ▷ Always accept first iteration
                else
                    $u \sim$ UNIFORM-CONTINUOUS$(0, 1)$
                if $u <$ ACCEPT($\beta, \mathcal{X}', \mathcal{X}, \log \mathcal{P}', \log \mathcal{P}$) **then**
                    $r, \mathcal{X}, \log \mathcal{P} \leftarrow c, \mathcal{X}', \log \mathcal{P}'$
                OUTPUT($r, 0.0$) ▷ MH samples are unweighted
                break
    until forever
until forever

calculated in exactly the same way as in Algorithm 9.

As with the implementation in Chapter 4, an inefficiency in this algorithm is that we need to re-run the entire program when proposing a change to a single random choice. The graph-based MH sampler from Section 3.3, in contrast, was able to avoid re-evaluation of expressions that do not reference the updated random variable. Recent work has explored a number of ways to avoid this overhead. In a CPS-based implementation, we store the continuation function at each address α. When proposing an update to variable α, we know that none of the steps in the computation that precede α can change. This means we can skip re-execution of this part of the program by calling the continuation at α. The implementation in Anglican makes use of this optimization (Tolpin et al., 2016). A second optimization is callsite caching (Ritchie et al., 2016a), which memoizes return values of functions in a manner that accounts for both the argument values and the environment that a function closes over, allowing re-execution in the proposal to be skipped when the arguments and environment are identical.

6.7 Sequential Monte Carlo

While the previous two algorithms were very similar to those presented for the FOPPL, running SMC in the HOPPL context is slightly more complex, though doing so opens up significant opportunities for scaling and efficiency of inference. We will need to take advantage of the `"fork"` message, and due to the (potentially) asynchronous nature in which the HOPPL code is executed, we will need to be careful in tracking execution ids of particular running copies of the model program.

An inference controller for SMC is shown in Algorithm 18. As in the implementation of likelihood weighting, we start L executions in parallel, and then listen for responses. When an execution reaches a sample interrupt, we simply sample from the prior and continue execution. When one of the executions reaches an observe, we will need to perform a resampling step, but we cannot do so until all executions have arrived at the same observe. For this reason we store the address of the current observe in a variable α_{cur}, and use a particle counter l to track how many of executions have arrived at the current observe.

Algorithm 18 Inference Controller for SMC

repeat
  $\log \hat{Z} \leftarrow 0.0$
  for l **in** $1, \ldots, L$ **do** ▷ Start L copies of the program
    SEND("start", NEWID())
  $l \leftarrow 0$ ▷ Particle counter
  while $l < L$ **do**
    $\mu \leftarrow$ RECEIVE()
    switch μ **do**
      case ("sample", σ, α, d)
        $x \leftarrow$ SAMPLE(d)
        SEND("continue", σ, x)
      case ("observe", σ, α, d, c)
        $l \leftarrow l + 1$
        $\sigma_l, \log W_l \leftarrow \sigma,$ LOG-PROB(d, c)
        if $l = 1$ **then**
          $\alpha_{\text{cur}} \leftarrow \alpha$ ▷ Set address for current `observe`
        if $l > 1$ **then**
          assert $\alpha_{\text{cur}} = \alpha$ ▷ Ensure same address
        if $l = L$ **then**
          $o_1, \ldots, o_L \leftarrow$ RESAMPLE($W_1, \ldots, W_L$)
          $\log \hat{Z} \leftarrow \log \hat{Z} + \log \frac{1}{L} \sum_{l=1}^{L} W_l$
          for $l' = 1, \ldots, L$ **do**
            for $i = 1, \ldots, o_l$ **do**
              SEND("fork", $\sigma_{l'}$, NEWID(), c)
            SEND("kill", $\sigma_{l'}$)
          $l \leftarrow 0$ ▷ Reset particle counter
      case ("return", σ, c)
        $l \leftarrow l + 1$
        OUTPUT($c, \log \hat{Z}$)
until forever

For each execution, we store the process id σ_l and the incremental log weight $\log W_l$ at the observe. Note that, since the order in which messages are received from the running programs is nondeterministic, the individual indices $1, \ldots, L$ for different particles do not hold any particular meaning from one observe to the next.

An important consideration in this algorithm, which also applies to the implementation in Section 4.3, is that resampling in SMC must happen at some sequence of interrupts that are reached in every execution of a program. In Section 4.3 we performed resampling at a user-specified sequence of breakpoint addresses $y_1, \ldots, y_N$. Here, we simply assume that the HOPPL program will always evaluate the same sequence of observes in the same order, and throw an error when this is not the case. A limitation of this strategy is that it cannot handle `observe` forms that appear conditionally; e.g. `observe` forms that appear inside branches of `if` forms. If we needed to support SMC inference for such programs, then we could carry resampling at a subset of `observe` forms which are guaranteed to appear in the same order in every execution of the program. This could be handled by manually augmenting the `observe` form (and the `"observe"` message) to annotate which observes should be treated as triggering a resample. Alternatively, one could implement an addressing scheme in which addresses are ordered, which is to say that we can define a comparison $\alpha < \alpha'$ that indicates whether an interrupt at address α precedes an interrupt at address at α' during evaluation. When addresses are ordered, we can implement a variety of resampling policies that generalize from SMC (Whiteley et al., 2016), such as policies that resample the subset of executions at an address α once all remaining executions have reached interrupts with addresses $\alpha' > \alpha$.

This SMC algorithm can additionally be used as a building-block for particle MCMC algorithms (Andrieu et al., 2010), which uses a single SMC sweep of L particles as a block proposal in a Metropolis-Hastings algorithm. Particle MCMC algorithms for HOPPL languages are discussed in detail in Wood et al. (2014b) and Paige and Wood (2014).

7

Advanced Topics

So far in this tutorial we have looked at how to design first-order and higher-order probabilistic programming languages, and provided a blueprint for implementation of automatic inference in each. In this chapter, we change direction, and describe some recent advances around current questions of research interest in the field at the time of writing. We look in a few different directions, beginning with two ways in which probabilistic programming can benefit from integration with deep learning frameworks, and then move on to looking at challenges to implementing Hamiltonian Monte Carlo and variational inference within the HOPPL and implementing expressive models by recursively nesting probabilistic programs. We conclude with an introduction to formal semantics for probabilistic programming languages.

7.1 Inference Compilation

Most of the inference methods described in the previous chapters have been specified assuming we are performing inference for a given model exactly *once*, on a single fixed dataset. In statistics it is usually the case that there is a (single, fixed) dataset which one would like to understand by employing a model to test hypotheses about its characteristics.

In many real-world applications in engineering, finance, science, and artificial intelligence, we would like to perform inference in the same model many times, whenever new data are collected. There is often a model in which, if it were possible, repeated, rapid inference given *new* data each time is, instead, of interest. Consider, for instance, stochastic simulators of engines, or of factories: in these, diagnostic queries could easily be framed as inference in the simulator given a sufficiently rapid inference procedure. In finance, rapid inference in more powerful, richly structured models than can be inverted analytically could lead to high-speed trading decision engines with higher performance. Science is no different in its use of simulators and the value that could be derived from rapid Bayesian inversion; take, for example, the software simulators that describe the standard model of physics and particle detector responses to see how useful efficient repeated inference could be, even in a fixed model. Artificial intelligence requires repeated, rapid inference; in particular for agents to understand the world around them that they only partially observe.

In all the situations described, the model — both structure and parameters — is fixed, and rapid repeated inference is desired. This setting has been described as *amortized* inference (Gershman and Goodman, 2014), due to the tradeoff made between up-front and inference-time computation. Specific implementations of this idea have appeared in the probabilistic programming literature (Kulkarni et al., 2015a; Ritchie et al., 2016b), where program-specific neural networks were trained to guide forward inference.

