Gen: A High-Level Programming Platform for Probabilistic Inference

by

Marco Francis Cusumano-Towner

B.S., University of California, Berkeley (2011)
M.S., Stanford University (2013)

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

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author .....
Department of Electrical Engineering and Computer Science
August 28, 2020

Certified by .....
Vikash K. Mansinghka
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Gen: A High-Level Programming Platform for Probabilistic Inference

by

Marco Francis Cusumano-Towner

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 2020, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

Probabilistic inference provides a powerful theoretical framework for engineering intelligent systems. However, diverse modeling approaches and inference algorithms are needed to navigate engineering tradeoffs between robustness, adaptability, accuracy, safety, interpretability, data efficiency, and computational efficiency. Structured generative models represented as symbolic programs provide interpretability. Structure learning of these models provides data-efficient adaptability. Uncertainty quantification is needed for safety. Bottom-up, discriminative inference provides computational efficiency. Iterative "model-in-the-loop" algorithms can improve accuracy by fine-tuning inferences and improve robustness to out-of-distribution data. Recent probabilistic programming systems fully or partially automate inference, but are too restrictive for many applications. Differentiable programming systems are also inadequate: they do not support structure learning of generative models or hybrids of "model-in-the-loop" and discriminative inference. Therefore, probabilistic inference is still often implemented by translating tedious mathematical derivations into low-level numerical programs, which are error-prone and difficult to modify and maintain.

This thesis presents the design and implementation of the Gen programming platform for probabilistic inference. Gen automates the low-level implementation of probabilistic inference algorithms while remaining flexible enough to support heterogeneous algorithmic approaches and extensible enough for practical inference engineering. Gen users define their models explicitly using probabilistic programs, but instead of compiling the model directly into an inference algorithm implementation, Gen compiles the model into data types that encapsulate low-level inference operations whose semantics are derived from the model, like sampling, density evaluation, and gradients. Users write their inference application in a general-purpose programming language using Gen's abstract data types as primitives. This thesis defines Gen's data types and shows that they can be used to compose a variety of inference techniques including sophisticated Monte Carlo algorithms and hybrids of Monte Carlo, variational, and discriminative techniques. The same data types can be generated from multiple probabilistic programming languages that strike different expressiveness and performance tradeoffs. By decoupling probabilistic programming language implementations from inference algorithm design, Gen enables more flexible specialization of both, leading to performance improvements over existing probabilistic programming systems.

Thesis Supervisor: Vikash K. Mansinghka
Title: Principal Research Scientist

Acknowledgments

My primary thesis advisor, Vikash Mansinghka, planted the seeds for the research described in this thesis and created an environment in which this work was possible. In the beginning of my PhD, he directed me to pull at research threads that seemed to never end, resulting in the most intellectually rewarding phase of my life. The research problem and the approach of this thesis build directly on his earlier work, and are a product of a new probabilistic programming research paradigm that he has persistently worked to shape. I am also grateful for his example of independent thinking and entrepreneurial attitude, his conscientiousness as an advisor, and his efforts to support his students in non-technical challenges. The past five years would have been much harder without knowing that I could count on him.

Several other advisors and mentors have played roles in my graduate school journey. Josh Tenenbaum has inspired me for years with his insightful work; and led me into probabilistic programming. I am grateful for his approachable demeanor, encouragment, and continuing inspiration over the past five years. I am also grateful to Martin Rinard and Michael Carbin for being on my thesis committee and helping me to communicate this work more effectively. Martin Rinard gave valuable advice on the core terminology, framing, and exposition in this document. Martin Vechev gave helpful mentorship and advice during our collaboration, which built the foundation for the 'trace translator' construct in this thesis. I also thank Sam Gershman for his advice and mentorship and support as I started my graduate career at MIT, Sivan Bercovici for supporting my transition back into graduate school, and Pieter Abbeel for providing my first exposure to computer science research.

