Foundations and Trends® in Machine Learning
Vol. XX, No. XX (2018) 1–209

DOI: 10.1561/XXXXXXXXXX

now
the essence of knowledge

An Introduction to Probabilistic Programming

Jan-Willem van de Meent
Northeastern University
j.vandemeent@northeastern.edu

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

Hongseok Yang
KAIST
hongseok00@gmail.com

Frank Wood
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 establishes a common foundation on which designers of future probabilistic programming languages may build. It is aimed at people who have an undergraduate level understanding of either or, ideally, both probabilistic machine learning and programming languages.

It starts with an ode to model-based reasoning, reminding readers about what a computational model is, and highlights the notion of conditioning as a foundational computation required in the fields of probabilistic machine learning and artificial intelligence. It then introduces a simple, restricted probabilistic programming language (PPL) that allows static-computation graph, finite-variable-cardinality model denotation and inference. In the context of this restricted PPL it also discusses various fundamental inference algorithms and describes how they can be implemented as language evaluators that "solve" denotable inference problems.

It then moves on to more current and speculative work by introducing a less-restricted language that opens up the possibility using established, unrestricted programming languages for model denotation at the cost of requiring inference algorithms that work on dynamic-computation-graph, infinite-variable-cardinality models. Foundational inference algorithms for this kind of probabilistic programming language are explained in the context of an interface between modeling language and inference evaluator that leads to efficient implementations.

This document finishes 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 the deep neural network community and the development of systems for next-generation artificial intelligence applications.

Introduction to Probabilistic Programming. Foundations and Trends® in Machine Learning, vol. XX, no. XX, pp. 1–209, 2018.
DOI: 10.1561/XXXXXXXXXX.

Acknowledgements

We would like to thank a large number of people who have read through preliminary versions of this introduction. These include in particular Rob Zinkov, Marcin Szymczak, Gunes Baydin, Andrew Warrington, Yuan Zhou, Celeste ..., and various other members of Frank Wood's Oxford research group.

Brooks Paige and Frank Wood were supported by The Alan Turing Institute under the EPSRC grant EP/N510129/1. Frank Wood was additionally supported by Intel and DARPA under its PPAML (FA8750-14-2-0006) and D3M (FA8750-17-2-0093) programs.

[**ToDo:** *Add everybody else's acknowledgements.*]

Notation

Grammars

[**FW:** *check to ensure that Bayesian network is changed to graphical model, the definition of Markov Blanket is correct, that "the probability mass or density" in the factor graph section of the text is changed to "factor" or "potential function" and that the cardinality of the set of observed nodes in the bayesian network section is not* $|V|$]

$c ::=$ A constant value or primitive operation.
$v ::=$ 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 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(`(let [`v_1 c_1`] (* `v_1 v_2`))` $= \{v_2\}$	The free variables in an expression.

Directed Graphical Models

$G = (V, A, \mathcal{P}, \mathcal{Y})$	A directed graphical model (i.e. a Bayesian network).
$V = \{v_1, \ldots, v_{\lvert V \rvert}\}$	The variable nodes in the network.
$Y = \text{dom}(\mathcal{Y}) \subseteq V$	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 $u_i \in A$ and children $v_i \in A$.

$\mathcal{P} = [v_1 \mapsto E_1, \ldots, v_{\lvert V\rvert} \mapsto E_{\lvert V\rvert}]$	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 $\mathcal{Y}(y_i) = c_i$.
$\textsc{pa}(v) = \{u : (u, v) \in A\}$	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.
$\Psi = [f_1 \mapsto E_1, \ldots, f_{\lvert F\rvert} \mapsto E_{\lvert F\rvert}]$	The probability mass or density of each factor $\mathcal{P}(f_i) = E_i$, represented as a target language expression..

Traces

$\mathcal{X} = [x_1 \mapsto c_1, \ldots, x_n \mapsto c_n]$	A trace of values $\mathcal{X}(x_i) = c_i$ assocated with the instantiated set of variables $X = \text{dom}(\mathcal{X})$.
$p(X = \mathcal{X}) = p(x_1 = c_1, \ldots, x_n = c_n)$	The probability of a trace.

Remaining TODOs

- ☐ Chapter 1
  - ☐ Editing pass
- ☐ Chapter 2
  - ☐ Decide on naming convention for ‘cmap‘, ‘foreach‘, etc
  - ☐ Editing pass
- ☐ Chapter 3
  - ☐ Figure out section / subsection hierarchy
  - ☐ Update Gibbs sampling section to address single-site vs block Gibbs updates
  - ☐ Explain the restrictions necessary to run HMC on a graph
  - ☐ Describe HMC as a block Gibbs proposal
  - ☐ Make the algorithm blocks for Gibbs and HMC clear
  - ☐ Check math, possibly merge the factor graph + EP sections

- ☐ Chapter 4
  - ☐ Editing pass, and hopefully not too much else?
- ☐ Chapter 5
  - ☐ Clean up Captcha example (needs updated now that Captcha is also up front)
  - ☐ Clean up figures / description for program induction section
  - ☐ Edit pass for coherence with chapter 2
- ☐ Chapter 6
  - ☐ Requires full review for both coherence and correctness
- ☐ Chapter 7
  - ☐ Consider going into detail (e.g. math) for Sid's paper, and including a figure
  - ☐ Consider going into detail (e.g. math) for AESMC, and including a figure
  - ☐ Add a section on why HMC and BBVI are difficult to reason about correctness in HOPPL
  - ☐ Edit / review nested programs section
  - ☐ Edit pass for coherence
- ☐ Chapter 8
  - ☐ Write anything at all, can be only a page

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 languages tools that automate the tedious and troublesome derivation and calculation of gra-

dients 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. [**FW:** *give a stats justification here*] 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 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 describe 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 2.3.2 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 inference 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 is 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.

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. The latter, 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 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

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

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., 2015a] appearing first in [Mansinghka et al., 2013] and then subsequently in [Le et al., 2017b] and [Le et al., 2017a] 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 that both 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 $P(X|Y)$ arrived at by conditioning reflects this uncertainty.

By this simple example, example source code for which appears in Chapter 5, 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

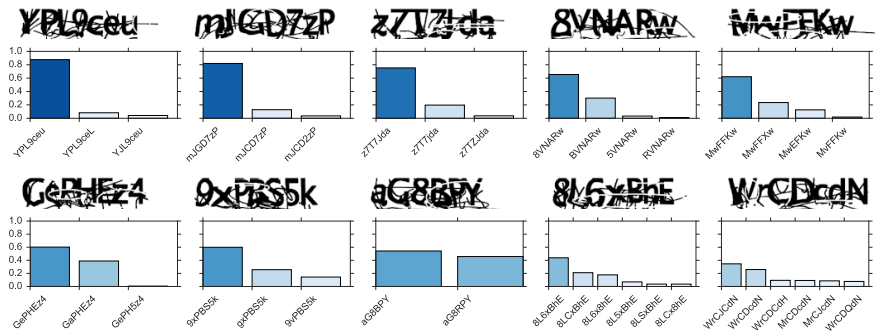

Figure 1.1: Posterior uncertainties after inference in a probabilistic programming language model of 2017 Facebook Captchas (reproduced from [Le et al., 2017b])

allow denotation of any such model. What this book 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 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, instead to do conditioning, here, abusing notation, to compute $p(x|y = \text{“heads}'')$ for instance. Using Bayes rule this is *theoretically* easy to do, it's 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 extremely 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, 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 thus owing to the fact that the beta distribution must sum to one

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

and consequently, by inspection,

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

or equivalently

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

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 owing to dimensionality.

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

[**FW:** *should the simple maths in the following be in capitals or in smalls?*] 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 the probability the bias of the coin is greater than 0.7 is given that the coin came up heads. This is mathematically denoted as

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

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 equivalent to a set of samples on the right 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 thereto 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 applications from machine learning using the inference algorithms and theory from statistics.

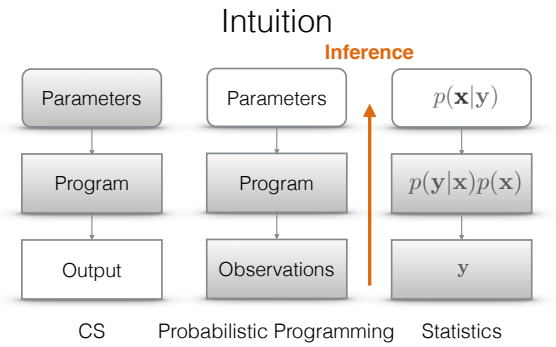

Figure 1.2: Probabilistic programming, an intuitive view.

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. Thinking back to our earlier example, reasoning about the bias of a coin is an example

of just this. 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 learn-

ing 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 because this sometimes trips up even sophisticates. 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, just programming languages.

There are many libraries for constructing graphical models and performing inference in an often programmatically constructed graphical model data-structure given observations. What distinguishes between this kind of approach and probabilistic programming is that a program is used to construct a model data-structure rather than the model arising directly from the evaluation of the program expression itself. In probabilistic programming the model data-structure is either constructed via a non-standard interpretation of the probabilistic program itself (if it can be, see Chapter 3) or is a general Markov model whose state is the evolving evaluation environment generated by the probabilistic programming language evaluator (see Chapter 4). Another way to make this distinction is that in Chapter 3, we consider inference techniques that compile a program to a density function whereas in Chapters 4 and 6 we consider methods that are fundamentally generative.

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., 2014] (updated syntax [Wood et al., 2015]), BLOG [Milch et al., 2005], Turing.jl [Ge et al., In submission], BayesDB [Mansinghka et al., 2015], Venture [Mansinghka et al., 2014], Probabilistic-C [Paige and Wood, 2014], webPPL [Goodman and Stuhlmüller, 2014a], CPProb [Casado, 2017], [Koller et al., 1997],[Thrun, 2000], statistics: Biips [Todeschini et al., 2014], LibBi [Murray, 2013], 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 if not all implementors will take. That said the real substance of this work

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

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, so far, 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 work 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 it is usually inefficient to do so, and, frankly, usually not helpful in terms intuition and clarity. We have already given an one example, Captcha from earlier in this chapter, let us proceed to more.

Constrained Simulation

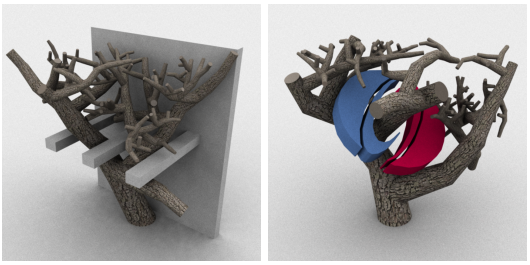

Figure 1.3: Posterior samples of procedurally generated, constrained trees (reproduced from [Ritchie et al., 2015])

A visually compelling and elucidating probabilistic programming example application is constrained procedural graphics [Ritchie et al., 2015]. Consider how one makes a computer graphics forest for a movie or computer game. One does not hire one thousand designers to each create a tree, one instead hires one 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 arranged 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 element or 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 each 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 at least one tree that obeys the statistical regularities specified by the prior and, as well, the specified constraints.

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]. 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 modelling 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. adapter 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 demon-

strate multi-agent, mutually-recursive reasoning agents, a number of examples for which are detailed in an excellent online tutorial [Goodman and Stuhlmüller, 2014b].

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 particularly 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 [Lezcano-Casado et al., 2017]. By "observing" the detector outputs, inference in the generative model specified by the simulation pipeline may prove able to produce the highest fidelity event reconstruction and science discoveries yet.

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 in earnest it is worth it to consider at least one simple probabilistic program to informally introduce a bit of syntax and relate a probabilistic programming language model denotation 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, here specifically to characterize the distribution on the return value `x`, 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 work.

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.

[**JW:** *ToDo: Make sure that we define static / dynamic graphs.*]

Although we have endeavored to make this tutorial as self-contained

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)

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 to 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 ex-

ample, 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 primitive data type such as a number, a string, or a boolean, a built-in primitive function such as `+`, or 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 $\ldots$ 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 $e_1 \ldots e_n$ 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 $\ldots$ 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 argument 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 input in the form of an expression. However, despite this relative simplicity, we will see that we can express just about 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 potentially non-halting computations are disallowed. 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.
- (`put` e_1 e_2 e_3) set replaces the element at index/key e_2 with the value e_3. Note that FOPPL primitives are pure functions, so the `set` primitive is assumed to return a modified copy of e_1 (i.e. `put` does not modify e_1 in place).

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

[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.

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 a 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

[**FW:** *By introducing [] as sugar for* `vector` *now all this let stuff is fucked up unless you make the let desugaring go first for instance, but it isn't first in the text so it's a little confusing*] [**JW:** *I've thought about this a bit. There is no expression form* `(let` e_1 e_2`)` *in the language, so the case where you would desugar* $e_1 =$ `[`v e_3`]` *to* `(vector` v e_3`)` *cannot arise. Any implementation of these desugaring rules would need to do pattern matching, but I think it is OK not to get into this.*]

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 [`v_1 e_1
⋮
v_n e_n`]`
e_{n+1} ... e_{m-1} e_m`).`

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 guaranteed-to-be-unique 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 used throughout the remainder, 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.

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 e1 ... en)
```

Once again, c must be a non-negative integer *constant* and f a procedure, primitive or user-defined. [**JW:** *Is there a reasonable primitive one might use here? One could write* `(loop 5 1 * 2)`*, which would return 32, but this seems a bit silly. Maybe we should just say 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)]
      (let [v2 (f 2 v1 a1 ... an)]
        ⋮
          (let [vc-1 (f (- c 1) vc-2 a1 ... an)]
            vc-1) ··· )))
```

where $v_0, \ldots, v_n$ and $a_0, \ldots, a_n$ are fresh variables. Note that the `loop` sugar computes an iteration over a fixed set of indices. [**FW:** *it occurs to me that the way we were desugaring before dispatched calls for the arguments to re-evaluated at each nested let level. that's not what i would expect. particularly when thinking about how to implement loop*

as a higher order procedure in the HOPPL - i've changed it, let me know what you think] [**JW:** *I think this is better. When compiling to a graph we translate* `(let [v e1] e)` *to* $e[v_1 := e_1]$*, which nullifies this optimization. However, it should definitely be more efficient in evaluation-based methods.*]

To illustrate how the loop form differs from the for 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`, which is 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 stasticial 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

```
(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.

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

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

$$\mu_k \sim \text{Normal}(0.0, 10.0), \tag{2.2}$$

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

$$z_n \sim \text{Discrete}(\pi), \qquad \text{for } n = 1, \ldots, 16, \tag{2.4}$$

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

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 pro-

```
(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

gram 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, this 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, we append 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.

[**celeste:** *This is a wall of text... Look up the "split attention effect" on Wikipedia–it currently has an integrated vs. not-integrated visual learning example. Making the reader go back and forth between the text*

```
(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 hmm-step states
    data trans-dists likes))
```

Program 2.5: FOPPL - Hidden Markov model

wall and the figure for each sentence is exhausting. A slight step up would be to put line numbers in and refer to them, but the ideal solution would be to–either before or after showing all of the code–break up the code into smaller chunks and integrate the explanation.]

2.3.3 A Bayesian Neural Network

[**JW:** *How about we reproduce the Edward example here:* ***http://edwardlib.org/tutorials/bayesian-neural-network*** *- this is also a nice way to mention that people are doing this.*]

[**ToDo:** *Harmonize text with new version of the program*]

Given that neural networks with stochastic nodes, i.e. stochastic computation graphs, are fixed dimension computation graphs, they too can be expressed in the FOPPL. In the following we denote a Bayesian approach to learning the parameters of a two layer "decoder" neural net with latent code of dimension `latent-dim`= 2, a hidden layer of dimension `hidden-dim`, and an independent and identically Gaussian

```
(defn relu [v]
  (mat-mul (mat-ge v 0.0) v))

(let [weight-prior (normal 0 1)
      W (foreach 4 []
          (foreach 2 [] (sample weight-prior)))
      b (foreach 4 [] (sample weight-prior))
      V (foreach 5 []
          (foreach 4 [] (sample weight-prior)))
      c (foreach 4 [] (sample weight-prior))]
  (foreach 3 [z [[0 1]
                 [1 0]
                 [0 1]]]
             y [[0.45 2.32 7.23 -1.40 0.01]
                [4.45 -3.2 0.78 -9.40 1.11]
                [8.10 5.13 3.90 -6.31 7.41]]
    (let [h (relu (add (mat-dot W z) b))
          mu (add (mat-dot V h) c)])
      (foreach 5 [yd y
                  mud mu]
        (observe (normal mud 1) yd)))
  [W b V c])
```

Program 2.6: FOPPL - A Bayesian Neural Network

distributed `output-dim`= 5. The program inlines three data points as before, then defined accessors for this data: `datum`, `code` (the latent code accessor), `output` (output observation accessor). A great deal of the code is boilerplate code that could be sugared away with almost the entirety of the neural net code being defined in `observe-datum` which starts with the latent code `z` and passes it through the neural net to produce the means of the observed output layer. We have assumed, in this code, a number of matrix primitive functions, e.g. `mat-exp`, `mat-mul` and , `mat-add`, etc.

While, again, the point of this exercise is not to suggest that the FOPPL is the model definition 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 encode any com-

```
# 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 model Pumps model from the BUGS examples [OpenBugs, 2009].

putation graph with a finite dimensional number of random variables. [**celeste:** *Can it be demonstrated/explained and stated somewhere that "because [explanation here], the FOPPL is sufficiently expressive to encode any computation graph with a finite dimensional number of random variables", rather than insisting that was the point of the exercise?*]

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

```
(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])
```

to say that it has the following generative model:

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

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

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

$$y_i \sim \text{Poisson}(\theta_i t_i). \tag{2.9}$$

Program 2.7 shows this model in the BUGS language. Program **??** 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 have 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. The big advantage of the FOPPL is that its programs are comparatively easy to reason about, since there are only 8 expression 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

[**JW:** *This is a bit out of place here. How about we just merge it with chapter 3?*]

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 graphical model construction case it will be desireable to have a language for purely deterministic expressions.

2.4.1 Deterministic Expressions

Some expressions in our FOPPL programs do not involve user-defined procedure calls and involve only deterministic computations, e.g. `(+ (/ 2.0 6.0) 17)`. Such deterministic "0th-order 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 here 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

and 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 ensure that expressions E cannot depend on any probabilistic choices or conditioning.

Foreshadowing, expressions in this sub-language will represent the deterministic computations required to transform the values of parent random variables in the graphical model induced by a FOPPL program to the parameters of their children.

The examples provided in this chapter should convince you that many common statistics and machine learning models and inference problems 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 Bayesian Network

Programs 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 Bayesian network G and an expression E in the deterministic sublanguage 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 Bayesian Network

We define a Bayesian network 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.

[**Zinkov:** *This feels like a place a graph should be shown for some example model*]

Before presenting a set of translation rules that can be used to compile any FOPPL program to a Bayesian network, 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)
```

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 expected return value $\mathbb{E}_{p(z \mid y)}[z]$.