Le et al. (2017a) and Paige and Wood (2016) introduce a general approach we will call "inference compilation", for amortized inference in probabilistic programming systems. This approach is diagrammed in Figure 7.1, where the denotation of the joint distribution provided by a probabilistic program is leveraged in two ways: both to obtain via source code analysis some parts of the structure of a bottom-up inference neural network, and to generate synthetic training data via ancestral sampling of the joint distribution. A network trained with these synthetic data is later used repeatedly to perform inference with real data. Paige and Wood (2016) introduced a strategy for learning inverse programs for models with finite parameter dimension, i.e. models denotable in

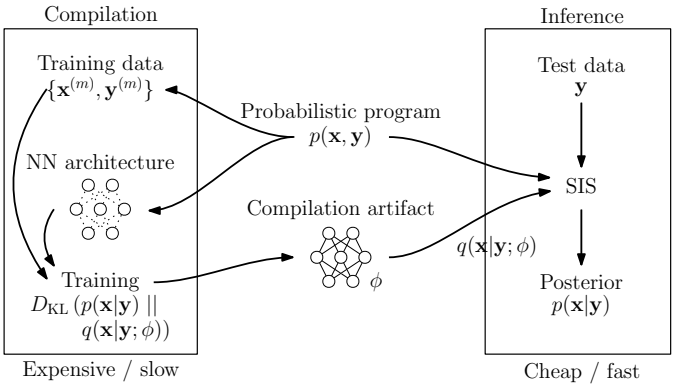

Figure 7.1: An outline of an approach to inference compilation for amortized inference for probabilistic programs. Re-used with permission from Le et al. (2017a).

the FOPPL. Le et al. (2017a) uses the same training objective, which we will describe next, but shows how to construct a neural inference compilation artifact compatible with HOPPL program inference.

We will follow Le et al. (2017a) and describe, briefly, the idea for HOPPLs. Recall Section 4.1.1 in which importance sampling in its general form was discussed. Immediately after the presentation of importance sampling a choice of the proposal distribution was made, namely, the prior, and this choice was fixed for the remainder leading to discussion of likelihood weighting rather than importance sampling throughout. In particular, in Chapters 4 and 6 where evaluation-based inference was introduced, in both likelihood weighting and SMC, the weights computed and combined were always simply the observe log likelihoods owing to the choice of prior as proposal.

This choice is easy — propose by simply running the program forward — and always available, but is not necessarily a good choice. In particular, when observations are highly informative, proposing from the prior distribution may be arbitrarily statistically inefficient (Del Moral and Murray, 2015). To motivate this approach to inference compilation, we note that this choice is not the only possibility, and if a good proposal were available then the quality of inference performed could be substantially improved in terms of number of effective samples per

unit computation.

Consider, for the moment, what would happen if someone provided you with a "good" proposal for every address α

$$q_\alpha(x_\alpha|X_{n-1}, Y) \tag{7.1}$$

noting that this is not the incremental prior and that it in general depends on all observations Y. Here we assume that the n-th sample is drawn at α for some n, and write X_{n-1} for the samples drawn beforehand. The likelihood-weighting evaluators can be transformed into importance sampling evaluators by changing the sample method implementations to draw x_α according to Equation (7.1) instead of $p_\alpha(x_\alpha|X_{n-1})$. The rules for `sample` would then need to generate a weight too (as opposed to generating such side-effects at the evaluation of only `observe` statements, not `sample` statements). This weight would be

$$W_\alpha^\ell = \frac{p_\alpha(x_\alpha|X_{n-1})}{q_\alpha(x_\alpha|X_{n-1}, Y)} \tag{7.2}$$

leading to, for importance sampling rather than likelihood weighting, a total unnormalized weight per trace ℓ of

$$W^\ell = p(Y|X) \prod_{\alpha \in \mathrm{dom}(X)} \frac{p_\alpha(x_\alpha|X_{n-1})}{q_\alpha(x_\alpha|X_{n-1}, Y)}. \tag{7.3}$$

The problem then becomes: what is a good proposal, and how do we find it? There is a body of literature on adaptive importance sampling and optimal proposals for sequential Monte Carlo that addresses this question. Doucet et al. (2000) and Cornebise et al. (2008) show that optimal proposal distributions are in general intractable. So, in practice, good proposal distributions are either hand-designed prior to sampling or are approximated using some kind of online estimation procedure to approximate the optimal proposal during inference (as in e.g. Van Der Merwe et al. (2000) or Cornebise et al. (2014) for state-space models).

Inference compilation trains a "good" proposal distribution at compile time — that is, before the observation Y is given — by minimizing the Kullback–Leibler divergence between the target posterior and the proposal distribution $D_{\mathrm{KL}}\left(p(X|Y) \,||\, q(X|Y;\phi)\right)$ with respect to the

parameters ϕ of the proposal distribution. The aim here is finding a proposal that is good not just for one observation Y but instead is good in expectation over the entire distribution of Y. To achieve this, inference compilation minimizes the expected KL under the distribution $p(Y)$

$$\begin{aligned}\mathcal{L}(\phi) &:= \mathbb{E}_{p(Y)}\left[D_{\mathrm{KL}}\left(p(X|Y)\,||\,q(X|Y;\phi)\right)\right] && (7.4)\\ &= \int_Y p(Y)\int_X p(X|Y)\log\frac{p(X|Y)}{q(X|Y;\phi)}\,\mathrm{d}X\,\mathrm{d}Y\\ &= \mathbb{E}_{p(X,Y)}\left[-\log q(X|Y;\phi)\right] + \text{const.} && (7.5)\end{aligned}$$

Conveniently, again, the probabilistic program denotes the joint distribution (simply de-sugar all observe statements to sample statements, e.g. `(observe d c)` becomes `(sample d)`) which means that an unbounded number of importance-weighted samples can be drawn from the joint distribution to compute Monte Carlo estimates of the expectation.

What remains is to choose a specific form for the proposal distribution to be learned. Consider a form like

$$q(X|Y;\phi) = \prod_{\alpha\in\mathrm{dom}(X)} q_\alpha(x_\alpha|\eta(\alpha, X_{n-1}, Y, \phi)) \tag{7.6}$$

and let η be a differentiable function parameterized by ϕ that takes the address of the next sample to be drawn, the trace sample values to that point, and the values of all of the observations, and produces parameters for a proposal distribution for that address. The values of X and Y are given and we choose q_α such that it will be differentiable with respect to its own parameters; if these parameters are computed by a differentiable function η, then learning of ϕ using Equation (7.5) can be done using standard gradient-based optimization techniques.

Now the question that remains is what form does the function η take. In, e.g. (Le et al., 2017a), a polymorphic neural network with a program specific encoder network and a stacked long-short-term-memory recurrent neural network backbone was used. In (Paige and Wood, 2016) a masked auto-regression density estimator was used. In short, and in particular in the HOPPL, whatever neural network architecture is used, it must be able to map to a variable number of outputs, and

incorporate sampled values in a sequential manner, concurrent with the running of the inference engine. It should be noted also that, once trained, the inference compilation network is entirely compatible with the client/server inference architecture explained in Chapter 6.

Such inference compilation has been shown to dramatically speed inference in the underlying models in a number of cases, bringing probabilistic programming ever closer to real practicality. There remain a number of interesting research problems currently under consideration here too. Chief amongst them is: is there a way to structure the bottom-up program advantageously and automatically given the top-down program or vice versa? Important initial work has been done on this problem (Webb et al., 2017; Paige and Wood, 2016; Stuhlmüller et al., 2013) but much remains. Were there to be good, broadly applicable algorithms, they would do much to close the emerging gap between the broadly independent research disciplines of discriminative learning and generative modeling.

7.2 Model Learning

It might seem like this tutorial has implicitly advocated for unsupervised model-based inference. One thing machine learning and deep learning have reminded us over and over again is that writing good, detailed, descriptive, and useful models is itself very hard. Time and again, data-driven discriminative techniques have been shown to be generally superior at solving specific posed tasks, with the significant caveat that large amounts of labeled training data are typically required. Taking inspiration from this encourages us to think about how we can use data-driven approaches to *learn* generative models, either in part or in whole, rather than writing them entirely by hand. In this section we look at how probabilistic programming techniques can generalize such that "bottom-up" deep learning and "top-down" probabilistic modeling can work together in concert.

Top-down model specification is equivalent to the act of writing a probabilistic program: top-down means, roughly, starting with latent variables and performing a forward computation to arrive at an observation. Our journey from FOPPL to HOPPL merely increased our

flexibility in specifying and structuring this computation.

In contrast, bottom-up computation starts at the observations and computes the value or parameters of a distribution over a latent quantity (such as a probability vector over possible output labels). Such bottom-up computation traditionally used compositions of hand-engineered feature extraction and combination algorithms but as of now is firmly the domain of deep neural networks. Deep networks are parameterized, structured programs that feed forward from a value in one domain to a value in another; the case of interest here being transformations from the space of observation to the parameters for the latents. Neural network programs only roughly structure a computation (for instance specifying that it uses convolutions) but do not usually fully specify the specific computation to be performed until being trained using input/output supervision data to perform a specific regression, classification, or inference computation task. Their observed efficacy is remarkable, particularly when they can be viewed as partially specified programs whose refinement or induction from input/output examples is computed by stochastic gradient descent.