The first few years of graduate school would not have been the same without the friendship and support of my cohort of fellow PhD students and researchers, including Feras Saad, Alex Lew, and Ulrich Schaechtle. Feras has been a good friend and ally during the ups and downs of the past five years. Alex has been a constant source of encouragement, and late night white board discussions with him helped to clarify many of the ideas in this thesis. Feras and Alex were crucial in helping to push the Gen PLDI paper over the finish line, and in various other efforts. The research in this thesis was also aided by discussions with many other current and former affiliates of the MIT Probabilistic Computing Project, including Alexey Radul and Anthony Lu, and with support from Amanda Brower, Rachel Paiste, and many others. Several people contributed to the applications of Gen described in this thesis. Ben Zinberg, Austin Garrett, and Javier Felip Leon contributed to the scene graph inference application; Ulrich Schaechtle and Feras Saad devised the algorithm used in the Gaussian process structure learning application. Since Gen was released, many people including Alex Lew, Ben Zinberg, Tan Zhi-Xuan, George Matheos, and Sam Witty have helped to improve it and their contributions and use of Gen have been incredibly encouraging. Alex contributed syntax improvements that are reflected in this thesis.

This thesis would not have been possible without the support of my parents Maria Cusumano and Mark Towner who consistently put me before themselves and did everything in their power help me succeed. Finally, I would not have embarked on this PhD journey without Lisa Bashkirova, who has been a constant source of good advice for the past decade.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Probabilistic inference in generative models is a core part of the modern toolkit for automated reasoning from data. Probabilistic inference plays a central role in Bayesian statistics, machine learning, and signal processing and enables applications in various fields where uncertainty is important, from robotics and autonomous vehicles [43] to biomedicine [106], natural sciences [57] and data analysis [39]. However, implementing probabilistic inference algorithms from scratch involves error-prone and tedious mathematical derivations and numerical programming. As a result, many software tools in recent decades have attempted to automate probabilistic inference [49]. Some tools have been widely adopted within certain fields [20], but are restricted in the type of inference problems they solve and their performance characteristics. Deep learning, which overlaps with probabilistic inference, is well-supported by differentiable programming languages [11, 1, 94], but these languages have limited support for inference in structured generative models, which can be more robust, interpretable, and data-efficient than deep learning.[1]

This thesis describes a new approach to implementing probabilistic inference in generative models (Figure 1-1), and a system called Gen [32] that implements the approach. As in many recent *probabilistic programming* systems, users of Gen explicitly define their generative model by writing a probabilistic program. However, instead of attempting to solve the inference problem for the user, Gen automatically generates data types from the probabilistic program that can be used to compose inference algorithms. The user implements their inference application using the generated data types. The current Gen implementation generates Julia [12] data types for implementing inference algorithms, but a similar approach can be implemented in other languages. The data types automate the low-level details of inference algorithm implementations, resulting in shorter and more maintainable inference code relative to implementations written from scratch, while maintaining the user's freedom to specialize the algorithm to their model and to flexibly integrate the algorithm into their application. This thesis describes abstract data types for probabilistic inference, a number of inference recipes that use the data types, how the data types are implemented by the compilers of modeling languages, and several applications of the resulting system.

[1] The approach proposed in this thesis complements existing programming languages and software tools for deep learning. The current Gen implementation is interoperable with TensorFlow [1] and PyTorch [94].

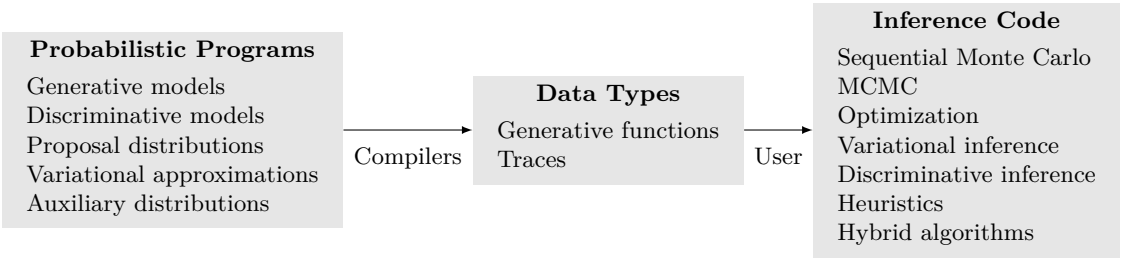


Figure 1-1: The high-level inference implementation approach proposed in this thesis

1.1 A new approach to implementing probabilistic inference

A defining feature of probabilistic modeling and inference is the use of a declarative mathematical representation for uncertain knowledge in the form of a probabilistic model that is distinct from inference queries made on that model and the algorithms that are used to answer them [66]. However, while the model is separate from the inference algorithm in the mind of the inference practitioner as they devise the algorithm, the separation is often lost during implementation. The algorithm is often first specialized to the model on pencil and paper and then implemented in a way that obscures the original model and makes it difficult to modify the model or the algorithm. For example, the data structures used to store the latent state may be specialized to the structure of the model, making it challenging to change the modeling approach without rewriting the implementation; or the implementation might exploit cancellations of factors that make it unclear how a change in the model that invalidates the cancellation should be reflected in the inference code.