In the evaluation relation $\rho, \phi, e \Downarrow G, E$, the source code of the pro-

gram 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 Bayesian network $G = (V, A, \mathcal{P}, \mathcal{Y})$ and the result expression E that this program translates to are

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

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 $(p_{\mathsf{bern}}\ z\ \texttt{0.5})$. 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 O 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 Bayesian network, in which we assign each vertex in the Bayesian network a newly generated unique symbol. However, when discussing programs in this tutorial, we will generally explicity assign program variables to sample and observe expressions to aid readability.

If Expressions in Bayesian Networks

When compiling a program to a Bayesian network, if expressions require special consideration. Before we set out to define a set of translation rules that allow us to construct a Bayesian network for a program, we will therefore first spend some time for building intuition about how we would like these translation rules to handle 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 3.1: 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 3.2: 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 can identify here [**Zinkov:** *Can a stronger equivalence be placed here?*] is that both programs define the same posterior expected return value z,

$$\mathbb{E}_{p(z,\mu\,|\,y)}[z] = \mathbb{E}_{p(z,\mu_0,\mu_1\,|\,y)}[z].$$

This equivalence would clearly no longer hold if we replaced the return value of the first program with `mu`, and that of the second program

with `(vector mu0 mu1)`. The first program would then require characterization of a bimodal posterior on $p(\mu \mid y)$, whereas the return values of the second program define a joint posterior $p(\mu_0, \mu_1 \mid y)$. However, if we are only interested in reasoning about the return value `z`, then we can say that two programs are equivalent, as long as they define the same posterior marginal

$$p(z \mid y) = \int d\mu \; p(z, \mu | y) = \int d\mu_0 d\mu_1 \; p(z, \mu_0, \mu_1 | y). \tag{3.2}$$

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 Bayesian network. 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 3.3: 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 are 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 3.4: 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 3.1, 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 3.3. 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]

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

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

$$p(y|\mu_0, \mu_1, z) = p_{\mathsf{norm}}(y; \mu_z, 1). \tag{3.3}$$

[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.

When translating our source code to a Bayesian network, 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

$$(y, \mu_0, \mu_1, z) \in (\mathbb{R} \times \mathbb{R} \times \{\texttt{nil}\} \times \{0\}) \cup (\mathbb{R} \times \{\texttt{nil}\} \times \mathbb{R} \times \{1\}). \quad (3.4)$$

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 3.4 once more. If we use eager evaluation, then this program is equivalent to Program 3.2 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 (3.3). 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 3.5: 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 here. When performing eager evaluation, we would be observing two variables 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 Bayesian network.[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 next section.

[2]The only exception to this rule is observe expressions that are conditionally independent of the program output, which implies that the Bayesian network 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.

Translation rules

Now that we have developed some intuition for how one might translate a program to a data structure that represents a Bayesian network, 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

$$\frac{top}{bottom} \tag{3.5}$$

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

$$\frac{\rho, \phi, e \Downarrow G, E}{\rho, \phi, (- e) \Downarrow G, (- E)} \tag{3.6}$$

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 each expression 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 Bayesian network:

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

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

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:

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

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 Bayesian networks: 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 Bayesian networks G_1 and G_2 use disjoint sets of vertices. This assumption always holds because every Bayesian network created by our translation uses fresh vertices, which do not appear in other networks previously generated.

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

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

As we have explained already, the Bayesian networks 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.

None of the rules for an expression e so far extends Bayesian networks 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 Bayesian networks to express random variables. The next two rules about `sample` and `observe` show this usage.

Sample We translate sample expressions using the following rule:

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

[**FW:** *I can't remember why we need the intersection of the free vars with the vertices in the existing graph*] This rule states that we translate `(sample` e`)` in three steps. First, we translate the argument e to a Bayesian network $(V, A, \mathcal{P}, \mathcal{Y})$ 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 $\bot$ 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:

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

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 $F = \texttt{(if}\ \phi\ F_1\ \texttt{1)}$ 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 Bayesian network. Translation fails when this requirement is not met. [**Zinkov:** *I think a better explanation of* FREE-VARS *is warranted*]

Procedure Call The remaining two cases are those for procedure calls, one for a user-defined procedure f and the other for a primitive procedure c. In both cases, we first translate arguments, and then the procedure call itself by looking at the definition of the invoked proce-

dure:

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

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

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

$$\text{PA}(v) := \{u : (u, v) \in A\}. \tag{3.7}$$

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

$$p(V) = \prod_{v \in V} p(v \,|\, \text{PA}(v)). \tag{3.8}$$

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

$\textsc{pa}(v)$ is nothing but the free variables in $\mathcal{P}(v)$ exclusive of v

$$\textsc{pa}(v) = \textsc{free-vars}(\mathcal{P}(v)) \setminus \{v\}. \tag{3.9}$$

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

$$p(v \mid \textsc{pa}(v)) \equiv \mathcal{P}(v). \tag{3.10}$$

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. 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 (3.8) as

$$p(V) = p(Y, X) = p(Y \mid X)p(X), \tag{3.11}$$

where

$$p(Y \mid X) = \prod_{y \in Y} p(y \mid \textsc{pa}(y)), \qquad p(X) = \prod_{x \in X} p(x \mid \textsc{pa}(x)). \tag{3.12}$$

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

$$p(X \mid Y) = p(X, Y)/p(Y), \qquad p(Y) := \int dX \, p(X, Y). \tag{3.13}$$

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

$$\gamma(X) = \psi(X)p(X). \tag{3.14}$$

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

$$\pi(X) := \gamma(X)/Z, \qquad Z := \int dX\, \gamma(X). \tag{3.15}$$

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

$$\psi(X) = \prod_{y \in Y} \psi_y(X_y), \tag{3.16}$$

where we define ψ_y and X_y as

$$\psi_y(X_y) := p(y{=}\mathcal{Y}(y) \mid \textsc{pa}(y)) \equiv \mathcal{P}(y)[y := \mathcal{Y}(y)] \tag{3.17}$$

$$X_y := \textsc{free-vars}(\mathcal{P}(y)) \setminus \{y\} = \textsc{pa}(y). \tag{3.18}$$

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

$$p_{\text{factor-dist}}(y; \lambda) := \begin{cases} \exp \lambda & y = \texttt{nil} \\ 0 & y \neq \texttt{nil} \end{cases} \tag{3.19}$$

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 optimization for our compilation procedure is to perform a partial evaluation step. This step pre-evaluates expressions E in the the target language that do not contain any free variables, since such expressions will always take on the same value in every execution of the program.

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 a number of rules for simplifying expressions. We will begin by considering the following 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 $\texttt{(if}\ E_1\ E_2\ E_3\texttt{)}$ 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 additionaly 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

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

Partial evaluation is not just a computational optimization. It can also reduce the number of edges in the Bayesian network. For example, the expression `(if true` v_1 v_2`)` nominally references two random variables v_1 and v_2. However, 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 $\ldots$ e_n`)` syntactic sugar, we can now substitute any fully deterministic expression for the number of loop iterations c. For example, we could substitute `(count data)` to 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 (nth A x1)))
      x3 (sample (discrete (nth 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 (peek xs)
        Ak (nth A k)]
    (conj 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 `(conj 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 `(peek 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

```
(nth [[0.9 0.1]
      [0.1 0.9]]
      (peek [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

```
(nth [[0.9 0.1]
      [0.1 0.9]]
      (peek (conj [v1] v2)))
```

The extra arc from v_1 to v_3 is of course not necessary here, since the expression `(peek (conj [v1] v2))` 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 procedures that return data

structures,

$$\text{EVAL}(\texttt{(vector}\ E_1\ \ldots\ E_n\texttt{)}) = \texttt{[}E_1\ \ldots\ E_n\texttt{]},$$
$$\text{EVAL}(\texttt{(hash-map}\ c_1\ E_1\ \ldots\ c_n\ E_n\texttt{)}) = \texttt{\{}c_1\ E_1\ \ldots\ c_n\ E_n\texttt{\}}.$$

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,

$$\text{EVAL}(\texttt{(conj (list}\ E_1\ \ldots\ E_n\texttt{)}\ E_0\texttt{)}) = \texttt{(list}\ E_0\ E_1\ \ldots\ E_n\texttt{)},$$
$$\text{EVAL}(\texttt{(conj [}E_1\ \ldots\ E_n\texttt{]}\ E_{n+1}\texttt{)}) = \texttt{[}E_1\ \ldots\ E_{n+1}\texttt{]},$$
$$\text{EVAL}(\texttt{(conj \{}c_1\ E_1\ \ldots\ c_n\ E_n\texttt{\}}\ \texttt{[}c_{n+1}\ E_{n+1}\texttt{]})) = \texttt{\{}c_1\ E_1\ \ldots\ c_{n+1}\ E_{n+1}\texttt{\}}.$$

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

```
(nth [[0.9 0.1]
      [0.1 0.9]]
      (peek [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

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

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

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

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

This gives us the correct dependency structure for the Bayesian network.

[**JW:** *Is this the best place for this section? Maybe we should remove this section now and merge this discussion into the preceding section. If we do this, then we should explain translations in the order if -> procedure -> sample -> observe. This way, once we get to the observe statement, we partial evaluation ensures that observed values must be constants.*]

[**HY:** *Typically we apply partial evaluation recursively. For instance, we expect the rule*

$$\begin{aligned}&\textsc{eval}(\texttt{(}\texttt{if}\ c_1\ E_2\ E_3\texttt{)}) = E_3\\&\textit{(where } c_1 \textit{ is the logical constant true)}\end{aligned}$$

to have the form

$$\begin{aligned}&\textsc{eval}(\texttt{(}\texttt{if}\ c_1\ E_2\ E_3\texttt{)}) = \textsc{eval}(E_3),\\&\textit{(where } c_1 \textit{ is the logical constant true)}\end{aligned}$$

and even to have the following form:

$$\begin{aligned}&\textsc{eval}(\texttt{(}\texttt{if}\ E_1\ E_2\ E_3\texttt{)}) = \textsc{eval}(E_3)\\&\textit{(where } \textsc{eval}(E_1) \textit{ is the logical constant true)}\end{aligned}$$

But such a recursive application is not needed because we may assume that all subexpressions are partially evaluated already.]

[**HY:** *I did not remove the list type constructor* `list` *because* $(E_1\ E_2\ \ldots\ E_n)$ *means the application of the function* E_1 *to* $E_2, E_3, \ldots, E_n$*, not a list.*]

3.3 Gibbs Sampling

[**JW:** *ToDos: (1) Find a way to eliminate notation of the form* $p(V \setminus \{x\}, x')$*, which interleaves the use of* x *as a symbol and* x' *as a value. (2) Define* $q(X' \mid V)$ *rather than* $q(X' \mid X)$ *everywhere. (3) Make sure we define conditional probabilities correctly. For example* $p(V_x)$ *should probably read* $p(V_x \mid \textsc{pa}(V_x))$ *or simply* $\prod_{v \in V_x} p(v \mid \textsc{pa}(v))$]

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.

Markov chain Monte Carlo (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 the 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 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 $q(X' \mid X)$, the MH algorithm simulates a candidate X' from $q(X' \mid X)$ conditioned on the value of the current sample X, and then evaluates the acceptance probability

$$\alpha(X', X) = \min\left\{1, \frac{p(Y, X')q(X \mid X')}{p(Y, X)q(X' \mid X)}\right\}. \tag{3.20}$$

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

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

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

$$p(x \mid Y, X \setminus \{x\}) = p(x \mid V \setminus \{x\}). \tag{3.21}$$

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 $p(x \mid V \setminus \{x\})$.

Given our compiled graph $(V, A, \mathcal{P}, \mathcal{Y})$, how can we compute the acceptance probability in Equation (3.22)? When we update a single variable x using a kernel $q(x' \mid X)$, this probability takes the form

$$\alpha(x', x) = \min\left\{1, \frac{p(V \setminus \{x\}, x')q(x \mid X')}{p(V \setminus \{x\}, x)q(x' \mid X)}\right\}. \tag{3.22}$$

From a computational point of view, the important thing to note is that many terms in this ratio will actually cancel out. Conditional densities $p(v \mid \textsc{pa}(v)) \equiv \mathcal{P}(v)$ include the value of the node x only if $v = x$, or $x \in \textsc{pa}(v)$, which equates to the condition

$$x \in \textsc{free-vars}(\mathcal{P}(v)). \tag{3.23}$$

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

$$\begin{aligned} V_x &:= \{v : x \in \textsc{free-vars}(\mathcal{P}(v))\}, \\ &= \{x\} \cup \{v : x \in \textsc{pa}(v)\}, \end{aligned} \tag{3.24}$$

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, which means that we can simplify the acceptance probability α to

$$\alpha(x', x) = \min\left\{1, \frac{p(V_x \setminus \{x\}, x')q(x \mid X')}{p(V_x \setminus \{x\}, x)q(x' \mid X)}\right\}. \tag{3.25}$$

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

From an implementation point of view, we can compute the acceptance probability by evaluating the expressions $\mathcal{P}(v)$ for each $v \in V_x$, 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,

$$X = (x_1, \ldots, x_N), \qquad \mathcal{X} = [x_1 \mapsto c_1, \ldots, x_N \mapsto c_N], \tag{3.26}$$

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_x as

$$p(V_x = \mathcal{V}_x) = \prod_{v \in V_x} \textsc{eval}(\mathcal{P}(v)[V := \mathcal{V}]). \tag{3.27}$$

Similarly, we can evaluate $p(V_x = \mathcal{V}'_x)$ by defining $\mathcal{V}'_x = \mathcal{V}_x[x \mapsto c']$, where c' is the candidate value.

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

$$\mathcal{Q} := [x_1 \mapsto E_1, \ldots, x_N \mapsto E_N]. \tag{3.28}$$

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 $q(x' \mid X = \mathcal{X})$ and $q(x \mid X = \mathcal{X}')$. To do so, we first evaluate the expression to a distribution

$$q = \textsc{eval}(\mathcal{Q}(x)[X := \mathcal{X}]). \tag{3.29}$$

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

$$x' = \textsc{sample}(q), \qquad q(x' \mid X) = \textsc{log-prob}(q). \tag{3.30}$$

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

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($x, \mathcal{X}', \mathcal{X}$)
  $q \leftarrow \text{EVAL}(\mathcal{Q}(x)[X := \mathcal{X}])$
  $q' \leftarrow \text{EVAL}(\mathcal{Q}(x)[X := \mathcal{X}'])$
  $\log \alpha \leftarrow \text{LOG-PROB}(q', \mathcal{X}(x)) - \text{LOG-PROB}(q, \mathcal{X}'(x))$
  $V_x \leftarrow \{v : x \in \text{FREE-VARS}(\mathcal{P}(v))\}$
  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**
    $q \leftarrow \text{EVAL}(\mathcal{Q}(x)[X := \mathcal{X}])$
    $\mathcal{X}' \leftarrow \mathcal{X}$
    $\mathcal{X}'(x) \leftarrow \text{SAMPLE}(q)$
    $\alpha \leftarrow \text{ACCEPT}(x, \mathcal{X}', \mathcal{X})$
    $u \sim \text{Uniform}(0, 1)$
    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 $\mathcal{X}^{(1)}, \ldots \mathcal{X}^{(S)}$

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 $\mathcal{P}(v) = \text{SCORE}(E, v)$, and finally define $\mathcal{Q}(v) = E$. In this type of proposal $q(x' \mid X) = p(x \mid \text{PA}(x))$, which means that the acceptance ratio simplifies further to

$$\alpha(x', x) = \min\left\{1, \frac{p(V_x \setminus \{x\} \mid x')}{p(V_x \setminus \{x\} \mid x)}\right\}. \tag{3.31}$$

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 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

$$\alpha(x, x') = \min\left\{1, \frac{p(V_x \setminus \{x\}, x')}{p(V_x \setminus \{x\}, x)}\right\}. \tag{3.32}$$

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. In 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 distribution $q(S \mid X \setminus S)$. This proposal has an acceptance ratio of

$$\alpha(S, S') = \min\left\{1, \frac{p(V_S \setminus S, S')q(S \mid X)}{p(V_S \setminus S, S)q(S' \mid X)}\right\}, \qquad V_S := \cup_{x \in S} V_x.$$

The "trivial" choice of proposal distribution used before — 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].