Consider what you have learned about how probabilistic programs are evaluated. Such evaluations require running one of the generic inference algorithms discussed. Each inference program, an evaluator, was, throughout this tutorial, taken to be fixed, i.e. fully parameterized. Furthermore, also throughout this tutorial, the probabilistic program itself – the model – was also assumed to be fixed in both structure and parameterization.

7.2.1 Helmholtz machines and variational autoencoders

What inference compilation does not do is to adapt the model; it assumes that the given model $p(X, Y)$ is fixed and correct. It can make inference in models fast and accurate, but writing accurate generative models is an extremely difficult task as well. Perhaps more to the point, manually writing an efficient, fully specified generative model that is accurate all the way down to naturally occurring observable data is fiendishly difficult; perhaps impossible, particularly when data are raw signals such as audio or video.

Additionally, it is clear that intelligent agents must adapt at least some parts of their model in response to a changing world. While human brain structure is regular between individuals at a coarse scale, it is clear that fine-grained structure is dictated primarily by exposure to the environment. People react differently to the same stimuli.

A large number of computational neuroscientists, cognitive scientists, machine learning researchers believe in a well-established formal model of how agents choose and plan their actions (Levine, 2018; Tenenbaum et al., 2011; Ghahramani, 2015). In this model agents construct and reason in models of their worlds (simulators), and use them to compute the expected utility of, or return to be had by, effecting some action — i.e. applying control inputs to their musculoskeletal systems, in the case of animals.

An abstraction of a significant part of this model building and inference paradigm was laid out by Dayan et al. (1995). They called their abstract machine for model-based perception and world-prediction the "Helmholtz machine." They posited the existence of intertwined forward and backward models in which the forward "decoder" model is used for simulating or predicting the world and the backward "encoder" model is used to encode a percept into a representation of the world in the latent space of the model. In other words, every state of the world is represented by a latent code, or, more precisely, a distribution over latent states of the world due to the fact that not everything is directly observable.

Kingma and Welling (2014) and Rezende et al. (2014) introduced a specific reduction of this idea to practice in the form of the variational autoencoder. In their work, specific architectures and techniques for realizing the general Helmholtz machine idea were proposed, using a variational inference objective of the form

$$\log p(Y;\theta) \geq \log p(Y;\theta) - D_{\mathrm{KL}}\left(q(X|Y;\phi) \,||\, p(X|Y;\theta)\right) \tag{7.7}$$

$$= \int q(X|Y;\phi)\left[\log p(X,Y;\theta) - \log q(X|Y;\phi)\right] dX \tag{7.8}$$

$$= \mathrm{ELBO}(\theta,\phi,Y). \tag{7.9}$$

Provided we assume that the generative model is differentiable with respect to its parameters θ, the sampling process for drawing a ran-

dom variate X from the distribution $q(X|Y;\phi)$ can be expressed in a manner which composes a differentiable deterministic function g and independent noise distribution $p(\epsilon)$, such that $g(Y,\epsilon;\phi) \stackrel{d}{=} q(X|Y;\phi)$, then the evidence lower bound $\text{ELBO}(\theta,\phi,Y)$ can be optimized via gradient ascent techniques using

$$\begin{aligned}\nabla_{\theta,\phi}\,\text{ELBO}(\theta,\phi,Y) &= \nabla_{\theta,\phi}\mathbb{E}_{q(X|Y;\phi)}\left[\log\frac{p(X,Y;\theta)}{q(X|Y;\phi)}\right] \\ &= \mathbb{E}_{p(\epsilon)}\left[\nabla_{\theta,\phi}\log\frac{p(g(Y,\epsilon;\phi),Y;\theta)}{q(g(Y,\epsilon;\theta)|Y;\phi)}\right] \\ &\approx \frac{1}{L}\sum_{\ell=1}^{L}\left[\nabla_{\theta,\phi}\log\frac{p(g(Y,\epsilon^{\ell};\phi),Y;\theta)}{q(g(Y,\epsilon^{\ell};\phi)|Y;\phi)}\right],\end{aligned}$$

where $\epsilon^{\ell} \sim p(\epsilon)$. Note that what is written here applies to a single observation Y only, and the log evidence of a dataset consisting of many observations would accumulate over the set of all observations yielding an outer loop around all gradient computations.

What the variational autoencoder does is quite elegant. Starting from observational data and parameterized encoder and decoder programs we can simultaneously adjust, via gradient ascent on the ELBO objective, the parameters of the generative model θ and the parameters of the inference network/encoder ϕ so as to simultaneously produce a good model $p(X,Y;\theta)$ and an amortized inference engine $q(X|Y;\phi)$ for the same model.

Variational autoencoders and probabilistic programming meet in many places. The most straightforward to see is that rather than the typically simple specific architecture choices for p and q, using probabilistic programming techniques to specify both can increase their expressivity and thereby potentially their performance too. Most variational autoencoder instantiations specify a single, conditionally independent and identically distributed prior for a single layer of latents, $p(X)$, and then via a purely deterministic differentiable procedure f map that code to the parameters of a usually simple likelihood $p(Y|f(X,\theta))$. This is, of course, rather different from the rich structure of possible generative programs denotable in probabilistic programming languages and means that simple, non-structured decoders must learn much of

what could be included explicitly as structural prior information. Some work has been done to increase the generality of the modeling language, such as making the decoder generative model a graphical model (Johnson et al., 2016). Work on program induction in the programming languages community, which is one way model learning can be understood, suggests a rule of thumb that says it is a good idea to impose as much structure as possible when learning or inducting a program. It remains to be seen whether very general model architectures and the magic of gradient descent will win out dominantly in the end over the top-down structuring approach.

On the flip side, the encoder $q(X|Y;\phi)$ is equivalent to our inference compilation artifacts in action. It so being, should it not reflect the structure of the generative model in order to achieve optimal performance? Also, would not it be better if the encoder could "reach into" the generative model and directly address conditional random choices made during the forward execution of the decoder if the decoder were more richly structured?

Various approaches to this have started to appear in the literature and these form the basis for the most tight connections between what we will also refer to as a variational autoencoder, but are more general and flexible than the original specification, and probabilistic programming. This has recently instantiated themselves in the form of probabilistic programming languages built on top of deep learning libraries. On top of TensorFlow, a distributions library (Dillon et al., 2017) provides implementations of random variable primitives which can be incorporated into deep generative models; Edward (Tran et al., 2016) provides a modeling and inference environment for defining structured, hierarchical distributions for encoders and decoders. On top of PyTorch, at time of writing a similar distributions library is in development, and two probabilistic programming languages (including Pyro (Uber, 2018) and probabilistic Torch (Siddharth et al., 2017)) are built. Unlike TensorFlow, PyTorch uses a dynamic approach to constructing computation graphs, making it easy to define models which include recursion and unbounded numbers of random choices — in short, HOPPL programs.

A potentially very exciting new chapter in the continuing collision between variational methods and probabilistic programming follows on

from the recent realization that general purpose inference methods like those utilized in probabilistic programming offer an avenue for tightening the lower bound for model evidence during variational autoencoder training while remaining compatible with more richly structured models and inference networks. Arguably the first example of this was the importance weighted autoencoder (Burda et al., 2016) which, effectively, uses an importance sampling inference engine during variational autoencoder training to both tighten the bound and minimize the variance of gradient estimates during training. A more advanced version of this idea that uses sequential Monte Carlo instead of importance sampling has appeared recently in triplicate simultaneously (Le et al., 2017c; Naesseth et al., 2017; Maddison et al., 2017). These papers all roughly identify and exploit the fact that the marginal probability estimate computed by sequential Monte Carlo tightens the ELBO even further (by giving it a new form), and moreover, the power of sequential Monte Carlo allows for the efficient and full intertwining of the decoder and encoder architectures even when either or both have internal latent stochastic random variables. The goal and hope is that the variational techniques combined with powerful inference algorithms will allow for simultaneous model refinement/learning and inference compilation in the full family of HOPPL-denotable models and beyond.

7.3 Hamiltonian Monte Carlo and Variational Inference

We introduced Hamiltonian Monte Carlo in Section 3.4 as a graph-based inference approach, and variational inference in Section 4.4 as evaluation-based inference in the FOPPL; neither of these we revisited in Chapter 6 as HOPPL inference algorithms. This is not because these algorithms are fundamentally difficult to adopt as HOPPL evaluators — in fact, it is straightforward to convert the BBVI evaluator in Algorithm 14 to the HOPPL — but rather because it is unclear whether these algorithms yield accurate posterior approximations for programs written in the HOPPL.