Since probabilistic models are mathematically well-defined objects (probability distributions), it is in principle possible to address these issues using inference 'solvers' that take as input a machine-readable formal representation of a model and observed data, and automate inference using general-purpose inference algorithms. This is the approach taken by many probabilistic programming systems [49, 95, 51, 20]. While this approach can be effective for specialized classes of inference problems and applications, probabilistic inference encompasses a very broad set of problems ranging from real-time object tracking to program synthesis to Bayesian statistics. No fixed set of solution strategies is sufficient to meet the requirements of such a diverse range of applications, and manual specialization of the algorithm to the model is important for acceptable performance in many applications. As a result, despite the invention of many probabilistic programming systems in recent years, probabilistic inference algorithms are still often implemented from scratch.

Probabilistic programming researchers have begun to explore the design space in between solver-oriented probabilistic programming and hand-coded probabilistic inference implementations by extending probabilistic programming languages with frameworks and domain-specific languages for customizing inference algorithms. This approach has been called *programmable inference* [80], in contrast to solver-oriented probabilistic programming systems that use built-in algorithms. The resulting systems can be more flexible than solver-oriented systems, while automating the low-level details of the algorithm's implementation. However these systems remain too rigid for many inference applications because they force

the user to write their inference application in a framework that takes control over the flow of the application away from the user. Domain-specific languages for describing inference algorithms can also be restrictive and difficult to extend.

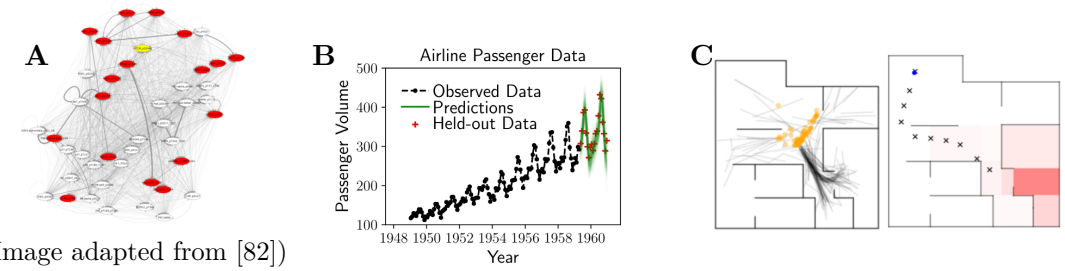
This thesis builds on research in probabilistic programming systems and programmable inference, but employs a different approach that favors flexibility and extensibility over automation. The approach is based on the observation that many Monte Carlo and variational inference techniques can be implemented using a core set of primitive data types and low-level operations. The key idea is to automatically generate code for these data types and operations from an explicit machine-readable representation of the user's model, such as a probabilistic program. The role of the operations is analogous to the role of automatic differentiation in deep learning algorithms, but for probabilistic inference algorithms. After defining the model, users implement an inference algorithm in a general-purpose programming language using the data types and operations that were generated from the model. This approach to implementing inference algorithms enjoys many of the benefits of existing probabilistic programming systems, including the presence of an explicit definitive representation of the user's probabilistic model, automation of low-level computations associated with the model, and the ability to easily use models that possess structure uncertainty. However, the approach avoids the algorithmic rigidity of existing probabilistic programming systems, and is better suited for integration into inference applications. Table 1.1 lists several applications that exercise the modeling and inference flexibility afforded by this approach and have been implemented using Gen, either by the author or others.