In the next section, we consider Hamiltonian Monte Carlo proposals, which incorporate gradient information to provide efficient high-dimensional 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 (3.15), in which each variable $x \in X$ is continuous. This unnormalized density is traditionally re-expressed by defining a *potential energy* function

$$U(X) := -\log \gamma(X). \tag{3.33}$$

With this definition we can write

$$\pi(X) = \frac{1}{Z} \exp\left\{-U(X)\right\}. \tag{3.34}$$

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

$$H(X, R) = U(X) + K(R). \tag{3.35}$$

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

$$\begin{aligned}\frac{dX}{dt} &= \nabla_R H(X, R) = M^{-1} R, \\ \frac{dR}{dt} &= -\nabla_X H(X, R) = -\nabla_X U(X).\end{aligned} \tag{3.36}$$

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 (3.36), 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 a FOPPL program is the computation of the gradient

$$\nabla U(X) = -\nabla \log \gamma(X). \tag{3.37}$$

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. We will 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, 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

$$\begin{aligned}\nabla_x U(X) &= -\frac{\partial \log p(Y, X)}{\partial x} \\ &= -\sum_{x' \in X} \frac{\partial \log p(x' | \mathrm{PA}(x'))}{\partial x} - \sum_{y \in Y} \frac{\partial \log p(y | \mathrm{PA}(y))}{\partial x}.\end{aligned} \tag{3.38}$$

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 $Y = \{y_1, \ldots, y_M\}$

$$E_X := \texttt{(+ (log } \mathcal{P}(x_1)\texttt{) ... (log } \mathcal{P}(x_N)\texttt{))}$$
$$E_Y := \texttt{(+ (log } \mathcal{P}(y_1)[y_1 := \mathcal{Y}(y_1)]\texttt{) ... (log } \mathcal{P}(y_M)[y_M := \mathcal{Y}(y_M)]\texttt{))}$$
$$E_U := \texttt{(* -1.0 (+ } E_X \ E_Y\texttt{))}$$

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

$$U(X = \mathcal{X}) := \text{EVAL}(E_U[X := \mathcal{X}]). \tag{3.39}$$

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

$$\mathcal{V} = [v_1 \mapsto c_1, \ldots, v_D \mapsto c_D], \tag{3.40}$$

Our goal is now to to define a gradient operator

$$\text{GRAD}\left(\text{EVAL}\left(F[V := \mathcal{V}]\right)\right), \tag{3.41}$$

which computes the map of partial derivatives

$$\mathcal{G} := \left[v_1 \mapsto \left.\frac{\partial F(V)}{\partial v_1}\right|_{V=\mathcal{V}}, \ldots, v_D \mapsto \left.\frac{\partial F(V)}{\partial v_D}\right|_{V=\mathcal{V}} \right]. \tag{3.42}$$

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
$$\frac{\partial E}{\partial v_i} = 0. \tag{3.43}$$

2. Variables $E = v$ have derivatives
$$\frac{\partial E}{\partial v_i} = \begin{cases} 1 & v = v_i, \\ 0 & v \neq v_i. \end{cases} \tag{3.44}$$

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,
$$\frac{\partial E}{\partial v_i} = \begin{cases} \partial E_2 \,/\, \partial v_i & c_1' = \texttt{true} \\ \partial E_3 \,/\, \partial v_i & c_1' = \texttt{false} \end{cases} \tag{3.45}$$

4. For primitive procedure applications $E =$`(`f E_1 $\ldots$ E_n`)` we apply the chain rule
$$\frac{\partial E}{\partial v_i} = \sum_{j=1}^{n} \frac{\partial f(v_1', \ldots, v_n')}{v_j'} \frac{\partial E_j}{\partial v_i}. \tag{3.46}$$

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 replace

Algorithm 2 Primitive function lifting for reverse-mode AD.

```
1: function UNBOX(c̃)
2:     if c̃ = (c, _) then
3:         return c                                  ▷ Unpack value from c̃
4:     else
5:         return c̃                                  ▷ Return value as is
6: function LIFT-AD(f, ∇f, n)
7:     function f̃(c̃_1, ..., c̃_n)
8:         c_1, ..., c_n ← UNBOX(c̃_1), ..., UNBOX(c̃_n)
9:         c ← f(c_1, ..., c_n)
10:        ċ_1, ..., ċ_n ← ∇f(c_1, ..., c_n)
11:        return (c, ((c̃_1, ..., c̃_n), (ċ_1, ..., ċ_n)))
12:    return f̃
```

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

$$\tilde{c} = (c, ((\tilde{c}_1, \ldots, \tilde{c}_n), (\dot{c}_1, \ldots, \dot{c}_n))), \tag{3.47}$$

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 3.44), 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 3.43), 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 3.46), 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

$$\tilde{U} = \textsc{eval}(E_U[V := \mathcal{V}]), \tag{3.48}$$

$$\mathcal{G} = \textsc{grad}(\tilde{U}). \tag{3.49}$$

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 left out of consideration.

Some of these implementation considerations are common to all HMC implementations. Algorithm 4 performs numerical integration using a leapfrog scheme, which discretizes the trajectory for the posi-

tion 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 in practice? To be safe, a program should not contain any observe expressions inside if branches. We can detect this condition in our graph compilation procedure, since it leads to an expression $\mathcal{P}(y)$ of the form `(if` ϕ E `1)`, where E is the expression for the likelihood of the observed variable and ϕ is an expression that cannot be partially evaluated, which is to say that it has free variables. Moreover all primitive procedures in the program that return doubles should have a well-defined derivative for all possible inputs. This can be a tricky condition for functions such as abs(x), for which the derivative is defined at all points except $x = 0$.

What is not necessarily the case is that a differentiable program must *only* make use of primitives that return numbers. In Algorithm 2, we assume that f is a primitive with a real-valued output and real-valued inputs. In practice, we can also perform reverse-mode AD on programs which incorporate non-differentiable primitives. However such primitives can once again lead to discontinuities. As an example, suppose that we want to take the gradient of an expression of the form `(get [`$\tilde{c}_2$ $\tilde{c}_3$`]` c_1`)`. This case is analogous to the gradient of an if expression GRAD(`(if` c_1 $\tilde{c}_2$ $\tilde{c}_3$`)`), which evaluates to either GRAD($\tilde{c}_2$) or GRAD($\tilde{c}_3$), depending on the value of c_1. In the same way, the gradient GRAD(`(get [`$\tilde{c}_1$ $\ldots$ $\tilde{c}_n$`]` k`)`) will evaluate to GRAD($\tilde{c}_k$). In general, we can interleave non-differentiable and differentiable operations in this manner, as long as we represent all differentiable values using boxed data structures, but when using HMC this can result in non-differentiable density functions. If one wanted to guarantee that a density is differentiable, one could use partial evalution to eliminate all examples of if expressions and data structure assessors in which the selected element or branch is determinable at compile time. If any if expressions or data structure accessors remain after partial evaluation, then one could issue a warning to this effect.

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

$$\gamma(X) := \prod_{f \in F} \psi_f(X_f), \tag{3.50}$$

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

$$X_f := \{x : (x, f) \in A\}. \tag{3.51}$$

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 other words we could define

$$\gamma(X) := \prod_{v \in V} \psi_v(X_v), \tag{3.52}$$

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

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

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

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

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

Figure 3.1: 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))`.

observed or unobserved,

$$\psi_v(X_v) \equiv \begin{cases} \mathcal{P}(v)[v := \mathcal{Y}(v)], & v \in \mathrm{dom}(\mathcal{Y}), \\ \mathcal{P}(v), & v \notin \mathrm{dom}(\mathcal{Y}). \end{cases} \tag{3.53}$$

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

$$\psi_h(X_h) := \psi_f(X_f)\psi_g(X_g), \qquad X_h := X_f \cup X_g. \tag{3.54}$$

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].[3] 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

[3] [**JW:** *and Figaro/Factorie?*]

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 3.1. 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 3.50.

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 and A is a set of undirected edges between variables and factors. The set of variables is

identical to the set of unobserved variables (i.e. the set of sample forms) in the corresponding directed graph

$$X := X^{\mathrm{dg}} = V^{\mathrm{dg}} \setminus \mathrm{dom}(\mathcal{Y}^{\mathrm{dg}}). \tag{3.55}$$

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

$$A := \{(v, f) : (v, v_f) \in A^{\mathrm{dg}}\}. \tag{3.56}$$

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 $\mathcal{P}^{\mathrm{dg}}(v_f)$,

$$\Psi(f) := \begin{cases} \mathcal{P}^{\mathrm{dg}}(v_f)[v_f := \mathcal{Y}^{\mathrm{dg}}(v_f)], & v_f \in \mathrm{dom}(\mathcal{Y}^{\mathrm{dg}}), \\ \mathcal{P}^{\mathrm{dg}}(v_f), & v_f \notin \mathrm{dom}(\mathcal{Y}^{\mathrm{dg}}). \end{cases} \tag{3.57}$$

This defines the usual correspondance 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 $\Psi(f)$,

$$\psi_f(X_f) \equiv \Psi(f), \qquad X_f = \textsc{free-vars}(\Psi(f)). \tag{3.58}$$

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 $(p_{\mathsf{dirac}}\ v\ c)$ or $(p_{\mathsf{dirac}}\ c\ v)$. 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 $(p_{\mathsf{dirac}}\ v_1\ v_2)$ 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

$$\gamma(s_1, s_2) = \left(p_{\mathsf{norm}}(s_1; 0, 1)\, p_{\mathsf{norm}}(s_2; 0, 1)\right)^{I[(s_1 - s_2) > \epsilon]}. \tag{3.59}$$

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

$$\Psi = \begin{bmatrix} f_{s_1} \mapsto (p_{\mathsf{norm}}\ s_1\ 0.0\ 1.0), \\ f_{s_2} \mapsto (p_{\mathsf{norm}}\ s_2\ 0.0\ 1.0), \\ f_{\delta} \mapsto (p_{\mathsf{dirac}}\ \delta\ (-\ s_1\ s_2)), \\ f_{w} \mapsto (p_{\mathsf{dirac}}\ w\ (>\ \delta\ 0.1)), \\ f_{y} \mapsto (p_{\mathsf{dirac}}\ \mathtt{true}\ w) \end{bmatrix}. \tag{3.60}$$

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 3.1. 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

$$\Psi = \begin{bmatrix} f_{s_1} \mapsto (p_{\mathsf{norm}}\ \texttt{0.0}\ \texttt{1.0}), \\ f_{s_2} \mapsto (p_{\mathsf{norm}}\ \texttt{0.0}\ \texttt{1.0}), \\ f_{\delta} \mapsto (p_{\mathsf{dirac}}\ \delta\ (\texttt{-}\ s_1\ s_2)), \\ f_{w} \mapsto (p_{\mathsf{dirac}}\ \texttt{true}\ (\texttt{>}\ \delta\ \texttt{0.1})) \end{bmatrix}. \tag{3.61}$$

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 `(>` v_1 v_2`)` or `(-` 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 factor graphs are amenable to 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

$$\pi(X) := \gamma(X)/Z^{\pi}, \qquad \gamma(X) := \prod_{f \in F} \psi_f(X_f). \tag{3.62}$$

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

$$q(X) := \frac{1}{Z^q} \prod_{f \in F} \phi_f(X_f). \tag{3.63}$$

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

$$\operatorname*{argmin}_q D_{\mathrm{KL}}\left(\pi(X) \,||\, q(X)\right) = \operatorname*{argmin}_q \int \pi(X) \log \frac{\pi(X)}{q(X)} dX, \tag{3.64}$$

EP algorithms iteratively update one factor ϕ_f at a time by performing the following computation

- Choose a factor f to refine
- Define the tilted distribution

$$\pi_f(X) := \gamma_f(X)/Z_f, \qquad \gamma_f(X) := \frac{\psi_f(X_f)}{\phi_f(X_f)} q(X). \tag{3.65}$$

- Update the factor by minimizing the KL divergence

$$\phi_f = \operatorname*{argmin}_{\phi_f} D_{\mathrm{KL}}\left(\pi_f(X) \,||\, q(X)\right). \tag{3.66}$$

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 for each of the factors in $q(X)$

$$\phi_f(X_f) := \prod_{x \in X_f} \phi_{f \to x}(x). \tag{3.67}$$

We refer to the unnormalized potential $\phi_{f \to v}(x)$ as the message from factor f to the variable x. We assume the exponential form

$$\phi_{f \to x}(x) = \exp[\lambda_{f \to x}^\top t(x)], \tag{3.68}$$

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

$$\begin{aligned} q(x) &= \frac{1}{Z_x^q} \prod_{f : x \in V_f} \phi_{f \to x}(x), \\ &= h(x) \exp\left(\lambda_x t(x) - a(\lambda_x)\right), \end{aligned} \tag{3.69}$$

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

$$\lambda_x = \sum_{f:x\in X_f} \lambda_{f\to x}. \tag{3.70}$$

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

$$Z^q := \prod_{x\in X} Z_x^q, \qquad Z_x^q := \int dx \prod_{f:x\in X_f} \phi_{f\to x}(x), \tag{3.71}$$

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

$$Z_x^q = \exp\left(a(\lambda_x)\right) = \exp\left(a\left(\textstyle\sum_{f\,:\,x\in X_f} \lambda_{f\to x}\right)\right). \tag{3.72}$$

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

$$\nabla_{\lambda_x} a(\lambda_x) = \mathbb{E}_{q(x)}[t(x)]. \tag{3.73}$$

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

$$D_{\mathrm{KL}}\left(\pi_f(X) \,||\, q(X)\right) = \log \frac{Z^q}{Z_f} + \mathbb{E}_{\pi_f(X)}\left[\log \frac{\psi_f(X_f)}{\phi_f(X_f)}\right]. \tag{3.74}$$

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

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

When we substitute the message $\phi_{f\to x}(x)$ from Equation 3.68, the normalizing constant $Z_x^q(\lambda_x)$ from Equation 3.72, and apply Equation 3.73, then we obtain the moment matching condition

$$\begin{aligned}\mathbb{E}_{q(x)}[t(x)] &= \nabla_{\lambda_{f\to x}} \mathbb{E}_{\pi_f(X)}[\log \phi_{f\to x}(x)], \\ &= \nabla_{\lambda_{f\to x}} \mathbb{E}_{\pi_f(X)}[\lambda_{f\to x}^\top t(x)], \\ &= \mathbb{E}_{\pi_f(X)}[t(x)].\end{aligned} \tag{3.75}$$

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 (3.65)
  $Z_f \leftarrow \int dX\, \gamma_f(X)$ ▷ Equation (3.77)
  for x in X_f **do**
    $\bar{t} \leftarrow 1/Z_f \int dX\, \gamma_f(X)t(x)$ ▷ Equation (3.79)
    $\lambda_x^* \leftarrow$ MOMENT-MATCH($\bar{t}$) ▷ Equation (3.75)
    $\lambda_{f\to x} \leftarrow \lambda_x^* - \sum_{f'\neq f\,:\,x\in X_{f'}} \lambda_{f'\to x}$ ▷ Equation (3.76)
  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**
    $\log Z_x^q \leftarrow a(\lambda_x)$ ▷ Equation (3.72)
  $\log Z^\pi \leftarrow \sum_f \log Z_f + \sum_x \log Z_x^q$ ▷ Equation (3.80)
  return $\lambda, \log Z^\pi$

If we assume that the parameters λ_x^* satisfy the condition above, then we can use Equation 3.70 to define the update for the message $\phi_{f\to x}$

$$\lambda_{f\to x} \leftarrow \lambda_x^* - \sum_{f'\neq f\,:\,x\in X_{f'}} \lambda_{f'\to x}. \tag{3.76}$$

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,

$$\begin{aligned} Z_f = \int dX \frac{\psi_f(X_f)}{\phi_f(X_f)} q(X) &= \int dX_f\, \frac{\psi_f(X_f)}{\phi_f(X_f)} \prod_{x\in X_f} \frac{1}{Z_x^q} \prod_{f'\,:\,x\in V_{f'}} \phi_{f'\to x}(x), \\ &= \int dX_f\, \psi_f(X_f) \prod_{x\in X_f} \frac{1}{Z_x^q} \phi_{x\to f}(x). \end{aligned} \tag{3.77}$$

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

$$\phi_{x \to f}(x) := \prod_{x \in X_f} \prod_{f' \neq f \,:\, x \in X_{f'}} \phi_{f' \to x}(x). \tag{3.78}$$

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

$$\bar{t} = \mathbb{E}_{\pi_f(V)}[t(v)] = \frac{1}{Z_f} \int dV_f \; t(x) \psi_f(X_f) \prod_{x \in X_f} \frac{1}{Z_x^q} \phi_{x \to f}(x). \tag{3.79}$$

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 computer $\bar{t}$, 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,

$$Z^\pi \simeq \prod_{f \in F} Z_f \prod_{x \in X} Z_x^q. \tag{3.80}$$

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 Systems like 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 (3.77).

3. For every variable message $\phi_{f\to x}(x)$ we need to be able to compute the sufficient statistics $\bar{t}$ in Equation (3.79).

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 the previous Section

```
(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])
```

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

$$\Psi = \begin{bmatrix} f_1 \mapsto (p_{\text{norm}}\ \texttt{0.0}\ \texttt{1.0}), \\ f_2 \mapsto (p_{\text{norm}}\ \texttt{0.0}\ \texttt{1.0}), \\ f_3 \mapsto (p_{\text{dirac}}\ \delta\ (\texttt{-}\ s_1\ s_2)), \\ f_4 \mapsto (p_{\text{dirac}}\ \texttt{true}\ (\texttt{>}\ \delta\ \texttt{0.1})) \end{bmatrix}. \tag{3.81}$$

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 (3.77) and Equation (3.79). We will start with the case of the integral for Z_f when updating the factor f_3,

$$Z_f = \frac{1}{Z^q_{s_1} Z^q_{s_2} Z^q_\delta} \int ds_1\, ds_2\, d\delta\; I[\delta = s_1 - s_2] \\ \phi_{s_1 \to f_3}(s_1)\phi_{s_2 \to f_3}(s_2)\phi_{\delta \to f_3}(\delta). \tag{3.82}$$

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 should be able to 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

$$\begin{aligned} Z_f &= \frac{1}{Z^q_\delta} \int_{-\infty}^{\infty} d\delta \;\; I[\delta > 0.1]\; \phi_{\delta \to f_4}(\delta), \\ &= \frac{1}{Z^q_\delta} \int_{0.1}^{\infty} d\delta\; \phi_{\delta \to f_4}(\delta). \end{aligned} \tag{3.83}$$

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

$$Z_f = \frac{1}{Z^q_{s_1} Z^q_{s_2}} \int ds_1\, ds_2 I[s_1 - s_2 > 0.1] \\ \phi_{s_1 \to f_{3+4}}(s_1)\phi_{s_2 \to f_{3+4}}(s_2). \tag{3.84}$$

In general, integrals involving constraints over multiple deterministic operations will be much harder to compute in an automated manner than integrals involving constraints over atomic operations.

In general, deterministic factors take the form $(p_{\mathsf{dirac}}\ v_0\ E)$ where E is a target language expression in which all sub-expressions must be variable references,

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

For each primitive c, we need to implement the integral for Z_f in Equation (3.77) and the integrals for the sufficient statistics in Equation (3.79). Note that these integrals do not 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 $\texttt{(-}\ v_1\ v_2\texttt{)}$ 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.