The particular challenge is the possibility of HOPPL programs to generate an unbounded number of random variables, where the number of random variables may differ from execution to execution.

Our BBVI evaluator in Section 4.4 produces a variational parameter at each random choice encountered during course of execution; for the FOPPL evaluator in which all random variables can be enumerated, this yields a fixed-size set of variational parameters. The direct approach to translating this algorithm to the HOPPL is to associate a variational parameter with each address α that corresponds to a `sample` statement. However, this can lead to pathological behaviour where certain addresses may *never* be encountered even once while fitting the approximating distribution, although those addresses may still eventually be produced by the program. A workaround is to consider a variational approximation which is defined by a finite set of approximating distributions, even if the probabilistic program itself can generate an arbitrary number of random variables during its execution; this can be done by defining equivalence classes over addresses α, such that multiple `sample` statements share a posterior approximation. This is particularly sensible for e.g. random choices which occur inside a rejection sampler, where multiple calls to the same distribution in the same function really do have the same posterior, conditioned on the fact that the random variable exists in the execution. However, in general it is difficult to decide automatically which random choices should share the same approximating distribution. Such systems can be used in practice so long as these mappings can be annotated manually, as for example in van de Meent et al. (2016).

Further difficulties arise if attempting to run Hamiltonian Monte Carlo on HOPPL programs. A Hamiltonian Monte Carlo transition kernel is not designed to run on spaces of varying dimensionality; instead, it could be used as a conditional update or proposal for a given fixed set of instantiated random variables. In a reversible jump MCMC setting, the HMC proposal could update these values while an additional alternative transition kernel could propose changes to the dimensionality of the model. However, this is also difficult to handle automatically in HOPPL programs; there is not necessarily any fixed parameter which denotes the dimensionality of the model, which instead depends on the control flow path of a particular execution. Changing the value of any continuous latent random variable could push the program onto a different branch (e.g. by changing the number of iterations of a stochastic recursion), which would then change the dimensionality of the model.

Implementing HMC in a higher-order programming language safely requires explicitly separating the latent random variables into those which are "structure preserving" (Yang et al., 2014), i.e. for which a change in value cannot change the dimensionality of the target, from those which may affect control flow.

7.4 Nesting

In this tutorial we have not provided a HOPPL construct for nesting probabilistic programs. However, embedded languages like Anglican, WebPPL, and Church inevitably tempt such a facility for many reasons including the existence of a porous boundary between the PPL and its host language. Nesting, in our terminology, means treating a HOPPL program like a distribution and using it within another HOPPL program as a distribution-type object, analogous to distributions provided as primitives in the language. In statistics this can correspond to a double intractable inference problem (Murray et al., 2012). Nesting would seem a natural and perhaps even straightforward thing to do because a HOPPL program denotes a parameterized conditional distribution (one for each set of observed variable values). As a consequence one might naturally ask why this HOPPL-denoted distribution ought not be treated just as any other distribution value, in the sense that it can be nested and passed as an argument to `sample` and `observe` in an another or outer HOPPL program. It turns out that this is very tricky to do correctly and that there remain opportunities to design languages and inference algorithm evaluators that do.

To even discuss this we need to assume the existence of two additional HOPPL features. One is a syntactic nesting construct; effectively a boundary around a HOPPL program. The shared `query` syntax that is used to separate Anglican, Church, and WebPPL programs from their host languages is one such example. Second, we need to assume the existence of a construct that reifies a HOPPL program into a distribution type. In Church and WebPPL this is implicitly tied up in the `query` construct itself, specifically `rejection-query` and `enumerate-query`. Other examples of this include the theoretically-grounded but impossible `norm` function of Staton et al. (2016), and the intentionally hidden

conditional construct from Anglican, which has flaws uncovered and criticized by Rainforth (2018).

Take for example the following hypothetical nesting-HOPPL program.

```
(let [inner (query [y D]
              (let [z (sample (gamma y 1))]
                (observe (normal y z) D)
               z))
      outer (query [D]
              (let [y (sample (beta 2 3))
                    z (sample (inner y D))]
              [y z D]))]
  (sample (outer D)))
```

Even the casual meaning of such a program is open to interpretation. Should the joint distribution over the return value be

$$\begin{aligned}\pi_1(y,z,D) &= p(y)p(z|y)p(D|y,z)\\ &= \text{Beta}(y;2,3)\Gamma(z;y,1)\mathcal{N}\left(D;y,z^2\right)\end{aligned}$$

or

$$\begin{aligned}\pi_2(y,z,D) = p(y)p(z|y,D) &= \frac{p(y)p(z|y)p(D|y,z)}{\int p(z|y)p(D|y,z)\mathrm{d}z} \overset{?}{}\\ &= \frac{p(y)p(z|y)p(D|y,z)}{p(D|y)} \neq \pi_1(z,y,D)\end{aligned}$$

The first interpretation is what you would expect if you were to inline the inner query as one can do for a function body in a pure functional language. While doing such a thing introduces no mathematical complications, it is incompatible with the conditional distribution semantics we have established for HOPPL programs. The second interpretation is correct in that it is in keeping with such semantics but introduces the extra marginal probability term $p(D|y)$ which is impossible to compute in the general case, complicating matters rather a lot.

Rainforth (2018) categorizes nesting into three types: "sampling from the conditional distribution of another query" (which we refer to as nested inference), "factoring the trace probability of one query with the partition function estimate of another" (which we refer to as nested conditioning), and "using expectation estimates calculated using

one query as first class variables in another." While this terminology is rather inelegant (and potentially confusing because it conflates problem and solution differences in the same categorization), the point remains. To even do "nested inference" one both must pay close attention to Rainforth et al. (2018) warnings about convergence rates for nested sampling and also utilize sampling methodologies that are specifically tailored to this situation (Rainforth, 2018; Naesseth et al., 2015).

Beyond these concerns Staton et al. (2016) also noticed that the posterior distribution fails to be defined in some models with certain observation distributions because the marginal likelihood of the observation can become infinite (Staton et al., 2016; Staton, 2017). Here is a variant of their example.

```
(let [x (sample (normal 0 1))
      px (* (/ 1 (sqrt (* 2 pi)))
            (exp (- (/ (* x x) 2))))]
  (observe (exponential (/ 1 px)) 0)
  x)
```

Program 7.1: HOPPL program with undefined posterior

This program defines a model with prior $p(x) = \text{Normal}(x; 0, 1)$ and likelihood $p(y|x) = \text{Exponential}(y; 1/p(x))$, and expresses that $y = 0$ is observed. The model fails to have a posterior because its marginal likelihood at $y = 0$ is infinite

$$\begin{aligned} p(y=0) = \int p(x, y=0)\mathrm{d}x &= \int p(x)p(y=0|x)\mathrm{d}x \\ &= \int p(x)\frac{1}{p(x)}\mathrm{d}x = \infty. \end{aligned}$$

Staton et al. (2016) argued that the formal semantics of a language construct for performing inference, such as `doquery` in Anglican, should account for this failure case. A message of this research to us is that when we define a model using the outcome of posterior inference of another nested model, we should make sure that this outcome is well-defined, because otherwise even a prior used within an outer model may become undefined.

Currently probabilistic programming languages perform inference on nested models in a way that is similar to the bad naïve nested

Monte-Carlo identified by Rainforth et al. (2018) and in a way that is not immune to the problem identified by Staton et al. (2016), and so they suffer from inefficiencies and, worse, inaccuracies. This suggests a potentially fruitful avenue for future research.

What should be noted is that nested query language constructs, were they to be operationalized efficiently and correctly, allow one to express extremely interesting and complex generative models that can involve mutually recursive theory-of-mind type reasoning, and so on. Goodman and his colleagues highlight many such models for agent interactions that capture agents' knowledge about other agents (Stuhlmüller and Goodman, 2014) and many source code examples are available online (Stuhlmüller, 2014).

7.5 Formal Semantics

Programs in probabilistic programming languages correspond to probabilistic models, and characterizing aspects of these models is the goal of inference algorithms. In most cases, the characterization is approximate, and describes the target model only partially. Although such partial description is good enough for many applications, it is not so for the developers of these languages, who have to implement compilers, optimizers, and inference algorithms and need to ensure that these implementations do not have bugs. For instance an optimizer within a PPL compiler should not change the probabilistic models denoted by programs, and an inference algorithm should be able to handle corner cases correctly, such as Program 7.1 in Section 7.4 that does not have a posterior distribution. To meet this obligation, developers need a method for mapping probabilistic programs to their precise meanings, i.e., a strict mathematical description of the denoted probabilistic models. The method does not have to be computable, but it should be formal and unambiguous, so that it can serve as ruler for judging correctness of transformations and implementations.