New data types for probabilistic inference: generative functions and traces The approach of this thesis is based on two new data types. The first data type the *generative function* and the second data type is the *trace*.[2] A generative function is an object that represents a generative probabilistic model. A trace is an object that represents a sample from a generative probabilistic model that includes assignments to all latent and observed variables. Generative functions support operations that generate traces in different ways, including by unconditional sampling (e.g. from the prior distribution), and by stochastically filling in a trace given values for some subset of the random variables. Traces support operations that include computing the log density of the model, taking gradients of the log density, and updating the values of some subset of the random variables. Generative functions and traces are *abstract data types* [75], which means that the user of these data types does not need to know about the data structures that are used internally to store the latent and observed variables or how the operations are implemented. Generative function and trace data types are often implemented by compiling a probabilistic program that encodes a model, but they can be implemented in other ways as well, and the inference code that uses them will run independently of how they are implemented. Chapter 2 gives a mathematical definition of generative functions and traces based on probability distributions on dictionaries that is flexible enough to represent generative models with structure uncertainty including models defined as programs with stochastic control flow.

[2]The word 'trace' has been used in probabilistic programming before. In this thesis a 'trace' is an instance of the trace abstract data type, which has specific mathematical content and a set of supported operations.

Application of Gen	Modeling approach	Inference approach
Estimating causal effects in the presence of latent confounders [129]	Hierarchical latent variable Gaussian process model	Elliptical slice sampling [88] and Metropolis-Hastings
(**A**) Inferring cell signaling pathways from time series [81]	Dynamic Bayesian network with prior on edge presence	Custom Metropolis-Hastings moves on network structure
(**B**) Time series interpretation and forecasting (Section 6.5)	Probabilistic context-free grammar prior on GP covariance functions [113]	Reversible jump MCMC [53]
(**C**) Human activity understanding and trajectory prediction (Section 6.1)	Generative simulator-based models of rational behavior using rapidly exploring random trees [31]	SMC [33]; reversible jump MCMC; surrogate models trained on simulated data; dynamic programming
(**D**) Inferring goals of boundedly rational agents for compositional tasks [133]	General-purpose planners; priors on reward functions; dynamic Bayesian networks	SMC; rejuvenation MCMC kernels with heuristic-based data-driven proposals
(**E**) Inferring physical relationships between objects from RGB-D video [134] (Section 6.4)	Dynamic Bayesian networks over scene graph structure and 6DoF poses; robust likelihood models	Deep neural network pose estimator [124]; particle filtering; reversible jump MCMC
(**F**) Inferring articulated 3D human body pose from depth images	Prior on articulated body pose; 3D rendering of articulated body model [67]	Deep neural network proposal trained on simulated data; importance sampling

Table 1.1: Selected Gen applications that use diverse modeling and inference approaches



(Image adapted from [82])

(Image adapted from [133])

Figure 1-2: Illustrations of selected Gen applications from Table 1.1

Efficient inference over symbolic structures Because generative functions and traces support models with structure uncertainty, they can be used to implement probabilistic inference in nonparametric Bayesian models and inference over compositional symbolic structures like the source code of programs. Section 3.7 presents an interface for specifying custom trans-dimensional Markov chain Monte Carlo (MCMC) kernels including arbitrary reversible jump [53] kernels and a procedure that uses generative functions and traces to automate the low-level implementation of these kernels, including computation of the acceptance probability. This interface is significantly more flexible than existing interfaces for specifying MCMC kernels in probabilistic programming systems, and can be used to construct algorithms that explore the space of structures more efficiently than the generic MCMC kernels used in many prior probabilistic programming systems [51, 130, 52, 79].

Robust hybrids of Monte Carlo and deep learning inference techniques Discriminative models and generative models have complementary strengths. Discriminative approaches are often relatively fast but require training time and data. Inference that uses the generative model "in the loop" is often able to more robustly generalize to unlikely observations, and structured generative models can be more easily modified. In Gen, generative functions and traces can be constructed for either generative models or discriminative models, and used together in inference code. It is therefore straightforward to implement hybrid inference algorithms using generative functions and traces that take advantage of the best aspects of both approaches. Section 3.2 and Section 3.4.2 and Section 3.5 describe procedures based on generative functions and traces for implementing importance sampling, MCMC, and sequential Monte Carlo algorithms respectively using arbitrary proposal distributions that can be based on discriminative models like deep neural networks or hand-crafted heuristics. Section 3.3 shows how generative functions that represent discriminative models can be trained using traces simulated from a generative model, which is a form of amortized variational inference [58, 118, 16].