In the case of if expressions $\texttt{(if}\ v_1\ v_2\ v_3\texttt{)}$, messages are typically calculated using constructs 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. [**JW:** *ToDo: discuss gates in more detail, or perhaps ask John Winn / Tom Minka to contribute.*]

Finally, EP needs to implement integrals for sample and observe expressions, where potential have the form $\Psi(f) = \texttt{(}p\ v_0\ v_1\ \ldots\ v_n\texttt{)}$ and $\Psi(f) = \texttt{(}p\ c_0\ v_1\ \ldots\ v_n\texttt{)}$ respectively. For these factors 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. [**JW:** *ToDo: Ask around to hear what distribution types Infer.NET can handle.*]

4

Evaluation-based Inference

[**JW:** *Discuss: looking at this section in its entirety, I am considering getting rid of the global variables by threading state σ through the evaluator so we get $c, \sigma' \leftarrow \text{EVAL}(e, \sigma, \ell)$ everywhere. I originally thought this would be more cumbersome, but upon reflecting further I feel that it would make almost all algorithms less confusing and easier to read. Alternatively, if we are keeping global state, then we should perhaps change the syntax "$\log W \leftarrow 0$" to "**global** $\log W \leftarrow 0$" when initializing / changing global variables so it is unambiguous whether we are modifying a global variable or simply a local variable with the same name.*]

[**JW:** *Discuss: In the context of graph compilation we use $V = X \cup Y$ to sets of vertices, in which each vertex is a variable v. In this section we use maps from variables to values to describe traces, i.e. $V = X \oplus Y$, where $\mathcal{X} = dom(X)$ and $\mathcal{Y} = dom(Y)$ are used to refer to the set of variables that is instantiated in a trace. It might be better to introduce a different notation, e.g. $\bar{X}$ so that we can write $X = dom(\bar{X})$.*]

[**JW:** *ToDo: Once we decide on how to dedal with notation for traces, we should add some discussion of this up front in the intro that indicates*

that a probabilistic program is something that we will run to instantiate a set of random variables, which we refer to as a trace, but also specify that it is not always necessary to explicitly represent this data structure when performing inference.]

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. 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 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 Bayesian network. 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 approximate the posterior expectation $\mathbb{E}_{p(X|Y)}[r(X)]$ on the return value $r(X)$ of a program with an average over 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 ensure that generating samples is easier,

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

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].

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 Bayesian network as described in Section 3.1. We could then implement likelihood weighting by implementing the following steps:

1. For each $x \in X$: sample from the prior $x^l \sim p(x \,|\, \text{PA}(x))$.
2. For each $y \in Y$: calculate the weights $W^l_y = p(y \,|\, \text{PA}(y))$.
3. Return the weighted average over program return values $r(X^l)$

$$E_{p(X|Y)}[r(X)] \simeq \sum_{l=1}^{L} \frac{W^l}{\sum_{k=1}^{L} W^k} r(X^l), \qquad W^l := \prod_{y \in Y} W^l_y.$$

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^l_y 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,

Algorithm 6 Base cases for evaluation of a FOPPL program.

```
1:  global ρ                                              ▷ Procedure definitions
2:  function EVAL(e, σ, ℓ)
3:      match e
4:          case (sample d)
5:              ...                                       ▷ Algorithm-specific
6:          case (observe d y)
7:              ...                                       ▷ Algorithm-specific
8:          case c
9:              return c, σ
10:         case v
11:             return ℓ(v), σ
12:         case (let [v1 e1] e0)
13:             c1, σ ← EVAL(e1, σ, ℓ)
14:             return EVAL(e0, σ ℓ[v1 ↦ c1])
15:         case (if e1 e2 e3)
16:             e1', σ ← EVAL(e1, σ, ℓ)
17:             if e1' then
18:                 return EVAL(e2, σ, ℓ)
19:             else
20:                 return EVAL(e3, σ, ℓ)
21:         case (e0 e1 ... en)
22:             for i in 1, ..., n do
23:                 ci, σ ← EVAL(ei, σ, ℓ)
24:             match e0
25:                 case f
26:                     (v1, ..., vn), e0' ← ρ(f)
27:                     return EVAL(e0', σ, ℓ[v1 ↦ c1, ..., vn ↦ cn])
28:                 case c
29:                     return c(c1, ..., cn), 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.

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 calculate the likelihood as a side-effect of running the program. To do so, we initalize a global variable $\log W = 0$, 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[1] the expression e against each of the 6 non-probabilistic expression forms. These forms are the 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

[1] `https://en.wikipedia.org/wiki/Pattern_matching`

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[2].

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 global 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 fo 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.

[2] `http://norvig.com/lispy.html`

Algorithm 7 Evaluation-based likelihood weighting

```
1:  global ρ, e                                          ▷ Program procedures, body
2:  function EVAL(e, σ, ℓ)
3:      match e
4:          case (sample e)
5:              d, σ ← EVAL(e, σ, ℓ)
6:              return SAMPLE(d), σ
7:          case (observe e_1 e_2)
8:              d_1, σ ← EVAL(e_1, σ, ℓ)
9:              c_2, σ ← EVAL(e_2, σ, ℓ)
10:             σ(log W) ← σ(log W)+ LOG-PROB(d_1, c_2)
11:             return c_2, σ
12:         ...                              ▷ Base cases (as in Algorithm 6)
13: function LIKELIHOOD-WEIGHTING(L)
14:     σ ← [log W ↦ 0]
15:     for l in 1, ..., L do                                ▷ Initialize state
16:         r^l, σ^l ← EVAL(e, σ, [ ])                       ▷ Run program
17:         log W^l ← σ(log W)                               ▷ Store log weight
18:     return ((r^1, log W^1), ..., (r^L, log W^L))
```

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

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,

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

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

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

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

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

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.

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 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 subexpression 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 Bayesian network. 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 $z^1, \ldots, z^S$ 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 $X^{\mathrm{s}} \leftarrow X$.

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 reuse 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.

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 $(r^1, \ldots, r^S)$

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 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 3.4, 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:

Algorithm 9 Acceptance ratio for single-site proposals

1: **function** ACCEPT($x_0, \mathcal{X}', \mathcal{X}, \log \mathcal{P}', \log \mathcal{P}$)
2: $X'^{\text{sampled}} \leftarrow \{x_0\} \cup (\text{dom}(\mathcal{X}') \setminus \text{dom}(\mathcal{X}))$
3: $X^{\text{sampled}} \leftarrow \{x_0\} \cup (\text{dom}(\mathcal{X}) \setminus \text{dom}(\mathcal{X}'))$
4: $\log \alpha \leftarrow \log |\text{dom}(\mathcal{X})| - \log |\text{dom}(\mathcal{X}')|$
5: **for** v **in** $\text{dom}(\log \mathcal{P}') \setminus X'^{\text{sampled}}$ **do**
6: $\log \alpha \leftarrow \log \alpha + \log \mathcal{P}'(v)$
7: **for** v **in** $\text{dom}(\log \mathcal{P}) \setminus X^{\text{sampled}}$ **do**
8: $\log \alpha \leftarrow \log \alpha - \log \mathcal{P}(v)$
9: **return** α

- 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 $\mathcal{X}'(x) \leftarrow \mathcal{X}(x)$.
  - 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 $X' = \mathrm{dom}(\mathcal{X}')$ and $Y' = \mathrm{dom}(\mathcal{P}') \setminus X'$.

To calculate the probability $q(X'|X, x_0)$ we decompose the set of variables $X' = X'^{\mathrm{sampled}} \cup X'^{\mathrm{reused}}$ 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

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

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

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

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

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

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}

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

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

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

Putting all the terms together, the acceptance ratio becomes:

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

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

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

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

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

Figure 4.2: Addressing transformation for FOPPL programs.

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 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.

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)
```

If needed, unique addresses can be constructed dynamically at run time. We will get back to this in Section **??**. 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 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.

Algorithm 11 Single-site Metropolis Hastings

global ρ, e
function EVAL(e, σ, ℓ)
    ... ▷ As in Algorithm 10
function ACCEPT($x_0, \mathcal{X}', \mathcal{X}, \log \mathcal{P}', \log \mathcal{P}$)
    ... ▷ As in Algorithm 9
function SINGLE-SITE-MH(S)
    $\sigma_0 \leftarrow [x_0 \leftarrow \texttt{nil}, \mathcal{C} \mapsto, [\,], \mathcal{X} \mapsto [\,], \log \mathcal{P} \mapsto [\,]]$
    $r, \sigma \leftarrow$ EVAL(e, σ_0, $[\,]$)
    for s **in** $1, \ldots, S$ **do**
        $v \sim$ UNIFORM($\mathrm{dom}(\sigma(\mathcal{X}))$)
        $\sigma' \leftarrow \sigma_0[x_0 \mapsto v, \mathcal{C} \mapsto \sigma(\mathcal{X})]$
        $r', \sigma' \leftarrow$ EVAL(e, σ', $[\,]$)
        $u \sim$ UNIFORM-CONTINUOUS$(0, 1)$
        $\alpha \leftarrow$ ACCEPT$(x_0, \sigma'(\mathcal{X}), \sigma(\mathcal{X}), \sigma'(\log \mathcal{P}), \sigma(\log \mathcal{P}))$
        if $u < \alpha$ **then**
            $r, \sigma \leftarrow r', \sigma'$
        $r^s \leftarrow r$
    return $(r^1, \ldots, r^S)$

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 $\sigma(x_0)$, 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 according to construct a proposal and either accept or reject according to the ratio defined in Algorithm 9.

4.3 Sequential Monte Carlo Methods

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. In Section **??** we will define intermediate densities $\gamma_n(X_n)$ as versions of the program that terminate early, and therefore evaluate a subset of sample and observe expressions.

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

$$\text{dom}(X_1) \subseteq \text{dom}(X_2) \subseteq \ldots \subseteq \text{dom}(X_N) = \text{dom}(X), \tag{4.29}$$

$$\text{dom}(Y_1) \subseteq \text{dom}(Y_2) \subseteq \ldots \subseteq \text{dom}(Y_N) = \text{dom}(Y). \tag{4.30}$$

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

$$W^l_{n\backslash n-1} = \frac{p(Y^l_n \,|\, X^l_n)}{p(Y^{a^l_{n-1}}_{n-1} \,|\, X^{a^l_{n-1}}_{n-1})} = \prod_{y \in Y^l_{n\backslash n-1}} p(y \,|\, X^l_n), \tag{4.31}$$

where $Y^l_{n\backslash n-1}$ is the set difference between the observed variables at generation n and the observed variables at generation $n-1$.

$$Y^l_{n\backslash n-1} = \mathrm{dom}(\mathcal{Y}^l_n) \setminus \mathrm{dom}(\mathcal{Y}^{a^l_{n-1}}_{n-1}).$$

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 $\textsc{propose}(\mathcal{X}_{n-1}, y_n)$. This function evaluates the program that truncates at the observe expression with adress 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

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(yr) then
14:                 error RESAMPLE-BREAKPOINT( )
15:             return c, σ
16:         ...                          ▷ Base cases (as in Algorithm 6)
17: function PROPOSE(X, y)
18:     σ ← [yr ↦ 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)
```

$\log \Lambda_n = \log p(Y_n|X_n)$. To construct the proposal for the final generation we will call PROPOSE($\mathcal{X}_{N-1}$, `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 global variable $\log \Lambda$ as we have done with likelihood weighting. When we reach the observe expression with a speci-

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$(\mathcal{X}_{N-1}^{a_{N-1}^l}, \texttt{nil})$
            $\log W_n^l \leftarrow \log \Lambda_n^l - \log \Lambda_{n-1}^{a_{n-1}^l} + \log \hat{Z}_{n-1}$
    return $((r^1, \log W_N^1), \ldots, (r^L, \log W_N^L))$

fied symbol 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.

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

$$\textsc{log-mean-exp}(\log W_{n-1}^{1:L}) = \log \left(\frac{1}{L} \sum_{l=1}^{L} W_{n-1}^{l} \right). \tag{4.33}$$

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 **??**, 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 $r(X;\lambda) := \log\big(p(Y,X)/q(X;\lambda)\big)$. 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 vector λ_v, then the variance reduction constant $\hat{b}_{v,d}$ for the component $\lambda_{v,d}$ is defined as

$$\hat{b}_{v,d} := \frac{\mathrm{covar}(F_{v,d}^{1:L}, G_{v,d}^{1:L})}{\mathrm{var}(G_{v,d}^{1:L})}, \tag{4.42}$$

$$F_{v,d}^l := \nabla_{\lambda_{v,d}} \log q(X_v^l; \lambda_v) \log W^l, \tag{4.43}$$

$$G_{v,d}^l := \nabla_{\lambda_{v,d}} \log q(X_v^{1:L}; \lambda_v). \tag{4.44}$$

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 $G_v^l = \nabla_{\lambda_v} \log q(X_v^l | \lambda_v)$.

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 $q(v)$ 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 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)
```

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.

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

Algorithm 15 Black Box Variational Inference

```
1: global ρ, e
2: function EVAL(e, σ, ℓ)
3:     ...                                              ▷ As in Algorithm 14
4: function OPTIMIZER-STEP(q, ĝ)
5:     for v in dom(ĝ) do
6:         λ(v) ← GET-PARAMETERS(q(v))
7:         λ′(v) ← λ(v) + ...                           ▷ SGD/Adagrad/Adam update
8:         q′(v) ← SET-PARAMETERS(q(v), λ′)
       return q′
9: function ELBO-GRADIENTS(G^{1:L}, log W^{1:L})
10:    for v in dom(G^1) ∪ ... ∪ dom(G^L) do
11:        for l in 1, ..., L do
12:            if v ∈ dom(G^l) then
13:                F^l(v) ← G^l(v) log W^{1:L}
14:            else
15:                F^l(v), G^l(v) ← 0, 0
16:        b̂ ← SUM(COVAR(F^{1:L}(v), G^{1:L}(v)))/SUM(VAR(G^{1:L}(v)))
17:        ĝ(v) ← SUM(F^{1:L}(v) − b̂ G^{1:L}(v))/L
18:    return ĝ
19: function BBVI(S, L)
20:    σ ← [log W ↦ 0, q ↦ [ ], G ↦ [ ]]
21:    for t in 1, ..., T do
22:        for l in 1, ..., L do
23:            r^{t,l}, σ^{t,l} ← EVAL(e, σ, [ ])
24:            G^{t,l}, log W^{t,l} ← σ^{t,l}(G), σ^{t,l}(log W)
25:        ĝ ← ELBO-GRADIENTS(G^{s,1:L}, log W^{s,1:L})
26:        σ(q) ← OPTIMIZER-STEP(σ(q), ĝ)
27:    return ((r^{1,1}, log W^{1,1}), ..., (r^{1,L}, log W^{1,L}), ..., (r^{T,L}, log W^{T,L}))
```

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

$$\begin{aligned} \text{MB}(v) = \text{PA}(v) &\cup \{w : w \in \text{PA}(v)\} \\ &\cup \Big\{w : \exists u \Big(v \in \text{PA}(u) \wedge w \in \text{PA}(u)\Big)\Big\}. \end{aligned} \tag{4.46}$$

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 rely upon in the implementation of Gibbs sampling **??**.

5

A Probabilistic Programming Language With Recursion

In the three preceding chapters we have introduced a first-order probabilistic programming language and thoroughly described graph- and evaluation-based inference evaluators. The defining characteristic of the FOPPL is that it was suitably restricted to ensure that there could only ever be a finite number of random variables in any model denoted by any program.

In this chapter we loosen that restriction by introducing a new probabilistic programming language that supports advanced programming language features, such as higher-order procedures and general recursion. There are multiple reasons for loosening these restrictions and multiple ways for introducing the fundamental problem that arises from the increased expressivity of the language. As will become clear, the fundamental difficulty we will encounter is that HOPPL programs can denote models with an unbounded number of random variables. This rules out graph-based evaluation strategies immediately as an infinite graph cannot be represented on any 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.

Consider, to start, some of the restrictions that were put in place in the FOPPL:

- `loop`'s first argument, the loop depth, had to be a constant and `loop` was syntactic sugar, unrolled to nested let expressions at compile time;
- `defn` forms disallowed recursion;
- functions were not first class objects.

Say that we wish to unburden ourselves of the first restriction and we 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.

Consider a simple way to implement such a loop function, with an interface like that defined in the FOPPL

```
(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 as a function we have to allow the `defn` form to make recursive calls, a large departure from the FOPPL restriction. What doing this gives us though is 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, let alone potentially-complex generative modeling or simulation programs. But, as soon as loop is a function that takes a runtime value as a bound there is nothing to prevent writing a program like

```
(defn flip-and-sum [i v]
```

```
  (+ v (sample (bernoulli 0.5))))
(let [c (sample (poisson 1))]
  (loop c 0 flip-and-sum)).
```

There are, in fact, many practical reasons to write programs like this, as we shall soon demonstrate. This particular example is, however, a probably useless distribution over the cumulative sums of the outcomes of fair coin flips up to a Poisson distributed loop bound. Useless or not it is one of the shortest programs that can illustrate concretely what we mean by a program that denotes an infinite number of random variables. [**brooks:** *Let us try to use something that we don't need to call "useless"*] 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 $+\infty$ is in the support of the Poisson distribution one would need to desugar one loop with an infinite bound leading, of course, to an infinite number of random variables, the Bernoulli draws, in that expanded expression. The corresponding graphical model would have an infinite number of nodes and our FOPPL evaluators no longer work.

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 following 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. Such a primitive would definitely be provided by a probabilistic programming language, but that is not the point; the point is to demonstrate that the use of infinitely many random variables arises with the introduction of stochastic recursion, notable here in that it could be that this particular computation never terminates. Leveraging referential transparency the helper function above theoretically could be

inlined

```
(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. With no 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 can be applied in general.

While expanding the class of denotable models is important, the primary reason for us to introduce the complications of a higher-order language modeling 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 a 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! While you may not agree with the particular syntax of the following higher order language and you might instinctively shy from higher-order language features, be warned that if you wish to implement your own probabilistic programming language using the syntax and compiler infrastructure for an existing language rather than designing your own language carefully restricted as our FOPPL was, you will

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)

face the same issues as those introduced by the higher order features we cover and explain in this and the following chapters.

5.1 Syntax

A higher-order probabilistic programming language has first class functions and allows general recursion. A syntax for such a language appears in Language 5.4.