A formal semantics is such a method. It defines the mathematical meaning of every program in a probabilistic programming language. For instance, the semantics may map

```
(let [x (sample (normal 0 1))]
```

```
(observe (normal x 1) 2)
x)
```

to the normalized posterior distribution of the returned latent variable x (namely, $\text{Normal}(x; 1, \sqrt{2})$), or to its unnormalized counterpart $\text{Normal}(x; 0, 1) \times \text{Normal}(2; x, 1)$, which comes directly from the joint distribution of the latent x and the observed y that has the value 2.

A formal semantics is like integration. The integral of a complicated function may be impossible to compute, but its mathematical meaning is clearly defined. Similarly, the semantics might not tell us how to compute a probabilistic model from a given complicated program, but it tells us precisely what the model is.

Giving a good formal semantics to probabilistic programming languages turns out to be very challenging, and even requires revising the measure-theoretic foundation of modern probability theory in some cases. These issues can be seen in articles by Borgström et al. (2013), Staton et al. (2016), and Staton (2017). In the rest of this section, we focus on *one* issue caused by so-called higher-order functions, which are functions that take other functions as arguments or return functions as results; higher-order function are fully or partially supported by many probabilistic programming languages such as Church, Venture, Anglican, WebPPL and Pyro.

A good way to understand the issue with higher-order functions is to attempt to build a formal semantics for a language with higher-order functions and to observe how a natural decision in this endeavor ultimately leads to a dead end. The first step of this attempt is to notice that a large class of probability distributions can be expressed in the HOPPL and most other probabilistic programming languages. In particular, using these languages, we can express distributions on real numbers that do not have density functions with respect to the Lebesgue measure, and go easily outside of the popular comfort zone of using density functions to express and reason about probabilistic models. For instance, the HOPPL program

```
(if (sample (flip 0.5)) 1 (sample (normal 0 1)))
```

expresses a mixture of the Dirac distribution at 1 and the standard normal distribution, but because of the Dirac part, this mixture does

not have a density function with respect to Lebesgue measure.

A standard approach of formally dealing with such distributions is to use measure theory. In this theory, we use a so called *measurable space*, which is just a set X equipped with a family Σ of subsets of X that satisfies certain conditions. Elements in Σ are called *measurable*, and they denote events that can be assigned probabilities. A representative example of measurable space is the set of reals $\mathbb{R}$ together with the family $\mathcal{B}$ of so called Borel sets. A probability distribution on X is then defined to be a function from Σ to $[0,1]$ satisfying certain properties. For instance, the above HOPPL program denotes a distribution that assigns

$$0.5 \times \mathbb{I}(a < 1 < b) + 0.5 \times \int_a^b \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-x^2}{2}\right) \mathrm{d}x$$

to every interval (a, b). Another important piece of measure theory is that we consider only *good* functions f between two measurable spaces (X, Σ) and (X', Σ') in the sense that the inverse image of a measurable $B \in \Sigma'$ according to f is always measurable (i.e. $f^{-1}(B) \in \Sigma$). These functions are called *measurable functions*. When the domain (X, Σ) of such a measurable function is given a probability distribution, we often say that the function is a random variable. Using measure theory amounts to formalizing objects of interest in terms of measurable spaces, measurable sets, measurable functions and random variables (instead of usual sets and functions).

The second step of giving a semantics to the HOPPL is to interpret HOPPL programs using measure theory. It means to map HOPPL programs to measurable functions, constants in measurable spaces, or probability distributions. Unfortunately, this second step cannot be completed because of the following impossibility result by Aumann (1961):

Theorem 7.1 (Aumann)**.** Let F be the set of measurable functions on $(\mathbb{R}, \mathcal{B})$. Then, no matter which family Σ of measurable sets we use for F, we cannot make the following evaluation function measurable:

$$app : F \times \mathbb{R} \to \mathbb{R}$$
$$app(f, r) = f(r).$$

Here we assume that $\mathcal{B}$ is used as a family of measurable sets for $\mathbb{R}$ and that $F \times \mathbb{R}$ means the standard cartesian product of measurable spaces (F, Σ) and $(\mathbb{R}, \mathcal{B})$.

The result implies that the HOPPL function

```
(fn [f x] (f x))
```

cannot be interpreted as a measurable function, and so it lives outside of the realm of measure theory, regardless of what measurable space we use for the set of measurable functions on $(\mathbb{R}, \mathcal{B})$. We thus have to look for a more flexible alternative than measure theory.

Finding such an alternative has been a topic of active research. Here we briefly review a proposal by Heunen, Kammar, Staton and Yang (Heunen et al., 2017). The key of the proposal lies in their new formalization of probability theory that treats the random variable as a primary concept and axiomatizes it directly. Contrast this with the situation in measure theory where measurable sets are axiomatized first and then the notion of random variable is derived from this axiomatization (as measurable function from a measurable space with a probability distribution). It turns out that this shift of focus leads to a new notion of good functions, which is more flexible than measurability and lets one interpret HOPPL programs, such as the application function from above, as good functions.

More concretely, Heunen et al. (2017) axiomatized a set X equipped with a collection of X-valued random variables in terms of what they call *quasi-Borel space*. A quasi-Borel space is a pair of a set X and a collection M of functions from $\mathbb{R}$ to X that satisfies certain conditions, such as all constant functions being included in M. Intuitively, the functions in M represent X-valued random variables, and they use real numbers as random seeds and are capable of converting such random seeds to values in X. The measurable space $(\mathbb{R}, \mathcal{B})$ is one of the best-behaving measurable spaces, and using real numbers in this space as random seeds ensures that quasi-Borel spaces avoid pathological cases in measure theory. A less exciting but useful quasi-Borel space is $\mathbb{R}$ with the set $M_{\mathbb{R}}$ of measurable functions from $\mathbb{R}$ to itself, which is an example of quasi-Borel space generated from a measurable space. But

there are more exotic, interesting quasi-Borel spaces that do not arise from this recipe.

Heunen et al.'s axiomatization regards a function f from a quasi-Borel space (X, M) to (Y, N) as good if $f \circ r \in N$ for all $r \in M$; in words, f maps a random variable in M to a random variable in N. They have shown that such good functions on $(R, M_{\mathbb{R}})$ themselves form a quasi-Borel space when equipped with a particular set of function-valued random variables. Furthermore, they have proved that the application function `(fn [f x] (f x))` from above is a good function in their sense because it maps a pair of such function-valued random variable and $\mathbb{R}$-valued random variable to a random variable in $M_{\mathbb{R}}$.

Heunen et al.'s axiomatization has been used to define the semantics of probabilistic programming languages with higher-order functions and also to validate inference algorithms for such languages. For interested readers, we suggest Heunen et al. (2017) as well as Scibior et al. (2018).

8

Conclusion

Having made it this far (congratulations!), we can now summarize probabilistic programming relatively concisely and conclude with a few general remarks.

Probabilistic programming is largely about designing languages, interpreters, and compilers that translate inference problems denoted in programming language syntax into formal mathematical objects that allow and accommodate generic probabilistic inference, particularly Bayesian inference and conditioning. In the same way that techniques in traditional compilation are largely independent of both the syntax of the source language and the peculiarities of the target language or machine architecture, the probabilistic programming ideas and techniques that we have presented are largely independent of both the source language and the underlying inference algorithm.

While some might argue that knowing how a compiler works is not really a requirement for being a good programmer, we suggest that this is precisely the kind of deep knowledge that distinguishes truly excellent developers from the rest. Furthermore, as traditional compilation and evaluation infrastructure has been around, at the time of this writing, for over half a century, the level of sophistication and reliability of

implementations underlying abstractions like garbage collection are sufficiently high that, indeed, perhaps one can be a successful user of such a system without understanding deeply how it works. However, at this point in time, probabilistic programming systems have not developed to such a level of maturity and as such knowing something about how they are implemented will help even those people who only wish to develop and use probabilistic programs rather than develop languages and evaluators.

It may be that this state of affairs in probabilistic programming remains for comparatively longer time because of the fundamental computational characteristic of inference relative to forward computation. We have not discussed computational complexity at all in this text largely because there is, effectively, no point in doing so. It is well known that inference in even discrete-only random variable graphical models is NP-hard if no restrictions (e.g. bounding the maximum clique size) are placed on the graphical model itself. As the language designs we have discussed do not easily allow denoting or enforcing such restrictions, and, worse, allow continuous random variables, and in the case of HOPPLs, a potentially infinite collection of the same, inference is even harder. This means that probabilistic programming evaluators have to be founded on approximate algorithms that work well some of the time and for some problem types, rather than in the traditional programming language setting where the usual case is that exact computation works most of the time even though it might be prohibitively slow on some inputs. This is all to say that knowing intimately how a probabilistic programming system works will be, for the time being, necessary to be even a proficient power user.

These being early days in the development of probabilistic programming languages and systems means that there exist multiple opportunities to contribute to the foundational infrastructure, particularly on the approximate inference algorithm side of things. While the correspondence between first-order probabilistic programming languages and graphical models means that research to improve general-purpose inference algorithms for graphical models applies more-or-less directly to probabilistic programming systems, the same is not quite as true for HOPPLs. The primary challenge in HOPPLs, the infinite-dimensional

parameter space, is effectively unavoidable if one is to use a "standard" programming language as the model denotation language. This opens challenges related to inference that have not yet been entirely resolved and suggests a research quest towards developing a truly general-purpose inference algorithm.

In either case it should be clear at this point that not all inference algorithm research and development is equally useful in the probabilistic programming context. In particular developing a special-purpose inference algorithm designed to work well for exactly one model is, from the programming languages perspective, like developing a compiler optimization for a single program – not a good idea unless that one program is very important. There are indeed individual models that are that important, but our experience suggests that the amount of time one might spend on an optimized inference algorithm will typically be more than the total time accumulated from writing a probabilistic program once, right away, a simply letting a potentially slower inference algorithm proceed towards convergence.

Of course there also will be generations and iterations of probabilistic programming language designs with technical debt in terms of programs written accruing along with each successive iterate. What we have highlighted is the inherent tension between flexible language design, the phase transition in model parameter count, and the difficulty of the associated underlying inference problem. We have, as individual researchers, generally striven to make probabilistic programming work even with richly expressive modeling languages (i.e. "regular" programming languages) for two reasons. One, the accrued technical debt of simulators written in traditional programming languages should be elegantly repurposable as generative models. The other is simply aesthetic. There is much to be said for avoiding the complications that come along with such a decision and this presents an interesting language design challenge: how to make the biggest, finite-variable cardinality language that allows natural model denotation, efficient forward calculation, and minimizes surprises about what will "compile" and what will not. And if modeling language flexibility is desired, our thinking has been, why not use as much existing language design and infrastructure as possible?

Our focus throughout this text has mostly been on automating

inference in known and fixed models, and reporting state of the art techniques for such one-shot inference, however we believe that the challenge of model learning and rapid, approximate, repeated inference are both of paramount importance, particularly for artificial intelligence applications. Our belief is that probabilistic programming techniques, and really more the practice of paying close attention to how language design choices impact both what the end user can do easily and what the evaluator can compute easily, should be considered throughout the evolution of the next toolchain for artificial intelligence operations.

References

Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng (2015), 'TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems'.

Abelson, H., G. J. Sussman, and J. Sussman (1996), *Structure and interpretation of computer programs*. Justin Kelly.

Alberti, M., G. Cota, F. Riguzzi, and R. Zese (2016), 'Probabilistic logical inference on the web'. In: *AI* IA 2016 Advances in Artificial Intelligence*. Springer, pp. 351–363.

Andrieu, C., A. Doucet, and R. Holenstein (2010), 'Particle Markov chain Monte Carlo methods'. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* **72**(3), 269–342.

Appel, A. W. (2006), *Compiling with Continuations*. Cambridge University Press.

Aumann, R. J. (1961), 'Borel structures for function spaces'. *Illinois Journal of Mathematics* **5**, 614–630.

Baydin, A. G., L. Heinrich, W. Bhimji, B. Gram-Hansen, G. Louppe, L. Shao, K. Cranmer, F. Wood, et al. (2018), 'Efficient Probabilistic Inference in the Quest for Physics Beyond the Standard Model'. *arXiv preprint arXiv:1807.07706*.

Baydin, A. G., B. A. Pearlmutter, A. A. Radul, and J. M. Siskind (2015), 'Automatic Differentiation in Machine Learning: A Survey'. *arXiv preprint arXiv:1502.05767*.

Bishop, C. M. (2006), *Pattern recognition and machine learning*. Springer.

Borgström, J., A. D. Gordon, M. Greenberg, J. Margetson, and J. V. Gael (2013), 'Measure Transformer Semantics for Bayesian Machine Learning'. *Logical Methods in Computer Science* **9**(3).

Burda, Y., R. Grosse, and R. Salakhutdinov (2016), 'Importance Weighted Autoencoders'. In: *ICLR*.

Bursztein, E., J. Aigrain, A. Moscicki, and J. C. Mitchell (2014), 'The End is Nigh: Generic Solving of Text-based CAPTCHAs.'. In: *WOOT*.

Casado, M. L. (2017), 'Compiled Inference with Probabilistic Programming for Large-Scale Scientific Simulations'. Master's thesis, University of Oxford.

Chen, T., M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang (2016), 'MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems'. *In Neural Information Processing Systems, Workshop on Machine Learning Systems*.

Cornebise, J., É. Moulines, and J. Olsson (2008), 'Adaptive methods for sequential importance sampling with application to state space models'. *Statistics and Computing* **18**, 461–480.

Cornebise, J., É. Moulines, and J. Olsson (2014), 'Adaptive sequential Monte Carlo by means of mixture of experts'. *Statistics and Computing* **24**, 317–337.

Davidson-Pilon, C. (2015), *Bayesian methods for hackers: probabilistic programming and Bayesian inference*. Addison-Wesley Professional.

Dayan, P., G. E. Hinton, R. M. Neal, and R. S. Zemel (1995), 'The Helmholtz machine'. *Neural Computation* **7**(5), 889–904.

Del Moral, P. and L. M. Murray (2015), 'Sequential Monte Carlo with highly informative observations'. *SIAM/ASA Journal on Uncertainty Quantification* **3**(1), 969–997.

Dillon, J. V., I. Langmore, D. Tran, E. Brevdo, S. Vasudevan, D. Moore, B. Patton, A. Alemi, M. Hoffman, and R. A. Saurous (2017), 'TensorFlow Distributions'. *arXiv preprint arXiv:1711.10604*.

Doucet, A., S. Godsill, and C. Andrieu (2000), 'On sequential Monte Carlo sampling methods for Bayesian filtering'. *Statistics and computing* **10**(3), 197–208.

Duchi, J., E. Hazan, and Y. Singer (2011), 'Adaptive subgradient methods for online learning and stochastic optimization'. *Journal of Machine Learning Research* **12**(Jul), 2121–2159.

Friedman, D. P. and M. Wand (2008), *Essentials of programming languages*. MIT press.

Ge, H., K. Xu, and Z. Ghahramani (2018), 'Turing: A Language for Flexible Probabilistic Inference'. In: A. Storkey and F. Perez-Cruz (eds.): *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, Vol. 84 of *Proceedings of Machine Learning Research*. Playa Blanca, Lanzarote, Canary Islands, pp. 1682–1690, PMLR.

Gehr, T., S. Misailovic, and M. Vechev (2016), 'PSI: Exact symbolic inference for probabilistic programs'. In: *International Conference on Computer Aided Verification*. pp. 62–83.

Gelman, A., J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, and D. B. Rubin (2013), 'Bayesian data analysis, 3rd edition'.

Geman, S. and D. Geman (1984), 'Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images'. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **6**, 721–741.

Gershman, S. J. and N. D. Goodman (2014), 'Amortized Inference in Probabilistic Reasoning'. In: *Proceedings of the Thirty-Sixth Annual Conference of the Cognitive Science Society*.

Ghahramani, Z. (2015), 'Probabilistic machine learning and artificial intelligence'. *Nature* **521**(7553), 452–459.

Goodfellow, I., Y. Bengio, and A. Courville (2016), *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Goodman, N., V. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum (2008), 'Church: a language for generative models'. In: *Proc. 24th Conf. Uncertainty in Artificial Intelligence (UAI)*. pp. 220–229.

Goodman, N. D. and A. Stuhlmüller (2014), 'The Design and Implementation of Probabilistic Programming Languages'. http://dippl.org. Accessed: 2017-8-22.

Google (2018), 'Protocol Buffers'. [Online; accessed 15-Aug-2018].

Gordon, A. D., T. A. Henzinger, A. V. Nori, and S. K. Rajamani (2014), 'Probabilistic programming'. In: *Proceedings of the on Future of Software Engineering*. pp. 167–181.