While a procedure had to be declared globally in the FOPPL, functions in the HOPPL can be defined anywhere locally as 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 use some 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

```
(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 is then 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, any valid FOPPL program is also a valid in the HOPPL.

5.3 Examples

In the HOPPL, we frequently follow standard design patterns for functional programming, allowing us to write more conventional code than was necessary to work around limitations of the FOPPL. We will ultimately be interested in probabilistic variants of existing higher-order languages, where we would hope for idiomatic model code. 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 coll]
  (if (empty? coll)
    (list)
    (cons (f (first coll))
```

```
    (map f (rest coll)))))
```

Program 5.5: Definition of `map`

This "loop" works by applying `f` to the first element of the collection `coll`, and then recursively calling `map` with the same function on the rest of the sequence. At the base case, for an empty input `coll`, we return a new empty list. More advanced implementations of `map` could apply multiple-argument functions across multiple sequences, or preserve type of the input sequence (e.g. list vs. vector arguments).

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 coll]
  (if (empty? coll)
    x
    (reduce f (f x (first coll)) (rest coll))))
```

Program 5.6: Definition of `reduce`

This function seems somewhat arcane at first glance, but is actually quite useful. For example, writing a function which computes the sum of all entries in a list can then be written in a single line:

```
(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 [original]
  (reduce
    (fn [reversed next-item] (cons next-item reversed))
```

```
    (list) ;; initialize reversed list
    original))
```

Design patterns such as these can significantly simplify and improve readability of model code relative to the FOPPL.

No need to inline data. A side effect of allowing unbounded numbers of random variables in the model is that we no longer need to "inline" our data. In the FOPPL, since each `loop` needed to have an explicit integer literal representing the total number of iterations in order to desugar to `let` blocks [**brooks:** *query: "blocks" or "expressions" for `let`?*], each program 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 needed to include an explicit function `datum` which provided an accessor to a particular data point n, and the model itself required hard-coding the number of loop iterations (there, 16). 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 state sequence
  (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 mixture

The ability to write loops of unknown or random iterations is not just a handy tool for writing more readable code; it also increases the expressivity of the model class. Consider the Gaussian mixture model example in Program 2.4: here there are 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 another useful higher-order helper function, `repeatedly`, which takes a number n and a function f, and constructs a list of length n where each entry is produced by invoking f.

```
(defn repeatedly [n f]
  (if (<= n 0)
    (list)
    (cons (f) (repeatedly (- n 1) f))))
```

The `repeatedly` function can stand in for the fixed-length loops that we used to sample mixture components from the prior in the original FOPPL implementation. An example implementation is in Program 5.7.

```
(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 5.7: 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, is sometimes known as an *open-universe* model: 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 stan-

dard 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. This is used frequently in the physical sciences, ... [**BP:** *give some examples from the sciences: stochastic simulation + constraint*]

Captcha-breaking

[**brooks:** *on hold until we decide what goes in the intro. need to describe captcha-breaking problem and show some pictures. things to mention: (1) unknown number of characters (2) we can also write a version which uses the fact that we have "real" captcha generators we can call (externally) as the generative model*]

```
(defquery captcha
 [image num-chars tol]
 (let [[w h] (size image)
       ;; sample random characters
       num-chars (sample
                  (poisson num-chars))
       chars (repeatedly
              num-chars sample-char)]
  ;; compare rendering to true image
  (map (fn [y z]
         (observe (normal z tol) y))
```

```
      (reduce-dim image)
      (reduce-dim (render chars w h)))
;; predict captcha text
{:text
 (map :symbol (sort-by :x chars))}))
```

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 a grammar:

Listing 5.8: Simple arithmetic function 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 $+$, $-$, $\times$, 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:

```
(def operations ['+ '- '* '/])

(defn gen-operation []
  (get operations
    (sample
      (uniform-discrete 0 (count operations)))))
```

```
(defn gen-arithmetic-expression []
  (let [expression-type (sample
                         (discrete [0.4 0.3 0.3]))]
    (cond
      (= expression-type 0)
        (sample (uniform-discrete 0 10))
      (= expression-type 1)
        'x
      :else
        (let [operation (gen-operation)]
          (list operation
                (gen-arithmetic-expression)
                (gen-arithmetic-expression))))))

(defn gen-function []
  (list 'fn ['x]
    (list (gen-operation)
          (gen-arithmetic-expression)
          (gen-arithmetic-expression))))
```

Program 5.9: generative model for function of a single variable

Note that `gen-function` in Program 5.9 returns a list, not a "function". That is, it does not directly produce a HOPPL function which we can call, but rather the uncompiled source code of a function. In defining it we have repeatedly used the single quote in front of symbol names: e.g. `'+` and `'fn`. This is a "quote" operation, distinct from string quoting (i.e., `"+"` or `"fn"`), with a return type *symbol*. We see here 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. Repeatedly 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 5.10: 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 5.9 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 here is `eval`, which we assume to be supplied as a primitive in the HOPPL target language.

```
;; A quoted expression
'(fn [x] (- x 8))

;; A function
(eval '(fn [x] (- x 8)))

;; Calling the function at x=10 (outputs: 2)
((eval '(fn [x] (- x 8))) 10)
```

Program 5.11: the magic word is `eval`

The usage of `eval` is demonstrated in Program 5.11. Running a single-site Metropolis-Hastings sampler, using an algorithm similar to that in Section 4.2 (which we will describe precisely in Section 6.5), we can draw posterior samples given particular data. Some example functions are shown in Figure 5.1, conditioning on three input-output pairs.

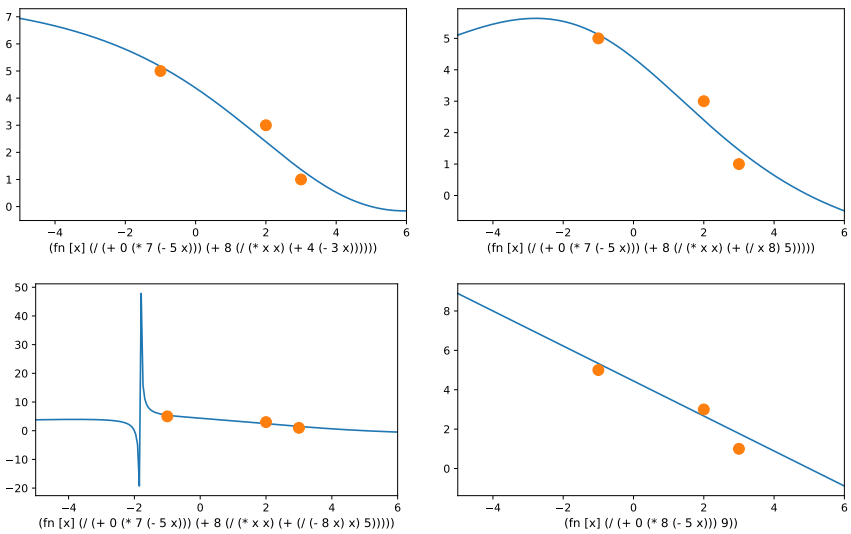

Figure 5.1: Examples of posterior sampled functions, drawn from the same MH chain.

6

Evaluation-based Inference II

Programs in the HOPPL may use unboundedly many random variables, and represent graphical models with infinitely many vertices. In order to perform probabilistic inference on HOPPL programs, therefore, we need an inference algorithm that works on only a finite subset of random variables of a HOPPL program at a time. For instance, standard message passing algorithms on graphical models, such as expectation propagation, cannot be used for the HOPPL language unless they are modified to work for infinite graphs.

In this chapter, we describe a class of inference algorithms for the HOPPL, which are able to address this challenge of unboundedly many random variables of HOPPL programs. One simple yet important insight behind these algorithms 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.

All the inference algorithms in this chapter use program evaluation as one of their core subroutines; up to a point that they all can be understood as non-standard schedulers of HOPPL programs. Furthermore, for any inference algorithms more complicated than simple likelihood

weighting, we use a flexible mechanism for running HOPPL programs, stopping them at some points, and resuming the stopped programs. In the inference algorithms in Chapter 4, we resorted to re-executing the entire program whenever modifying even a single random value. For the single-site Metropolis-Hastings algorithm, this was clearly inefficient relative to the graph-based algorithm which only needed to evaluate the probability expressions whose free variables included that which was modified. In the sequential Monte Carlo algorithm, the situation was even worse: at each `observe` checkpoint, needing to re-execute the entire program for each particle leads to an algorithm whose overall runtime is quadratic in the number of observations, rather than linear.

Explicit separation of model and inference code. The algorithms for inference in Chapters 3 and 4 were tightly tailored to our specific first-order PPL, requiring us to implement a specific graph compilation approach, or writing a custom evaluator for the entire language for each new inference algorithm.

A primary advantage of using a higher-order probabilistic programming language is that we can actually leverage existing compilers for real-world languages, rather than needing to write custom evaluators and custom languages. In the interface we consider here, the only means by which the inference code interacts with the model code is through implementations of the functions `sample` and `observe`, and through capturing the ultimate return value of each program execution.

This communication channel can actually be made very thin. In fact, it is not even necessary that the inference algorithm and the program itself execute on the same physical machine, an idea we will make explicit in Section 6.3. We suppose that a "controller" process which implements a particular inference algorithm is running independently from any copies of the program. To perform inference in the model, the controller only needs to be able to interact with the execution of the program at a few key points: it needs to be able to start new copies of the program, it needs to receive the final return value of the program, and (less trivially) it needs to be able to manipulate program execution at each `sample` and `observe` statement.

During any individual execution of the HOPPL program, when a `sample` or `observe` statement is encountered, normal execution pauses, sends a message back to the controller, and waits for a response. These messages will typically include an identifier (address) for the particular random choice, and a representation of the fully-evaluated arguments to `sample` and `observe`. The controller then performs whatever operations locally are necessary for inference, and sends back a value to the running program, which it interprets as the return value from the specific `sample` or `observe` call and resumes normal execution. This can be thought of as implementing a particular inference algorithm as an outer loop which launches program instances and captures their return values, in conjunction with specific implementations of `sample` and `observe` "checkpoint" operations.

As an example, we can informally outline how a controller that performs likelihood weighting inference could be implemented in this setting. (A precise algorithm will be presented in Section **??**.) To perform likelihood weighting, we need to repeatedly draw samples from the prior and score them against the observe statements. This can be thought of as, repeating forever:

- The controller starts a new execution of the HOPPL program, and sets a log weight $\log W = 0.0$;
- The controller repeatedly receives messages back the running program, and dispatches based on type:
  - If we are at a (`sample` d) statement, sample a value x from the distribution d and send x back to the running program;
  - If we are at a (`observe` d c), compute the log probability of c under distribution d, add it to $\log W$, and send c back to the running program;
  - If the program has terminated and is sending a return value c, emit as a weighted sample $(c, \log W)$ and exit this loop.

Running this algorithm relies on being able to pause and resume the execution of the program on the host flexibly. Furthermore, to implement sequential Monte Carlo, we will even need to be able to launch

multiple copies of the program which resume from the same point, after a resampling step.

How can we introduce a mechanism for stopping and resuming HOPPL programs? One simple approach is to write a custom interpreter for the language, 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 "stop" a running program, we simply store the memory state. The program can be "resumed" by making a (deep) copy of the saved memory back into the interpreter, and resuming execution. The trouble here is that although the asymptotic performance may be better — since this depends now only 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. One of the primary goals of moving to a higher-order (that is, "real world") language is to be able to leverage existing compilers to generate fast, efficient code.

A better option is available if we restrict the language to prohibit mutable state, in which we do not permit in-place operations on existing variables. 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. This separates the stateful inference code from the functionally-pure HOPPL code.

In languages with no mutable state, a standard way to support flexibly stopping, resuming, and re-executing programs is to transform HOPPL programs to those in the so-called continuation-passing style (CPS). This procedure 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.

Although we will present this explicitly for the HOPPL, the inference algorithms as presented can be implemented in any language whatsoever (they only need to be able to handle messages passed from the running program), and can be implemented independent of the language used to define the probabilistic program (so long as operations analogous to `sample` and `observe` send appropriate messages, and can pause, resume, and re-execute as required). Thus, this approach can be used to define probabilistic variants of any existing programming language.

In this chapter we will first describe the source code transformations we perform on the HOPPL to introduce addresses and to compile into 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.

6.1 Addressing Transformation

An addressing transformation modifies the original source code of the program to a new program, which performs the same computation, while simultaneously keeps track of an *address*: a representation of the current execution point of the program. The address ideally would uniquely identify any particular point in the computation. Note that the straightforward procedure we used for addresses in Chapter 4 is not possible here, as it required inlining the bodies of all function applications to create an exhaustive list of `sample` and `observe` statements, which is not possible in the HOPPL.

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 procedural languages, an even better version of address would also track values of important local variables at each nesting level; for example, the value of loop iteration counters.) In functional programming languages like the HOPPL, however, a stack trace effectively provides a unique identifier for the current location in the program execution. When it comes time to implement inference algorithms, this will be invaluable: we can store the sequence of random choices made during execution of a program in a database, which uses these addresses as its keys.

We will denote the current address by the symbol α. The addressing transformation we present here for the HOPPL follows the procedure given in Wingate et al. [2011]; all function calls, `sample` statements, and `observe` statements are modified to take an additional argument which provides the current address.

As with the evaluators and graph compilers in previous chapters, we will describe the addressing transform in terms of an evaluator $\Downarrow_\alpha$, which translates expressions within the same language, adding addresses along the way. We additionally define a secondary evaluator $\downarrow_\alpha$, which instead of operating on expressions e, operates on the top-level HOPPL program q. This secondary evaluator defines the top-level outer address; that is, the base of the stack trace.

Variables, procedure names, constants, and if. Since the addresses track the current call stack, variables and function names are unaffected.

$$\frac{}{v \Downarrow_\alpha v} \qquad \frac{}{f \Downarrow_\alpha f}$$

Constants may refer to either literal values (e.g. the boolean value `true`, or a floating point number), in which case they are unaltered, or to primitive functions, in which case the additional argument α which carries the current address is discarded.

$$\frac{c \text{ is a constant value}}{c \Downarrow_\alpha c} \qquad \frac{c \text{ is a primitive function with } n \text{ arguments}}{c \Downarrow_\alpha \texttt{(fn [}\alpha\ v_1\ \ldots\ v_n\texttt{] (}c\ v_1\ \ldots\ v_n\texttt{))}}$$

We are unable to "step in" to primitive functions.

Control flow on `if` statements also does not modify the address in our implementation.

$$\frac{e_1 \Downarrow_\alpha e_1' \quad e_2 \Downarrow_\alpha e_2' \quad e_3 \Downarrow_\alpha e_3'}{\texttt{(if } e_1\ e_2\ e_3\texttt{)} \Downarrow_\alpha \texttt{(if } e_1'\ e_2'\ e_3'\texttt{)}}$$

[**BP:** *probably worth describing this rule further, or talking about relative merits*]

Functions, sample, and observe. Functions are updated to take an extra address argument.

$$\frac{e \Downarrow_\alpha e'}{\texttt{(fn [}v_1\ \ldots\ v_n\texttt{]}\ e\texttt{)} \Downarrow_\alpha \texttt{(fn [}\alpha\ v_1\ \ldots\ v_n\texttt{]}\ e'\texttt{)}}$$

So far, we have done nothing; we have simply threaded an address through the entire program, but this address does not change. We update the address when calling a function:

$$\frac{e_i \Downarrow_\alpha e_i' \text{ for } i = 0, \ldots, n \quad \text{fresh } v}{\texttt{(}e_0\ e_1\ \ldots\ e_n\texttt{)} \Downarrow_\alpha \texttt{(}e_0'\ \texttt{(new-addr }\alpha\ v\texttt{)}\ e_1'\ \ldots\ e_n'\texttt{)}}$$

The outcome of this rule is that when the body of the function e_0' gets executed, the address will be updated to reflect that we are now nested one level deeper in the call stack.

Here, we have introduced the function `new-addr` which creates a new address. This new address is some composition of the current address α, and some new identifier v for the current function call. If we take the stack trace metaphor literally, then if α is a list-like data structure, `new-addr` can concatenate v onto the end of the list. Alternatively, `new-addr` could perform some sort of hash on α, v to yield an address of constant size regardless of recursion depth. Note, also, that v can be chosen flexibly — human-readable choices for v are likely based somehow on the expression e_0.

The expressions for `sample` and `observe` can be thought of as special cases of general function application.

$$\frac{e \Downarrow_\alpha e' \quad \text{fresh } v}{\texttt{(sample } e\texttt{)} \Downarrow_\alpha \texttt{(sample (new-addr }\alpha\ v\texttt{)}\ e'\texttt{)}}$$

$$\frac{e_1 \Downarrow_\alpha e_1' \quad e_2 \Downarrow_\alpha e_2' \quad \text{fresh } v}{\texttt{(observe } e_1 \ e_2\texttt{)} \Downarrow_\alpha \texttt{(observe (new-addr } \alpha \ v\texttt{)} \ e_1' \ e_2'\texttt{)}}$$

The entire process of creating addresses is introduced primarily to ensure that `sample` and `observe` receive addresses as arguments which allow inference algorithms to compare the values of random variables encountered through different executions of the program.

Top-level addresses and program translation. While the $\Downarrow_\alpha$ operator above can translate any expressions, we still have not addressed how to convert a HOPPL program as a whole to a form which keeps track of addresses. To do so, we need to introduce the program addressing relation $\downarrow_\alpha$. This relation wraps the overall program execution, binding an initial fixed address α_0 as the starting address (e.g. an empty stack trace) for the execution of the program.

$$\frac{e \Downarrow_\alpha e'}{e \downarrow_\alpha \texttt{(fn [}\alpha\texttt{]} \ e'\texttt{)}}$$

For programs which include functions that are user-defined at the top level, this relation also inserts the additional address argument into the function definitions.

$$\frac{e \Downarrow_\alpha e' \qquad q \downarrow_\alpha q'}{\texttt{(defn } f \ \texttt{[}v_1 \ \ldots \ v_n\texttt{]} \ e\texttt{)} \ q \downarrow_\alpha \texttt{(defn } f \ \texttt{[}\alpha \ v_1 \ \ldots \ v_n\texttt{]} \ e'\texttt{)} \ q'}$$

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.

6.2 Continuation-Passing-Style Transformation

Now that each function call in the program has been augmented with an address which tracks our location in the program execution, the next step is to linearize the program in a manner which allows us to flexibly pause and resume execution. The continuation-passing-style (CPS) transformation does this linearization. Here we use a simplest version of the transformation; for better optimized CPS transformations, see

Appel [2006]. Our description is given in terms of the following $\Downarrow_c$ relation:

$$e,\ \kappa,\ \sigma \Downarrow_c e'.$$

Here e is an HOPPL expression, and κ is a function, called continuation, which consumes the result of e and produces an answer. The last e' is the result of CPS-transforming e under the continuation κ. In contrast to a typical CPS transformation, here we also include a state σ which we thread through the execution of the program. This is not generally part of converting a program to continuation passing style, but here we will use it to store any information about the execution which is necessary to perform inference; this will be described in detail later when we discuss implementation of `sample` and `observe`. We define the $\Downarrow_c$ relation by considering each case of e separately and using the inference-rules notation. Then, as in our addressing transformation, we use this relation to define the CPS transformation of program q, which is specified by another relation

$$q,\ \sigma \downarrow_c q'.$$

Variables, Procedure Names and Constants

$$\frac{}{v,\ \kappa,\ \sigma \Downarrow_c (\kappa\ \sigma\ v)} \qquad \frac{}{f,\ \kappa,\ \sigma \Downarrow_c (\kappa\ \sigma\ f)} \qquad \frac{\text{CPS}(c) = c'}{c,\ \kappa,\ \sigma \Downarrow_c (\kappa\ \sigma\ c')}$$

These rules say that when e is a variable, a procedure name or a constant, its CPS transform simply calls the continuation on e itself or its slight adjustment. The adjustment is needed for the constant case $e \equiv c$, and is done by the subroutine CPS. When c is a constant value of a ground type, such as boolean values, integers and real numbers, the subroutine returns its input c without any adjustment. But when c is a primitive operator, such as $+$, the subroutine returns the CPS variant of this operator. For instance,

$$\text{CPS}(+) = (\texttt{fn}\ [v_1\ \ v_2\ \ \kappa_0\ \ \sigma]\ \ (\kappa_0\ \ \sigma\ \ (+\ \ v_1\ \ v_2))).$$

For all the usual operators c, such as $+$ and $\times$, we represent their CPS variants with $\widetilde{c}$, such as $\widetilde{+}$ and $\widetilde{\times}$. The CPS-transformed variants add two additional arguments: a continuation κ, and a state σ. This

state σ will be used to hold any side-effect computations needed to implement inference algorithms, and the state is threaded through as the first argument of the call to the next continuation. This means our continuations are actually functions of two arguments.

[**HY:** *I try to use "procedure" instead of "function" throughout. But this might not be an optimal choice. Typically, a procedure means a function that does not return a value or a value of the unit type, and that is invoked mainly for its side effect. This is not the case here. Anyhow, although suboptimal, I will stick to my choice for now.*]

If

$$\frac{\begin{array}{ll} e_2,\ \kappa,\ \sigma \Downarrow_c e_2' & e_3,\ \kappa,\ \sigma \Downarrow_c e_3' \\ \text{Choose a fresh variable } v & e_1, (\texttt{fn}\ [\sigma\ v]\ (\texttt{if}\ v\ e_2'\ e_3')),\ \sigma \Downarrow_c e' \end{array}}{(\texttt{if}\ e_1\ e_2\ e_3),\ \kappa,\ \sigma \Downarrow_c e'}$$

We first transform the true and false branches, e_2 and e_3, under the current continuation κ, and then the condition e_1 under a newly constructed continuation that calls the CPS-transformed e_2 or e_3 depending on the result of e_1.

Procedure Definition

$$\frac{\text{Choose a fresh variable } \kappa' \qquad e,\ \kappa',\ \sigma \Downarrow_c e'}{(\texttt{fn}\ [v_1\ \ldots\ v_n]\ e),\ \kappa,\ \sigma \Downarrow_c (\kappa\ \sigma\ (\texttt{fn}\ [v_1\ \ldots\ v_n\ \kappa'\ \sigma]\ e'))}$$

We transform an (anonymous) procedure $(\texttt{fn}\ [v_1\ \ldots\ v_n]\ e)$ in three steps. First, we introduce a new continuation parameter κ' to the procedure that is intended to consume the return value of the procedure. Next, we transform the procedure body e recursively under this newly introduced κ'. Finally, we use the result of this transformation for constructing a new procedure, and pass it to the current continuation κ, which expects a procedure, not the result of a procedure, as its input. For instance,

$$(\texttt{fn}\ []\ 1),\ \kappa,\ \sigma \Downarrow_c (\kappa\ \sigma\ (\texttt{fn}\ [\kappa'\ \sigma]\ (\kappa'\ \sigma\ 1)))$$

Notice that 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

$$\frac{\begin{array}{l}\text{Choose fresh variables } v_0, \ldots, v_n \\ e_n,\ (\texttt{fn}\ [\sigma\ v_n]\ (v_0\ v_1 \ldots v_n\ \kappa\ \sigma)),\ \sigma \Downarrow_c e'_n \\ e_i,\ (\texttt{fn}\ [\sigma\ v_i]\ e'_{i+1}),\ \sigma \Downarrow_c e'_i \quad \text{for } i = (n-1), \ldots, 0\end{array}}{(e_0\ e_1 \ldots e_n),\ \kappa,\ \sigma \Downarrow_c e'_0}$$

We transform all of the arguments of the call from the last parameter e_n to the first e_1, and then the operator e_0, while using the CPS-transformed e_i to build a continuation for e_{i-1}. The CPS transformation e'_0 of e_0 becomes that of the call $(e_0\ e_1 \ldots e_n)$. This CPS transformation makes the order of evaluation explicit: when e'_0 gets executed, it first evaluates the (CPS-transformed) e_0 to find out an operator v_0, then the arguments $e_1 \ldots e_n$ in that order in order to get parameter values $v_1 \ldots v_n$, and finally the application of the operator to these values. For instance, instantiating the rule for $(+\ e_1\ e_2)$ and κ and simplifying it slightly gives:

$$\frac{\begin{array}{l}e_2,\ (\texttt{fn}\ [\sigma\ v_2]\ (\widetilde{+}\ v_1\ v_2\ \kappa\ \sigma)),\ \sigma \Downarrow_c e'_2 \\ e_1,\ (\texttt{fn}\ [\sigma\ v_1]\ e'_2),\ \sigma \Downarrow_c e'_1\end{array}}{(+\ e_1\ e_2),\ \kappa,\ \sigma \Downarrow_c e'_1}$$

Observe and Sample

$$\frac{\begin{array}{l}\text{Choose fresh variables } v_1, v_2 \\ e_2,\ (\texttt{fn}\ [\sigma\ v_2]\ (\texttt{observe}^\star\ \alpha\ v_1\ v_2\ \kappa\ \sigma)),\ \sigma \Downarrow_c e'_2 \\ e_1,\ (\texttt{fn}\ [\sigma\ v_1]\ e'_2),\ \sigma \Downarrow e'_1\end{array}}{(\texttt{observe}\ \alpha\ e_1\ e_2),\ \kappa,\ \sigma \Downarrow_c e'_1}$$

$$\frac{\begin{array}{l}\text{Choose a fresh variable } v \\ e,\ (\texttt{fn}\ [\sigma\ v]\ (\texttt{sample}^\star\ \alpha\ v\ \kappa\ \sigma)),\ \sigma \Downarrow_c e'\end{array}}{(\texttt{sample}\ \alpha\ e),\ \kappa,\ \sigma \Downarrow_c e'}$$

These two rules are unique for the CPS transform of probabilistic programming languages. They replace `observe` and `sample` operators by $\texttt{observe}^\star$ and $\texttt{sample}^\star$, which take two additional parameters κ for the current continuation and σ for the current state. We also want to remind the reader that the first parameter α is an address and serves as a

marker for a particular subexpession in a given probabilistic program. Implementing `observe`* and `sample`* corresponds to writing an inference algorithm for probabilistic programs. Typically, 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 label α, parameters, current continuation κ and current state σ. The algorithm updates the state σ based on this information, and resumes the execution with this continuation and the modified state, or switches to the execution of another suspended expression.

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':

$$\frac{\text{Choose fresh variables } v, \sigma, \sigma' \qquad e, \texttt{(fn [}v\ \sigma\texttt{] (list }v\ \sigma\texttt{))}, \sigma' \Downarrow_c e'}{\texttt{(fn [}\alpha\texttt{] }e\texttt{)} \downarrow_c \texttt{(fn [}\alpha\ \sigma\texttt{] }e'\texttt{)}}$$

$$\frac{\text{Choose fresh variables } \kappa, \sigma \qquad e, \kappa, \sigma \Downarrow_c e' \qquad q \downarrow_c q'}{\texttt{(defn }f\ \texttt{[}v_1\ \ldots\ v_n\texttt{] }e\texttt{)}\ q \downarrow_c \texttt{(defn }f\ \texttt{[}v_1\ \ldots\ v_n\ \kappa\ \sigma\texttt{] }e'\texttt{)}\ q'}$$

One major 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 as a list, behaving almost like the identity function.

6.3 An Interface Between Model and Inference

Once we have inserted addresses into our programs, and transformed them into CPS, we are ready to perform inference. But how do we do this? The body of a CPS-transformed program, instead of being some expression e, is a new expression e' which has as free variables an initial state σ_0 and an outermost continuation κ_0.

For purposes of presenting algorithms, we take the separation of model and inference to its extreme, and show how inference backends can be implemented over a client-server network infrastructure. We assume the existence of an architecture that supports centrally-coordinated asynchronous message passing in the form of request and

response. Common networking packages provide abstractions of such patterns, e.g. ZeroMQ[1]. We will also implicitly define message types by their use, and assume mechanisms for defining and serializing to and from messages, e.g. protobuf[2]. We will assume the existence of two methods, SEND and RECEIVE which pack and unpack their arguments automatically.

The general architecture of a client-server probabilistic programming inference engine is that there is a "server" which, somewhat confusingly, is a stateful evaluator, potentially multi-threaded, that evaluates the HOPPL in response to requests from the "client" which makes requests of the server about particular ways it would like the server to run the program. It is perhaps less confusing to refer to the "client" as a controller because that, in effect, is what it is doing to the probabilistic program.

Communication from the running program to the controller. A running program needs to send a message to the controller when it reaches a `sample` or `observe` statement, and its return value when it terminates. The controller processes the message, and responds with instructions on how to continue execution. We define these program-to-controller messages to take the form

$$(\text{type}, \alpha, d, c, \sigma)$$

where "type" is a string identifying the message type, α is an "address" (an identifier for the current location in the program execution), d is a distribution, c is a value, and σ is a map which holds any additional state required by the inference algorithm. For some message types, d or c may take the null value `nil`. These messages must only contain values which are easily serializable; that is, values which we can instantiate and process in both the HOPPL and the language which implements the inference algorithm itself. In particular this requires a bit of care for the distribution d; it is necessary to send it in some language-agnostic manner (e.g., as a data structure containing the distribution name, and the values of its parameters).

[1] `http://blog.scottlogic.com/2015/03/20/ZeroMQ-Quick-Intro.html`
[2] `https://developers.google.com/protocol-buffers/`

There are three types of messages which can be sent from the HOPPL evaluation process to the inference controller process:

- `"sample"` messages are sent from (`sample` d) expressions, and include the distribution d;
- `"observe"` messages are sent from (`observe` d c) expressions, and include the distribution d and value c; and
- `"return"` messages are sent when the program terminates, and include the return value as c.

The controller responds to each of these with a message that tells the running HOPPL program how to continue execution. For example, at `sample` statements it is necessary to provide a sampled random value from the distribution d; how this value is generated will depend on the particular inference algorithm. Similarly, at `observe` statements, when running algorithms such as SMC we will want to instruct particular executions of the program to fork and run multiple copies, with others terminating prematurely.

Communication from the controller to the program. In order to define inference algorithms we also need to specify the format of messages sent from the controller to the HOPPL programs, directing execution. To handle the fact that communication is not just between the controller and a single running program, but possibly multiple programs, and that the controller needs to be able to launch new copies of the program, we imagine a "server" which handles and routes incoming messages from the controller; these messages are then either dispatched to a single currently-running HOPPL program, or (in the case of launching a new execution of the program) handled by the server itself. We suppose there are three different message types which can be sent to the server:

- A "start" message (`"start"`, σ_0) which tells the server to begin execution of a new copy of the program, with the specified initial state σ_0. In order to disambiguate between multiple copies of the program running on the server simultaneously (which could occur

in SMC, or in e.g. asynchronous implementations of likelihood weighting), for all inference algorithms we choose to include in the state an identifier field "id", which we require to be unique among all currently-running programs.

- A "continue" message ("continue", id, σ, c) which tells the server to resume execution of a currently running program with the specified id, with a (potentially modified) state σ and a (potentially `nil`) value c to be passed into the current continuation.
- A "fork" message ("fork", id, `list-of-states`, `list-of-values`) which tells the server to continue the program, launching potentially several independent copies by calling the continuation multiple times, each time with a different state and a different value. The total number of copies launched is the length of `list-of-states` and `list-of-values`, which much be equal. When this length is 1, sending a "fork" message is effectively equivalent to sending a "continue" message; when the length is 0 (e.g. an empty list), then the continuation is not called at all, terminating this particular execution early and without generating any return value. Note that each state σ in `list-of-states` must have unique values as their entry σ(id).

6.3.1 Mechanics of message passing

Implementing the server which runs the transformed HOPPL programs requires defining the outermost continuation and implementing the `sample`* and `observe`* checkpoints. These three methods will be responsible for sending messages back to the controller which actually implements each particular inference algorithm. To do this we introduce two new functions `send` and `receive` which can be run on the server, and which we assume to be provided by an underlying message passing implementation. These are defined such that:

- (`send` type α d c σ) creates a new message (type, α, d, c, σ) which it passes to the client, and then returns `nil`.
- (`receive` id) blocks and waits for a new "continue" or "fork"

message to arrive from the client. It then returns a tuple (σ, x), where x contains a list of values and σ contains a list of states; if the message was a `"continue"` message, then the single value and state are wrapped in lists of length one.

We will use the `send` and `receive` signatures to define the CPS-transformed procedures corresponding to the language constructs for sampling, observing, and top-level returning of running HOPPL programs; these were assumed to be accessible and implemented as primitives in Section 6.2. In the following we introduce a bit of destructuring sugar, in which the output tuple from a `receive` call is assigned into two separate values directly. The function interfaces here for `sample***` and `observe***` are exactly those we had in the previous section, after first introducing addresses, and then applying the CPS transformation to the address-transformed code:

```
(defn sample*** [α d κ σ]
  (do
    (send "sample" α d nil σ)
    (let [[σ′ x] (receive (get σ :id))]
      (assert (= (count x) 1))
      (map (fn [σℓ xℓ] (κ σℓ xℓ)) σ′ x))))
```

Program 6.1: sample***

```
(defn observe*** [α d c κ σ]
  (do
    (send "observe" α d c σ)
    (let [[σ′ nil] (receive (get σ :id))]
      (map (fn [σℓ] (κ σℓ c)) σ′))))
```

Program 6.2: observe***

These implementations are very similar: both send out a message of their respective type, and then apply the passed continuation κ to the values received from the controller. The main differences are that `observe***` always passes as its value the original observed value c, and that we enforce `sample***` only accepts `"continue"` messages (rather than `"fork"`) messages.

The outer continuation, or the "return" continuation, will wrap the return value in a message which it then sends to the controller.

Algorithm 16 HOPPL inference engine program executor, a "server"

```
1: repeat
2:     [type inits] ← RECEIVE()
3:     switch type do
4:         case "start"
5:             START-THREAD((start*** inits))
               ▷ Other messages handled by currently running programs
6: until forever
```

```
(defn return*** [r σ]
    (send "return" α0 nil r σ))
```

Program 6.3: return***

This continuation is applied to the overall CPS-transformed program p by the following `start***` function

```
(defn start*** [init-state]
  (let [[r σ] (p α0 (merge {} init-state))]
    (return*** r σ)))
```

Program 6.4: start***

With these functions defined, we are now ready to launch the "server" which runs HOPPL code on-demand, as scheduled by the controller. The HOPPL server outer loop, shown in Algorithm 16, listens for start messages and launches new copies of the program in new threads appropriately.

6.4 Likelihood Weighting

Setting up the capability to run HOPPL programs on demand, in a flexible manner, required a fair amount of work. However, the payoff now is that we have an interface which we can use to easily write many different evaluation-based inference algorithms. We illustrate this benefit with a series of inference algorithms, starting with likelihood weighting. Algorithm 17 makes explicit the procedure we described high-level at the beginning of this chapter. The procedure EMIT outputs a weighted particle (e.g. to disk, or to standard output).

Algorithm 17 HOPPL inference engine likelihood weighting controller, a "client"

```
1:  ℓ ← 0
2:  repeat
3:      ℓ ← ℓ + 1
4:      SEND("start", [id ↦ ℓ])
5:      repeat
6:          [type α d c σ] ← RECEIVE()
7:          id ← σ[id]
8:          switch type do
9:              case "sample"
10:                 x ← SAMPLE(d)
11:                 SEND("continue", id, σ, x)
12:             case "observe"
13:                 σ(log W) ← σ(log W) + LOG-PROB(d, c)
14:                 SEND("continue", id, σ, c)
15:             case "return"
16:                 EMIT(c, σ(log W))
17:                 break
18:     until forever
19: until forever
```

In this implementation, we store in the state σ both the id (here, just an integer counter) and the log weight of the execution. One question you could ask is whether or not it is strictly necessary to use the state at all; we could simply track the log weight in a global variable in the controller (that is, client-side). In this case we would not even need ids. However, this would preclude a parallel implementation. There is nothing preventing us from running multiple executions of the HOPPL program simultaneously (e.g. in multiple threads); however, in that case it is necessary for the controller to track which log weight is associated with which particle. While we could store that information in a data structure in the controller process, it still would be necessary to pass back and forth at least the id. Note that while this controller imple-

mentation in Algorithm 17 runs in a single thread, a parallel version could be obtained by running the outermost **repeat** loop in a multi-threaded way and synchronizing access to the counter ℓ when sending `"start"`.