Gram-Hansen, B., Y. Zhou, T. Kohn, H. Yang, and F. Wood (2018), 'Discontinuous Hamiltonian Monte Carlo for Probabilistic Programs'. *arXiv preprint arXiv:1804.03523*.

Griewank, A. and A. Walther (2008), *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM.

Gulwani, S., O. Polozov, R. Singh, et al. (2017), 'Program synthesis'. *Foundations and Trends® in Programming Languages* **4**(1-2), 1–119.

Haario, H., E. Saksman, and J. Tamminen (2001), 'An adaptive Metropolis algorithm'. *Bernoulli* pp. 223–242.

Herbrich, R., T. Minka, and T. Graepel (2007), 'TrueSkill$^{\mathrm{TM}}$: A Bayesian Skill Rating System'. In: *Advances in Neural Information Processing Systems*. pp. 569–576.

Heunen, C., O. Kammar, S. Staton, and H. Yang (2017), 'A convenient category for higher-order probability theory'. In: *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. pp. 1–12.

Hickey, R. (2008), 'The Clojure Programming Language'. In: *Proceedings of the 2008 Symposium on Dynamic Languages*. New York, NY, USA, pp. 1:1–1:1, ACM.

Hwang, I., A. Stuhlmüller, and N. D. Goodman (2011), 'Inducing probabilistic programs by Bayesian program merging'.

Johnson, M., T. L. Griffiths, and S. Goldwater (2007), 'Adaptor grammars: A framework for specifying compositional nonparametric Bayesian models'. In: *Advances in neural information processing systems*. pp. 641–648.

Johnson, M. J., D. Duvenaud, A. Wiltschko, S. Datta, and R. Adams (2016), 'Structured VAEs: Composing probabilistic graphical models and variational autoencoders'. *NIPS 2016*.

Kimmig, A. and L. De Raedt (2017), 'Probabilistic logic programs: Unifying program trace and possible world semantics'.

Kimmig, A., B. Demoen, L. De Raedt, V. S. Costa, and R. Rocha (2011), 'On the implementation of the probabilistic logic programming language ProbLog'. *Theory and Practice of Logic Programming* **11**(2-3), 235–262.

Kingma, D. and J. Ba (2015), 'Adam: A method for stochastic optimization'. In: *Proceedings of the International Conference on Learning Representations (ICLR)*.

Kingma, D. P. and M. Welling (2014), 'Auto-encoding variational Bayes'. In: *Proceedings of the International Conference on Learning Representations (ICLR)*.

Koller, D. and N. Friedman (2009), 'Probabilistic graphical models: principles and techniques'.

Koller, D., D. McAllester, and A. Pfeffer (1997), 'Effective Bayesian inference for stochastic programs'. *AAAI* pp. 740–747.

Kulkarni, T. D., P. Kohli, J. B. Tenenbaum, and V. K. Mansinghka (2015a), 'Picture: a probabilistic programming language for scene perception'. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Kulkarni, T. D., W. F. Whitney, P. Kohli, and J. Tenenbaum (2015b), 'Deep convolutional inverse graphics network'. In: *Advances in Neural Information Processing Systems*. pp. 2539–2547.

Łatuszyński, K., G. O. Roberts, J. S. Rosenthal, et al. (2013), 'Adaptive Gibbs samplers and related MCMC methods'. *The Annals of Applied Probability* **23**(1), 66–98.

Le, T. A., A. G. Baydin, and F. Wood (2017a), 'Inference Compilation and Universal Probabilistic Programming'. In: *20th International Conference on Artificial Intelligence and Statistics, April 20–22, 2017, Fort Lauderdale, FL, USA*.

Le, T. A., A. G. Baydin, R. Zinkov, and F. Wood (2017b), 'Using synthetic data to train neural networks is model-based reasoning'. *2017 International Joint Conference on Neural Networks (IJCNN)* pp. 3514–3521.

Le, T. A., M. Igl, T. Jin, T. Rainforth, and F. Wood (2017c), 'Auto-Encoding Sequential Monte Carlo'. *arXiv preprint arXiv:1705.10306*.

LeCun, Y., Y. Bengio, and G. Hinton (2015), 'Deep learning'. *Nature* **521**(7553), 436–444.

Levine, S. (2018), 'Reinforcement Learning and Control as Probabilistic Inference: Tutorial and Review'. *arXiv preprint arXiv:1805.00909*.

Liang, P., M. I. Jordan, and D. Klein (2010), 'Learning programs: A hierarchical Bayesian approach'. pp. 639–646.

Maddison, C. J., D. Lawson, G. Tucker, N. Heess, M. Norouzi, A. Mnih, A. Doucet, and Y. W. Teh (2017), 'Filtering Variational Objectives'. *arXiv preprint arXiv:1705.09279*.

Mansinghka, V., T. D. Kulkarni, Y. N. Perov, and J. Tenenbaum (2013), 'Approximate Bayesian image interpretation using generative probabilistic graphics programs'. In: *Advances in Neural Information Processing Systems*. pp. 1520–1528.

Mansinghka, V., D. Selsam, and Y. Perov (2014), 'Venture: a higher-order probabilistic programming platform with programmable inference'. *arXiv* p. 78.

Mansinghka, V., R. Tibbetts, J. Baxter, P. Shafto, and B. Eaves (2015), 'BayesDB: A Probabilistic Programming System for Querying the Probable Implications of Data'. *arXiv preprint arXiv:1512.05006*.

McCallum, a., K. Schultz, and S. Singh (2009), 'Factorie: Probabilistic programming via imperatively defined factor graphs'. In: *Advances in Neural Information Processing Systems*, Vol. 22. pp. 1249–1257.

McLachlan, G. and D. Peel (2004), *Finite mixture models*. John Wiley & Sons.

Milch, B., B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov (2005), 'BLOG : Probabilistic Models with Unknown Objects'. In: *IJCAI*.

Minka, T. and J. Winn (2009), 'Gates'. In: *Advances in Neural Information Processing Systems*. pp. 1073–1080.

Minka, T., J. Winn, J. Guiver, and D. Knowles (2010a), 'Infer .NET 2.4, Microsoft Research Cambridge'.

Minka, T., J. Winn, J. Guiver, and D. Knowles (2010b), 'Infer.NET 2.4, 2010. Microsoft Research Cambridge'.

Murphy, K. P. (2012), 'Machine learning: a probabilistic perspective'.

Murray, I., Z. Ghahramani, and D. MacKay (2012), 'MCMC for doubly-intractable distributions'. *arXiv preprint arXiv:1206.6848*.

Murray, L., D. Lundën, J. Kudlicka, D. Broman, and T. B. Schön (2018), 'Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs'. In: *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*. pp. 1037–1046.

Murray, L. M. (2013), 'Bayesian state-space modelling on high-performance hardware using LibBi'. *arXiv preprint arXiv:1306.3277*.

Naesseth, C., F. Lindsten, and T. Schon (2015), 'Nested sequential Monte Carlo methods'. In: *International Conference on Machine Learning*. pp. 1292–1301.

Naesseth, C. A., S. W. Linderman, R. Ranganath, and D. M. Blei (2017), 'Variational Sequential Monte Carlo'. *arXiv preprint arXiv:1705.11140*.

Narayanan, P., J. Carette, W. Romano, C. Shan, and R. Zinkov (2016), 'Probabilistic inference by program transformation in Hakaru (system description)'. In: *International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*. pp. 62–79.

Neal, R. M. (1993), 'Probabilistic inference using Markov chain Monte Carlo methods'.

Nori, A. V., C.-K. Hur, S. K. Rajamani, and S. Samuel (2014), 'R2: An Efficient MCMC Sampler for Probabilistic Programs.'. In: *AAAI*. pp. 2476–2482.

Norvig, P. (2010), '(How to Write a (Lisp) Interpreter (in Python))'. [Online; accessed 14-Aug-2018].

Okasaki, C. (1999), *Purely functional data structures*. Cambridge University Press.

OpenBugs (2009), 'Pumps: conjugate gamma-Poisson hierarchical model'. Available online at http://www.openbugs.net/Examples/Pumps.html.

Paige, B. and F. Wood (2014), 'A compilation target for probabilistic programming languages'. In: *Proceedings of the 31st international conference on Machine learning*, Vol. 32 of *JMLR: W&CP*. pp. 1935–1943.

Paige, B. and F. Wood (2016), 'Inference Networks for Sequential Monte Carlo in Graphical Models'. In: *Proceedings of the 33rd International Conference on Machine Learning*, Vol. 48 of *JMLR: W&CP*. pp. 3040–3049.