6.5 Metropolis-Hastings

We next implement a single-site Metropolis-Hastings algorithm using this interface. The full algorithm, given in Algorithm 18, has an overall structure which closely follows that of the evaluation-based algorithm for the first-order language given in Section 4.2.

The primary difference between this algorithm and that of Section 4.2 is due to the dynamic addressing. In the FOPPL, since each function is guaranteed to be called only a finite number of times, we can unroll the entire computation, inlining functions, and literally annotate every `sample` and `observe` which could be encountered during execution with a unique identifier. In the HOPPL, a program has an unbounded number of addresses that can be encountered, if a program is repeatedly executed an arbitrary number of times. We maintain on the controller-side a database (i.e. a key-value store, or hashmap) which tracks the current value of the random choice at each address, and its log probability under the distribution at that address.

In both the HOPPL and the evaluation-based approach to the FOPPL, although the total number of `sample` and `observe` statements in each execution can be random, we only ever compare two execution traces, each of fixed finite size. The acceptance ratio when running single-site MH on a HOPPL program is the same as that used previously and described in Algorithm 9.

Relative to the graph-based MH sampler from Section 3.3, a weakness here is that this algorithm still needs to re-run the entire program when proposing a change to a single random choice, whereas the graph-based algorithm was able to prune out expressions which did not include the modified random variable. Recent work has suggested ways to avoid this overhead. In a CPS-based implementation, we can keep around copies of the continuations at each address and instead of re-

Algorithm 18 HOPPL inference Metropolis-Hastings controller

```
1: ℓ ← 0                                                    ▷ Iteration counter
2: r, 𝒳, log 𝒫 ← nil, [], []                                ▷ Current trace
3: 𝒳′, log 𝒫′                                               ▷ Proposal trace
4: function ACCEPT(β, 𝒳′, 𝒳, log 𝒫′, log 𝒫)
5:     ...                                                  ▷ as in Algorithm 9
6: repeat
7:     ℓ ← ℓ + 1
8:     β ∼ UNIFORM(dom(𝒳))          ▷ Choose a single address to modify
9:     SEND("start", [id ↦ ℓ])
10:    repeat
11:        [type, α, d, c, σ] ← RECEIVE()
12:        switch type do
13:            case "sample"
14:                if α ∈ dom(𝒳) \ {β} then
15:                    𝒳′(α) ← 𝒳(α)
16:                else
17:                    𝒳′(α) ← SAMPLE(d)
18:                log 𝒫′(α) ← LOG-PROB(d, 𝒳′(α))
19:                SEND("continue", σ[id], σ, 𝒳′(α))
20:            case "observe"
21:                log 𝒫′(α) ← LOG-PROB(d, c)
22:                SEND("continue", σ[id], σ, c)
23:            case "return"
24:                if ℓ = 1 then
25:                    u ← 1                  ▷ Always accept first iteration
26:                else
27:                    u ∼ UNIFORM-CONTINUOUS(0, 1)
28:                if u < ACCEPT(β, 𝒳′, 𝒳, log 𝒫′, log 𝒫) then
29:                    r, 𝒳, log 𝒫 ← c, 𝒳′, log 𝒫′
30:                EMIT(r, 0.0)               ▷ MH samples are unweighted
31:                break
32:    until forever
33: until forever
```

suming from the beginning of the program we can resume by calling the continuation at the address we are modifying; this is done by the Anglican language [Tolpin et al., 2016]. While each proposal still requires re-running the entire program in the worst case, proposals which modify random variables later in the execution will be more efficient. This can be refined further using the C3 approach [Ritchie et al., 2016a], which additionally caches return values of continuations for particular input arguments; this can yield inference with asymptotics matching those of single-site MH in the graph-based algorithm for models such as the hidden Markov model.

6.6 Sequential Monte Carlo

While the previous two algorithms were very similar to those presented for the FOPPL, running SMC in this context is a bit more challenging. 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 algorithm is given in Algorithm 19. The function NEWID is assumed to generate a new unique value which does not collide with any current execution id. Note that this implementation of SMC resamples after every single `observe` statement; in order to behave properly, it requires that every execution of the program generates the same sequence of `"observe"` messages; that is, the same number of `observe` statements, and with the same sequence of addresses, must be produced regardless of the random choices which are sent in response to `sample` statements.

Much of the apparent complexity of this algorithm block is due to run-time checks to ensure these properties holds. We add three variables to check correctness: an observe counter m_o, a return counter m_r, and an address β. Each SMC iteration initializes by launching a fixed number L of particles. Unlike the other algorithms which only executed a single copy of the program at a time, the inner loop of the execution algorithm for SMC must handle all sending and receiving from multiple currently executing programs.

Conceptually, the algorithm works by breaking down execution into a series of partial executions where each of the L copies is run in parallel until reaching the next `observe`, where execution pauses and waits; once every copy of the program has reached the `observe`, then we perform a resampling step and resume execution. 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. Instead, we use the counter m_o to number the observe messages as they come in; whichever execution sends an `"observe"` message first becomes particle 1 for that stage of the algorithm. Unlike in the previous algorithms, we do not immediately respond with a send message, instead storing the particle weight and id. To ensure that each particle is at the "same" `observe` statement, we also record the address in the first `"observe"` message, and throw an error if any of the subsequent $L - 1$ particles to reach the `observe` have a different address. Once $m_o = L$, all particles have reached the next `observe` and can resample. Resampling here is handled by sampling a number of offspring o_ℓ for each of the $\ell = 1, \ldots, L$ particles. Each of the paused L executions is then sent a `"fork"` message. Within the running HOPPL program, this is then handled by `observe***`, which will call each continuation an appropriate number of times. We also enforce that each copy of the program terminates at the "same time", by which we mean each copy of the program contains the same total number of `observe` statements. This is handled by tracking the number of `"return"` messages in the m_r counter, in a similar manner to tracking observes.

An obvious weakness of this algorithm is that it cannot handle `observe` statements which appear conditionally; e.g., `observe` statements which may be pruned from the execution due to lazy evaluation of the program. This could be avoided by only resampling at a subset of `observe` statements which are guaranteed to appear in the same order in every execution of the program. This could be handled manually by augmenting the `observe` statement (and the `"observe"` message) to annotate which observes should be treated as triggering a resample, and which should simply accumulate log probability in the state (à la

the likelihood weighting implementation).

A more subtle problem with this implementation, if implemented literally, is that the `map` operation used in the implementation of `observe***` is likely to be a serial implementation. This means we have some surprising computational behavior, where the initial L particles launched by the `"start"` message are all running in separate threads, as the algorithm continues, after several resampling steps the particles will gradually coalesce to a single ancestor; at this point all particles will be running in a single server thread. This can be bypassed by replacing the function we `map` over in `observe***` with a function that wraps each call to the current continuation in a separate thread.

Algorithm 19 HOPPL inference engine SMC controller

$\log W_1, \ldots, \log W_L$ ▷ Log weights of L executions
$\text{id}_1, \ldots, \text{id}_L$ ▷ Stored ids of L executions
repeat
  $m_o, m_r \leftarrow 0$ ▷ Observe counter, return counter
  $\beta \leftarrow \text{nil}$ ▷ Address of current `observe`
  $\log \hat{Z} \leftarrow 0.0$
  for ℓ **in** $1, \ldots, L$ **do**
    $\log W_\ell \leftarrow 0.0$
    SEND("start", $[\text{id} \mapsto \ell]$) ▷ Start L copies of the program
  repeat
    $[\text{type}, \alpha, d, c, \sigma] \leftarrow$ RECEIVE()
    switch type **do**
      case "sample"
        $x \leftarrow$ SAMPLE(d)
        SEND("continue", $\sigma(\text{id}), \sigma, x$)
      case "observe"
        $m_o \leftarrow m_o + 1$
        $\log W_{m_o} \leftarrow$ LOG-PROB(d, c)
        $\text{id}_{m_o} \leftarrow \sigma(\text{id})$
        if $\beta = \text{nil}$ **then** $\beta \leftarrow \alpha$ ▷ new `observe`
        else if $\beta \neq \alpha$ **then error** ▷ ensure same address
      case "return"
        $m_r \leftarrow m_r + 1$
        EMIT($c, \log W_{m_r}$)
    if $m_o = L$ **then**
      $W_1, \ldots, W_L \leftarrow \exp(\log W_1), \ldots, \exp(\log W_L)$
      $o_1, \ldots, o_L \leftarrow$ RESAMPLE($W_{1:L}$)
      $\log \hat{Z} \leftarrow \log \hat{Z} + \log \frac{1}{L} \sum_{\ell=1}^{L} W_\ell$
      for ℓ **in** $1, \ldots, L$ **do**
        for $i = 1, \ldots, o_\ell$ **do**
          $\sigma'_i \leftarrow \sigma_\ell[\text{id} \mapsto \text{NEWID}()]$
        SEND("fork", $\text{id}_\ell, \sigma'_1, \ldots, \sigma'_{o_\ell}$, nil)
      $m_0, \beta \leftarrow 0$, nil
    if $m_r = L$ **then**
      break
  until forever
until forever

7

Advanced Topics

So far in this paper 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 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, then move on to looking at challenges to implementing Hamiltonian Monte Carlo and variational inference within the HOPPL, implementing expressive models by recursively nesting probabilistic programs, and 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 is 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 higher performance high-speed trading decision engines. 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., 2015b, Ritchie et al., 2016b], where program-specific neural networks were trained to guide forward inference.

Le et al. [2017b] 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

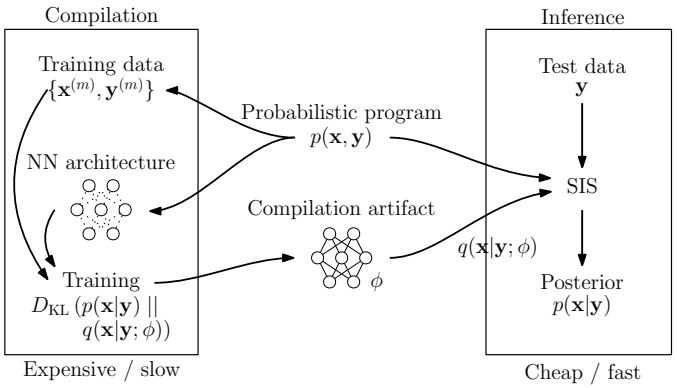

Figure 7.1: An outline of an approach to inference compilation for amortized inference for probabilistic programs. Re-used with permission from [Le et al., 2017b].

real data. Paige and Wood [2016] introduces a strategy for learning inverse programs for models with finite parameter dimension, i.e. models denotable in the FOPPL. Le et al. [2017b] 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. [2017b] 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

$$q_\alpha(x_\alpha|X_{n-1}, Y) \tag{7.1}$$

[**FW:** *should* x_α *be* $X_n(\alpha)$ *instead?*] noting that this is not the incremental prior and that it in general depends on all observations Y. 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

$$W^\ell = p(Y|X) \prod_{\mathrm{dom}(X)} \frac{p_\alpha(x_\alpha|X_{n-1})}{q_\alpha(x_\alpha|X_{n-1}, Y)}. \tag{7.3}$$

The problem then becomes one of "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.

[**HY:** *Previously, what comes in the paranthesis was: (simply de-sugar all observe statements to statements, e.g.* `(observe d c)` *becomes* `(fn [] (factor (observe* d c)) c)`*). I think that we can just change observe to sample. Frank's warning:* [**FW:** *this is not true anymore when observe returns* `c` *and I can't think of a quick way to fix it quickly right now whilst writing; perhaps importance sampling from the joint by transforming* `(observe d c)` *into* `(fn [] (factor (observe* d c)) c)` *and using importance sampling from the joint? – i've tried introducing the fix – see what you think, it's a mess now*]]

What remains is to choose a specific form for the proposal distribution to be learned. Consider a form like

$$q(X|Y;\phi) = \prod_{\mathrm{dom}(X)} q_\alpha(x_\alpha|\eta(\alpha, X_{n-1}, Y, \phi)) \qquad (7.6)$$

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., 2017b], 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? Were there to be merely good, broadly applicable heuristics, 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, 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/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

parametrization.

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 [**BP:** *cite?*]. 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] introduce 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

$$\begin{aligned}\log p(Y;\theta) &\geq \log p(Y;\theta) - D_{\mathrm{KL}}\left(q(X|Y;\phi) \,||\, p(X|Y;\theta)\right) && (7.7)\\ &= \int q(X|Y;\phi)\left[\log p(X,Y;\theta) - \log q(X|Y;\phi)\right] dX && (7.8)\\ &= \mathrm{ELBO}(\theta,\phi,Y). && (7.9)\end{aligned}$$

[**BP:** *is this necessary? I think it would be fine to just define the ELBO itself...*] Here it is assumed the generative model is differentiable with respect to its parameters θ. If the sampling process for drawing a random 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 $g(Y,\epsilon;\phi) \overset{d}{=} q(X|Y;\phi)$, the evidence lower bound $\mathrm{ELBO}(\theta,\phi,Y)$ can be optimized via gradient ascent techniques using

$$\begin{aligned}\nabla_{\theta,\phi}\,\mathrm{ELBO}(\theta,\phi,Y) &= \nabla_{\theta,\phi}\mathbb{E}_{q(X|Y;\phi)}\left[\log \frac{p(X,Y;\theta)}{q(X|Y;\phi)}\right]\\ &= \mathbb{E}_{p(\epsilon)}\left[\nabla_{\theta,\phi}\log \frac{p(g(Y,\epsilon;\phi),Y;\theta)}{q(g(Y,\epsilon;\theta)|Y;\phi)}\right]\\ &\approx \frac{1}{L}\sum_{\ell=1}^{L}\left[\nabla_{\theta,\phi}\log \frac{p(g(Y,\epsilon^{\ell};\phi),Y;\theta)}{q(g(Y,\epsilon^{\ell};\phi)|Y;\phi)}\right],\end{aligned}$$

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 could potentially increase their expressivity and thereby performance. 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 from above 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 [**?**] provides implementations of random variable primitives which can be incorporated into deep generative models; Edward [**?**] 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 [**?**] and probabilistic Torch [**?**] [**BP:** *include. . . ?*]) are built on top of it. 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.

[**BP:** *Here we could include a paragraph describing the objective function used in Sid's NIPS paper, if that is considered in-scope.*]

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 evidence bound 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

[**brooks:** *Place discussion here about challenges for HMC and BBVI, give some lead-in to next section*]

7.4 Nesting

One benefit of using an expressive probabilistic programming language is that it makes it easy to develop and reason about complex models. In particular, such PPLs help us to explore models defined in terms of inference results on other nested models. In these models, latent random variables or likelihood scores are defined by means of the expectations of some functions over those nested models. A good example is Goodman and his colleagues' models for agents that capture agents' knowledge about other agents, such as what an agent A knows about what an agent B knows about an agent C [Stuhlmüller and Goodman, 2014]. The model for the agent B in this case includes her knowledge about the agent C, but this knowledge is represented by means of an expection over the model for C. [**FW:** *Should put a source-code example here, perhaps from Tom's thesis or from forestdb.org*] Operationally, such models including other nested models can be understood as samplers that invoke inference algorithms over those nested models while producing a single sample. Invoking an inference algorithm this way is allowed in probabilistic programming languages like Anglican, WebPPL and Venture, so that these models can be easily expressed in those languages.

However, the recent studies have shown that nested models should be used with due diligence. First, Staton et al. spotted that posterior distribution fails to be defined in some model and observation because the marginal likelihood of the observation is infinite in that case [Staton et al., 2016, Staton, 2017]. Here is a variant of their example:

```
(let [x (sample (normal 0 1))
      v (exp (- (/ (* x x) 2)))]
  (observe (exponential (/ 1 v)) 0)
```

```
x)
```

This program defines a model with prior $p(x) = \mathit{normal}(x; 0, 1)$ and likelihood $p(y|x) = \mathit{exponential}(y; 1/(\sqrt{2\pi} \cdot p(x)))$, and expresses that $y = 0$ is observed. The model fails to have a posterior because its marginal likelihood at $y = 0$ is infinite:

$$\begin{aligned} p(y = 0) = \int p(x, y = 0)\mathrm{d}x &= \int p(x)p(y = 0|x)\mathrm{d}x \\ &= \int p(x) \cdot \frac{1}{\sqrt{2\pi} \cdot p(x)}\mathrm{d}x = \infty. \end{aligned}$$

Staton et al. argued that the formal semantics of a language construct for performing inference, such as `doquery` in Anglican, should account for this failure case. The 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 the prior of this outer model may become undefined. Second, Rainforth et al. analyzed the convergence properties of algorithms that perform multiple inference tasks in a nested fashion [Rainforth et al., 2017]. In particular, they studied a naive nested Monte-Carlo algorithm that generates a single sample from a model after drawing multiple samples from another nested model and computing an estimate based on these samples. Rainforth et al. pointed out that the mean square error of this naive nested Monte-Carlo has a very bad convergence rate, and suggested to identify additional properties of the target model, such as linearity or Taylor expansion, and to convert the estimation problem over the model into an equivalent one that does not involve a nested use of Monte-Carlo algorithms. Currently probabilistic programming languages perform inference on nested models in a way similar to the naive nested Monte-Carlo, and so they may suffer from its inefficiency. A good research direction is to develop an inference algorithm taylored for such nested models, possibly in the style of what Rainforth et al. have suggested.

7.5 Formal Semantics

Programs in probabilistic programming languages mean probabilistic models, and computing information about these models is the goal of inference algorithms. In most cases, the computed information 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 or inference algorithms and need to ensure that these implementations do not have bugs. An optimizer should not change the probabilistic models denoted by programs, and an inference algorithm should be able to handle corner cases correctly, such as the program in Section 7.4 that does not have the posterior distribution. To meet this obligation, the developers need a method for mapping probabilistic programs to their precise meanings, i.e., their probabilistic models. The method does not have to be computable, but it should be formal and unambiguous, so that it can serve as a judge whenever there is a dispute on the correctness of an implementation.