Paszke, A., S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer (2017), 'Automatic Differentiation in PyTorch'.

Perov, Y. and F. Wood (2016), 'Automatic Sampler Discovery via Probabilistic Programming and Approximate Bayesian Computation'. In: *Artificial General Intelligence*. pp. 262–273.

Pfeffer, A. (2001), 'IBAL: A probabilistic rational programming language'. *IJCAI International Joint Conference on Artificial Intelligence* pp. 733–740.

Pfeffer, A. (2009), 'Figaro: An object-oriented probabilistic programming language'. Technical report.

Pfeffer, A. (2016), *Practical probabilistic programming*. Manning Publications Co.

Plummer, M. (2003), 'JAGS: A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling'. *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003). March* pp. 20–22.

Powell, H. (2015), 'A quick and dirty introduction to ZeroMQ'. [Online; accessed 15-Aug-2018].

Rabiner, L. R. (1989), 'A tutorial on hidden Markov models and selected applications in speech recognition'. *Proceedings of the IEEE* **77**(2), 257–286.

Rainforth, T. (2018), 'Nesting Probabilistic Programs'. *arXiv preprint arXiv:1803.06328*.

Rainforth, T., R. Cornish, H. Yang, and A. Warrington (2018), 'On nesting Monte Carlo estimators'. In: *International Conference on Machine Learning*. pp. 4264–4273.

Ranganath, R., S. Gerrish, and D. M. Blei (2014), 'Black box variational inference'. *International Conference on Machine Learning*.

Rasmussen, C. E. and Z. Ghahramani (2001), 'Occam's razor'. In: *Advances in neural information processing systems*. pp. 294–300.

Rezende, D. J., S. Mohamed, and D. Wierstra (2014), 'Stochastic Backpropagation and Approximate Inference in Deep Generative Models'. In: *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*. pp. 1278–1286.

Ritchie, D., B. Mildenhall, N. D. Goodman, and P. Hanrahan (2015), 'Controlling procedural modeling programs with stochastically-ordered sequential Monte Carlo'. *ACM Transactions on Graphics (TOG)* **34**(4), 105.

Ritchie, D., A. Stuhlmüller, and N. Goodman (2016a), 'C3: Lightweight Incrementalized MCMC for Probabilistic Programs using Continuations and Callsite Caching'. In: *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*. pp. 28–37.

Ritchie, D., A. Thomas, P. Hanrahan, and N. Goodman (2016b), 'Neurally-Guided Procedural Models: Amortized Inference for Procedural Graphics Programs using Neural Networks'. In: *Advances In Neural Information Processing Systems*. pp. 622–630.

Salvatier, J., T. V. Wiecki, and C. Fonnesbeck (2016), 'Probabilistic programming in Python using PyMC3'. *PeerJ Computer Science* **2**, e55.

Sato, T. and Y. Kameya (1997), 'PRISM: A language for symbolic-statistical modeling'. *IJCAI International Joint Conference on Artificial Intelligence* **2**, 1330–1335.

Schulman, J., N. Heess, T. Weber, and P. Abbeel (2015), 'Gradient estimation using stochastic computation graphs'. In: *Advances in Neural Information Processing Systems*. pp. 3528–3536.

Scibior, A., O. Kammar, M. Vákár, S. Staton, H. Yang, Y. Cai, K. Ostermann, S. K. Moss, C. Heunen, and Z. Ghahramani (2018), 'Denotational validation of higher-order Bayesian inference'. *PACMPL* **2**(POPL), 60:1–60:29.

Seide, F. and A. Agarwal (2016), 'CNTK: Microsoft's Open-Source Deep-Learning Toolkit'. In: *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA, pp. 2135–2135, ACM.

Siddharth, N., B. Paige, J.-W. van de Meent, A. Desmaison, N. Goodman, P. Kohli, F. Wood, and P. Torr (2017), 'Learning Disentangled Representations with Semi-Supervised Deep Generative Models'. In: *Advances in Neural Information Processing Systems*. pp. 5925–5935.

Spiegelhalter, D. J., A. Thomas, N. G. Best, and W. R. Gilks (1995), 'BUGS: Bayesian inference using Gibbs sampling, Version 0.50'.

Stan Development Team (2014), 'Stan: A C++ Library for Probability and Sampling, Version 2.4'.

Staton, S. (2017), 'Commutative Semantics for Probabilistic Programming'. In: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. pp. 855–879.

Staton, S., H. Yang, F. Wood, C. Heunen, and O. Kammar (2016), 'Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints'. In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*. pp. 525–534.

Stuhlmüller, A. (2014). [Online; accessed 15-Aug-2018].

Stuhlmüller, A. and N. D. Goodman (2014), 'Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs'. *Cognitive Systems Research* **28**, 80–99.

Stuhlmüller, A., J. Taylor, and N. Goodman (2013), 'Learning Stochastic Inverses'. In: *Advances in Neural Information Processing Systems 26*. pp. 3048–3056.

Tenenbaum, J. B., C. Kemp, T. L. Griffiths, and N. D. Goodman (2011), 'How to grow a mind: Statistics, structure, and abstraction'. *science* **331**(6022), 1279–1285.

Thrun, S. (2000), 'Towards programming tools for robots that integrate probabilistic computation and learning'. *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)* **1**(April).

Todeschini, A., F. Caron, M. Fuentes, P. Legrand, and P. Del Moral (2014), 'Biips: Software for Bayesian Inference with Interacting Particle Systems'. *arXiv preprint arXiv:1412.3779*.

Tolpin, D., J. W. van de Meent, H. Yang, and F. Wood (2016), 'Design and implementation of probabilistic programming language Anglican'. *arXiv preprint arXiv:1608.05263*.

Tran, D., M. D. Hoffman, R. A. Saurous, E. Brevdo, K. Murphy, and D. M. Blei (2017), 'Deep probabilistic programming'. *arXiv preprint arXiv:1701.03757*.

Tran, D., A. Kucukelbir, A. B. Dieng, M. Rudolph, D. Liang, and D. M. Blei (2016), 'Edward: A library for probabilistic modeling, inference, and criticism'. *arXiv preprint arXiv:1610.09787*.

Tristan, J.-B., D. Huang, J. Tassarotti, A. C. Pocock, S. Green, and G. L. Steele (2014), 'Augur: Data-Parallel Probabilistic Modeling'. In: Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (eds.): *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc., pp. 2600–2608.

Uber (2018), 'Pyro'. [Online; accessed 15-Aug-2018].

van de Meent, J. W., B. Paige, D. Tolpin, and F. Wood (2016), 'Black-box policy search with probabilistic programs'. In: *Proceedings of the 19th International conference on Artificial Intelligence and Statistics*, Vol. 41 of *JMLR: W&CP*. pp. 1195–1204.

Van Der Merwe, R., A. Doucet, N. De Freitas, and E. Wan (2000), 'The unscented particle filter'. In: *Advances in Neural Information Processing Systems*. pp. 584–590.

Webb, S., A. Golinski, R. Zinkov, N. Siddharth, T. Rainforth, Y. W. Teh, and F. Wood (2017), 'Faithful Inversion of Generative Models for Effective Amortized Inference'. *arXiv preprint arXiv:1712.00287*.

Whiteley, N., A. Lee, K. Heine, et al. (2016), 'On the role of interaction in sequential Monte Carlo algorithms'. *Bernoulli* **22**(1), 494–529.

Wikipedia contributors (2018), 'Pattern Matching'. [Online; accessed 14-Aug-2018].

Wingate, D., A. Stuhlmueller, and N. D. Goodman (2011), 'Lightweight implementations of probabilistic programming languages via transformational compilation'. In: *Proceedings of the 14th international conference on Artificial Intelligence and Statistics*. p. 131.

Wingate, D. and T. Weber (2013), 'Automated variational inference in probabilistic programming'. *arXiv preprint arXiv:1301.1299*.

Wood, F., J. van de Meent, and V. Mansinghka (2014a), 'A new approach to probabilistic programming inference'. In: *Artificial Intelligence and Statistics*. pp. 1024–1032.

Wood, F., J. van de Meent, and V. Mansinghka (2015), 'A new approach to probabilistic programming inference'. *arXiv preprint arXiv:1507.00996*.

Wood, F., J. W. van de Meent, and V. Mansinghka (2014b), 'A New Approach to Probabilistic Programming Inference'. In: *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*.

Yang, L., P. Hanrahan, and N. D. Goodman (2014), 'Generating Efficient MCMC Kernels from Probabilistic Programs'. In: *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*. pp. 1068–1076.