A formal semantics is such a method. It defines the mathematical meaning of every program in a probabilistijc 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, $normal(x; 1, \sqrt{2})$), or to its unnormalized counterpart $normal(x; 0, 1) \times normal(2; x, 1)$, 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 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. Good places to see these issues are 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, and which are fully or partially supported by popular 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 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 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 well-known impossibility result by Aumann [1961]:

Theorem 7.1 (Aumann)**.** Let F be the set of measurable functions on $(\mathbb{R}, \mathcal{B})$. Then, no matther 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 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 intepret HOPPL programs, such as the application function from above, as good functions.

More concretely, Heunen et al. 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 to read Heunen et al.'s original article [Heunen et al., 2017] and the one by Scibior et al. [2018].

8

Conclusion

Congratulations for having made it this far. Having made it through all this way we can now summarize probabilistic programming relatively concisely.

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 probabilistic inference, particularly Bayesian inference and conditioning. While the correspondence between first-order probabilistic programming languages and graphical models means that research to improve inference algorithms for graphical models applies more-or-less directly to probabilistic programming systems, the same is not true for HOPPLs. The challenge there, potentially infinite-dimensional parameter spaces,

Our focus throughout has mostly been on automating inference in known and fixed models.

Appendices

.1 Probability distributions

Distribution	Parameters	Support	PDF/PMF	Mean	Variance/-Covariance
Bernoulli	$\theta \in [0,1]$	$x \in \{0,1\}$	$\begin{cases}\theta & \text{if } x=1\\ 1-\theta & \text{if } x=0\end{cases}$	θ	$\theta(1-\theta)$
Beta	$\alpha,\beta>0$	$x \in [0,1]$	$\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}x^{\alpha-1}(1-x)^{\beta-1}$	$\frac{\alpha}{\alpha+\beta}$	$\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$
Binomial	$N \in \mathbb{N}, \theta \in [0,1]$	$x \in \{0,\dots,N\}$	$\binom{N}{x}\theta^x(1-\theta)^{N-x}$	$N\theta$	$N\theta(1-\theta)$
Gamma	$\alpha,\beta>0$	$x>0$	$\frac{\beta^\alpha}{\Gamma(\alpha)}x^{\alpha-1}\exp(-\beta x)$	$\frac{\alpha}{\beta}$	$\frac{\alpha}{\beta^2}$
Categorical	$\boldsymbol{\theta} \in [0,1]^K, \sum_k \theta_k = 1$	$x \in \{1,\dots,K\}$	θ_x	N/A	N/A
Dirichlet	$\boldsymbol{\alpha} \in (0,\infty)^K$	$\boldsymbol{x} \in [0,1]^K, \sum_k x_k = 1$	$\frac{\Gamma\left(\sum_k \alpha_k\right)}{\prod_k \Gamma(\alpha_k)}\prod_k x_k^{\alpha_k-1}$	$\frac{\boldsymbol{\alpha}}{\sum_k \alpha_k}$	$\mathrm{Var}[x_k] = \frac{\alpha_k(\alpha_0-\alpha_k)}{\alpha_0^2(\alpha_0+1)}$
Multivariate Normal	$\boldsymbol{\mu} \in \mathbb{R}^d$, $\boldsymbol{\Sigma}$ positive semi-definite $d \times d$	$\boldsymbol{x} \in \mathbb{R}^d$	$\frac{1}{\sqrt{\lvert 2\pi\boldsymbol{\Sigma}\rvert}}\exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right)$		
Normal	$\mu \in \mathbb{R}$, $\sigma^2>0$	$x \in \mathbb{R}$	$\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$	μ	σ^2
Poisson	$\lambda>0$	$x \in \{0,1,2,\dots\}$	$\frac{\lambda^x}{x!}\exp(-\lambda)$	λ	λ

Table 1: Summary of common probability distributions

a

[a] Where $\Gamma(z) := \int_0^\infty x^{z-1}e^{-x}\,\mathrm{d}x$ is the *Gamma function*.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. 2015.

Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. Justin Kelly, 1996.

Marco Alberti, Giuseppe Cota, Fabrizio Riguzzi, and Riccardo Zese. Probabilistic logical inference on the web. In *AI* IA 2016 Advances in Artificial Intelligence*, pages 351–363. Springer, 2016.

Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 2006. ISBN 978-0-521-03311-4.

R. J. Aumann. Borel structures for function spaces. *Illinois Journal of Mathematics*, 5:614–630, 1961.

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. *arXiv preprint arXiv:1502.05767*, 2015.

Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.

Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. Measure transformer semantics for bayesian machine learning. *Logical Methods in Computer Science*, 9(3), 2013.

Yuri Burda, Roger Grosse, and Ruslan Salakhutdinov. Importance weighted autoencoders. In *ICLR*, 2016.

Elie Bursztein, Jonathan Aigrain, Angelika Moscicki, and John C Mitchell. The end is nigh: Generic solving of text-based captchas. In *WOOT*, 2014.

Mario Lezcano Casado. Compiled inference with probabilistic programming for large-scale scientific simulations. Master's thesis, University of Oxford, 2017.

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *In Neural Information Processing Systems, Workshop on Machine Learning Systems*, 2016.

Julien Cornebise, Éric Moulines, and Jimmy Olsson. Adaptive methods for sequential importance sampling with application to state space models. *Statistics and Computing*, 18:461–480, 2008.

Julien Cornebise, Éric Moulines, and Jimmy Olsson. Adaptive sequential Monte Carlo by means of mixture of experts. *Statistics and Computing*, 24: 317–337, 2014.

Cameron Davidson-Pilon. *Bayesian methods for hackers: probabilistic programming and Bayesian inference*. Addison-Wesley Professional, 2015.

Peter Dayan, Geoffrey E Hinton, Radford M Neal, and Richard S Zemel. The Helmholtz machine. *Neural Computation*, 7(5):889–904, 1995.

Pierre Del Moral and Lawrence M Murray. Sequential Monte Carlo with highly informative observations. *SIAM/ASA Journal on Uncertainty Quantification*, 3(1):969–997, 2015.

Arnaud Doucet, Simon Godsill, and Christophe Andrieu. On sequential Monte Carlo sampling methods for Bayesian filtering. *Statistics and computing*, 10(3):197–208, 2000.

Daniel P Friedman and Mitchell Wand. *Essentials of programming languages*. MIT press, 2008.

Hong Ge, Adam Scibior, and Zoubin Ghahramani. Turing: rejuvenating probabilistic programming in Julia. In submission.

Timon Gehr, Sasa Misailovic, and Martin Vechev. PSI: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*, pages 62–83. Springer, 2016.

Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. Bayesian data analysis, 3rd edition, 2013.

S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–741, 1984.

Samuel J Gershman and Noah D Goodman. Amortized inference in probabilistic reasoning. In *Proceedings of the Thirty-Sixth Annual Conference of the Cognitive Science Society*, 2014.

Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459, 2015.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. In *Proc. 24th Conf. Uncertainty in Artificial Intelligence (UAI)*, pages 220–229, 2008.

Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014a. Accessed: 2017-8-22.

Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014b. Accessed: 2016-8-27.

Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181. ACM, 2014.

Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

Heikki Haario, Eero Saksman, and Johanna Tamminen. An adaptive Metropolis algorithm. *Bernoulli*, pages 223–242, 2001.

Ralf Herbrich, Tom Minka, and Thore Graepel. TrueSkill$^{\text{TM}}$: A Bayesian skill rating system. In *Advances in Neural Information Processing Systems*, pages 569–576, 2007.

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 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*, pages 1–12, 2017.

Rich Hickey. The clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 1:1–1:1, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-270-2. . URL `http://doi.acm.org/10.1145/1408681.1408682`.

Mark Johnson, Thomas L Griffiths, and Sharon Goldwater. Adaptor grammars: A framework for specifying compositional nonparametric Bayesian models. In *Advances in neural information processing systems*, pages 641–648, 2007.

Matthew J Johnson, D Duvenaud, AB Wiltschko, SR Datta, and RP Adams. Structured vaes: Composing probabilistic graphical models and variational autoencoders. *NIPS 2016*, 2016.

Angelika Kimmig and Luc De Raedt. Probabilistic logic programs: Unifying program trace and possible world semantics. 2017.

Angelika Kimmig, Bart Demoen, Luc De Raedt, Vitor Santos Costa, and Ricardo Rocha. On the implementation of the probabilistic logic programming language problog. *Theory and Practice of Logic Programming*, 11 (2-3):235–262, 2011.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

Diederik P Kingma and Max Welling. Auto-encoding variational Bayes. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.

Daphne Koller and Nir Friedman. Probabilistic graphical models: principles and techniques, 2009.

Daphne Koller, David McAllester, and Avi Pfeffer. Effective Bayesian inference for stochastic programs. *AAAI*, pages 740–747, 1997. ISSN 01635980. . URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.7936$\delimiter"026E30F$npapers2://publication/uuid/0259F2B2-CDE1-459B-BC6A-FD482DAC0394`.

Tejas D Kulkarni, William F Whitney, Pushmeet Kohli, and Josh Tenenbaum. Deep convolutional inverse graphics network. In *Advances in Neural Information Processing Systems*, pages 2539–2547, 2015a.

Tejas Dattatraya Kulkarni, Pushmeet Kohli, Joshua B Tenenbaum, and Vikash Kumar Mansinghka. Picture: a probabilistic programming language for scene perception. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015b.

Krzysztof Łatuszyński, Gareth O Roberts, Jeffrey S Rosenthal, et al. Adaptive gibbs samplers and related mcmc methods. *The Annals of Applied Probability*, 23(1):66–98, 2013.

Tuan Anh Le, Atilim Günes Baydin, Robert Zinkov, and Frank Wood. Using synthetic data to train neural networks is model-based reasoning. *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 3514–3521, 2017a.

Tuan Anh Le, Atõlõm G§ne? Baydin, and Frank Wood. Inference Compilation and Universal Probabilistic Programming. In *20th International Conference on Artificial Intelligence and Statistics, April 20–22, 2017, Fort Lauderdale, FL, USA (To Appear)*, 2017b.

Tuan Anh Le, Maximilian Igl, Tom Jin, Tom Rainforth, and Frank Wood. Auto-encoding sequential monte carlo. *arXiv preprint arXiv:1705.10306*, 2017c.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

M. Lezcano-Casado, A.G. Baydin, D. Martinez-Rubio, T.A. Le, F. Wood, G. Heinrich, L. and Louppe, K. Cranmer, K. Ng, W. Bhimji, and Prbhat. Improvements to inference compilation for probabilistic programming in large-scale scientific simulators. In *NIPS Workshop on Deep Learning for the Physical Sciences*, 2017.

Chris J Maddison, Dieterich Lawson, George Tucker, Nicolas Heess, Mohammad Norouzi, Andriy Mnih, Arnaud Doucet, and Yee Whye Teh. Filtering variational objectives. *arXiv preprint arXiv:1705.09279*, 2017.

Vikash Mansinghka, Tejas D Kulkarni, Yura N Perov, and Josh Tenenbaum. Approximate bayesian image interpretation using generative probabilistic graphics programs. In *Advances in Neural Information Processing Systems*, pages 1520–1528, 2013.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv*, page 78, March 2014. URL `http://arxiv.org/abs/1404.0099`.

Vikash Mansinghka, Richard Tibbetts, Jay Baxter, Pat Shafto, and Baxter Eaves. BayesDB: A probabilistic programming system for querying the probable implications of data. *arXiv preprint arXiv:1512.05006*, 2015.

a McCallum, K Schultz, and S Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems*, volume 22, pages 1249–1257, 2009. ISBN 9781615679119. URL `papers2://publication/uuid/6C6DEB21-3B95-495A-97C3-2C4C3957E157`.

Geoffrey McLachlan and David Peel. *Finite mixture models*. John Wiley & Sons, 2004.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG : Probabilistic Models with Unknown Objects. In *IJCAI*, 2005.

T Minka, J Winn, J Guiver, and D Knowles. Infer .NET 2.4, Microsoft Research Cambridge, 2010a.

Tom Minka and John Winn. Gates. In *Advances in Neural Information Processing Systems*, pages 1073–1080, 2009.

Tom Minka, J Winn, J Guiver, and D Knowles. Infer.NET 2.4, 2010. Microsoft Research Cambridge, 2010b.

Kevin P Murphy. Machine learning: a probabilistic perspective, 2012.

L. M. Murray. Bayesian state-space modelling on high-performance hardware using LibBi. *arXiv preprint arXiv:1306.3277*, 2013.

Christian A Naesseth, Scott W Linderman, Rajesh Ranganath, and David M Blei. Variational sequential monte carlo. *arXiv preprint arXiv:1705.11140*, 2017.

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 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*, pages 62–79. Springer, 2016. . URL `http://dx.doi.org/10.1007/978-3-319-29604-3_5`.

Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. R2: An efficient MCMC sampler for probabilistic programs. In *AAAI*, pages 2476–2482, 2014.

OpenBugs. Pumps: conjugate gamma-poisson hierarchical model, 2009. Available online at `http://www.openbugs.net/Examples/Pumps.html`.

B. Paige and F. Wood. A compilation target for probabilistic programming languages. In *JMLR; ICML 2014*, pages 1935–1943, 2014.

Brooks Paige and Frank Wood. Inference networks for sequential Monte Carlo in graphical models. In *Proceedings of the 33rd International Conference on Machine Learning*, volume 48 of *JMLR: W&CP*, pages 3040–3049, 2016.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

Y. Perov and F. Wood. Automatic sampler discovery via probabilistic programming and approximate Bayesian computation. In *Artificial General Intelligence*, pages 262–273, 2016.

Avi Pfeffer. IBAL: A probabilistic rational programming language. *IJCAI International Joint Conference on Artificial Intelligence*, pages 733–740, 2001. ISSN 10450823.

Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, 2009. URL `http://www.cs.tufts.edu/~nr/cs257/archive/avi-pfeffer/figaro.pdf$\delimiter"026E30F$npapers2://publication/uuid/0E83E526-451F-41EA-ACBE-7150FF7584D4`.

Avi Pfeffer. *Practical probabilistic programming*. Manning Publications Co., 2016.

Martyn Plummer. 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*, pages 20–22, 2003.

Lawrence R Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

Tom Rainforth, Robert Cornish, Hongseok Yang, Andrew Warrington, and Frank Wood. On the Opportunities and Pitfalls of Nesting Monte Carlo Estimators, 2017. Available at `https://arxiv.org/abs/1709.06181`.

Rajesh Ranganath, Sean Gerrish, and David M Blei. Black box variational inference. *International Conference on Machine Learning*, 2014.

Carl Edward Rasmussen and Zoubin Ghahramani. Occam's razor. In *Advances in neural information processing systems*, pages 294–300, 2001.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. 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*, pages 1278–1286, 2014.

Daniel Ritchie, Ben Mildenhall, Noah D Goodman, and Pat Hanrahan. Controlling procedural modeling programs with stochastically-ordered sequential Monte Carlo. *ACM Transactions on Graphics (TOG)*, 34(4):105, 2015.

Daniel Ritchie, Andreas Stuhlmüller, and Noah Goodman. C3: Lightweight incrementalized mcmc for probabilistic programs using continuations and callsite caching. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, pages 28–37, 2016a.

Daniel Ritchie, Anna Thomas, Pat Hanrahan, and Noah Goodman. Neurally-guided procedural models: Amortized inference for procedural graphics programs using neural networks. In *Advances In Neural Information Processing Systems*, pages 622–630, 2016b.

John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. Probabilistic programming in python using pymc3. *PeerJ Computer Science*, 2:e55, 2016.

Taisuke Sato and Yoshitaka Kameya. PRISM: A language for symbolic-statistical modeling. *IJCAI International Joint Conference on Artificial Intelligence*, 2:1330–1335, 1997. ISSN 10450823.

Adam Scibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. Denotational validation of higher-order bayesian inference. *PACMPL*, 2(POPL):60:1–60:29, 2018.

Frank Seide and Amit Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4232-2. .

David J Spiegelhalter, Andrew Thomas, Nicky G Best, and Wally R Gilks. BUGS: Bayesian inference using Gibbs sampling, Version 0.50, 1995.

Stan Development Team. Stan: A C++ Library for Probability and Sampling, Version 2.4, 2014.

Sam Staton. 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*, pages 855–879, 2017.

Sam Staton, Hongseok Yang, Frank Wood, Chris Heunen, and Ohad Kammar. 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*, pages 525–534, 2016.

Andreas Stuhlmüller and Noah D Goodman. Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs. *Cognitive Systems Research*, 28:80–99, 2014.

Joshua B Tenenbaum, Charles Kemp, Thomas L Griffiths, and Noah D Goodman. How to grow a mind: Statistics, structure, and abstraction. *science*, 331(6022):1279–1285, 2011.

S. Thrun. 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), 2000. ISSN 1050-4729. .

Adrien Todeschini, François Caron, Marc Fuentes, Pierrick Legrand, and Pierre Del Moral. Biips: Software for Bayesian inference with interacting particle systems. *arXiv preprint arXiv:1412.3779*, 2014.

David Tolpin, Jan Willem van de Meent, Hongseok Yang, and Frank Wood. Design and implementation of probabilistic programming language Anglican. *arXiv preprint arXiv:1608.05263*, 2016.

Dustin Tran, Matthew D Hoffman, Rif A Saurous, Eugene Brevdo, Kevin Murphy, and David M Blei. Deep probabilistic programming. *arXiv preprint arXiv:1701.03757*, 2017.

Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam C Pocock, Stephen Green, and Guy L Steele. Augur: Data-Parallel Probabilistic Modeling. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2600–2608. Curran Associates, Inc., 2014.

Rudolph Van Der Merwe, Arnaud Doucet, Nando De Freitas, and Eric Wan. The unscented particle filter. In *Advances in Neural Information Processing Systems*, pages 584–590, 2000.

David Wingate and Theophane Weber. Automated variational inference in probabilistic programming. *arXiv preprint arXiv:1301.1299*, 2013.

David Wingate, Andreas Stuhlmueller, and Noah D Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the 14th international conference on Artificial Intelligence and Statistics*, page 131, 2011.

F Wood, JW van de Meent, and V Mansinghka. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*, pages 1024–1032, 2014. URL `http://jmlr.org/proceedings/papers/v33/wood14.pdf`.

F Wood, JW van de Meent, and V Mansinghka. A new approach to probabilistic programming inference. *arXiv preprint arXiv:1507.00996*, 2015.