Surrogate modeling and bridging multiple latent representations Unlike in solver-oriented probabilistic programming systems, there is no single probabilistic model in a Gen inference application. Instead, generative models are simply a type of data, and like for the combinations of discriminative and generative models described above, it is possible to construct inference algorithms that make use of multiple generative models. Section 3.6 describes a construct called *trace translators* that allows traces of one generative model to be translated into traces of another generative model, even when the models use different latent representations. Section 3.6.4 shows a sequential Monte Carlo algorithm based on trace translators, and includes an example that first performs inference in a coarse-grained surrogate model that was trained offline on data from a fine-grained model, and then translates traces of the surrogate model into traces of the fine-grained model.

Modularity and reuse via internal proposals Chapter 4 extends the generative function and trace data types with the ability to encapsulate *internal proposal* distributions that generate values for some random variables given values for others. Because internal proposals are encapsulated with the model and implement a common specification, a

user's inference code can use the internal proposal without knowing its details. Modeling languages automatically implement default internal proposals, but users can also override internal proposal distributions with more efficient custom distributions, and inference code that uses the model will automatically benefit from the improved proposal.

Extensibility and performance Because generative functions and traces are *abstract* data types, they can be implemented in a variety of ways. Gen includes multiple probabilistic modeling languages with different strengths and weaknesses that all compile into generative function and trace data types (Chapter 5). Users can choose the modeling language that best suits their model. For example, if the model includes stochastic control flow and black-box simulators, then a Turing-universal modeling language can be used. If the model does not require these features then a more specialized modeling language can be used that gives better performance. The system is also straightforward to extend with new modeling languages. Gen has already been extended with plugins that allow generative functions to be constructed from TensorFlow [1] and PyTorch [94] models. It is also possible to hand-code optimized implementations of the generative function and trace data types later in the development lifecycle to improve performance, without modifying the inference code. Finally, generative functions are composable—so generative functions written in different modeling languages can invoke one another, and the user can choose the appropriate modeling language for different parts of their model.

A platform for investigating probabilistic computational rationality Studies in computational rationality [46] aim to understanding how natural and artificially intelligent agents can make decisions while accounting for the costs and feasibility of computation. Gen and the approach to implementing probabilistic inference described in this thesis are natural tools for studying computational rationality because of the flexible support for bottom-up, heuristic, model-free, and discriminative inference techniques and top-down model-based techniques, and hybrids of these approaches. The ability to write inference code in a general-purpose language allows for open exploration of heterogeneous inference architectures that may be necessary for optimal resource-constrained decision-making [107], while still using probabilistic knowledge representations (generative functions).

Open-source implementation and modeling and inference ecosystem The ideas in this thesis are the basis for an open-source implementation[3] of Gen that is embedded in the Julia language [12]. The Julia implementation of Gen is itself the basis for a set of libraries for probabilistic modeling and inference that use Gen's core data types of generative functions and traces. Examples include plugins for wrapping TensorFlow and PyTorch models in generative functions, and modeling and inference libraries for probabilistic reasoning about three-dimensional objects and scenes. Implementations of Gen in other languages are in progress. Chapter 6 describes selected applications using the Julia implementation, and the other chapters of the thesis give pedagogical code examples that also use the syntax of this implementation. The Julia implementation has been used

[3] https://gen.dev

to teach probabilistic inference by groups at multiple universities, has an active community of users and enthusiastic contributors, and has already been used in several research projects [81, 133, 14, 128, 129, 134].

1.2 Overview of programming languages concepts in Gen

This section gives an overview of some of the key ideas in Gen's design from a programming languages perspective using pedagogical examples to introduce new concepts when possible.

As described above, Gen's design is based on new abstract data types (ADTs) for probabilistic programs (the *generative function* ADT) and execution traces of probabilistic programs (the *trace* ADT). These ADTs provide a set of primitive operations for implementing inference algorithms that include generating execution traces, querying execution traces for the value of random choices and their gradients, and updating execution traces. Implementations of these ADTs are typically automatically generated from the source code of a probabilistic program. Gen also provides multiple probabilistic programming languages that all generate the same ADTs but strike different tradeoffs between ease-of-use and performance. Furthermore, these languages are interoperable so that different parts of a model can be described in different languages, and users can also hand-implement the ADTs for performance-critical parts of their model. While Gen is already a practical tool for prototyping probabilistic inference algorithms, Gen's design provides fertile ground for future research on new probabilistic programming languages with more efficient implementations of Gen's ADTs, and in analyzing, compiling, optimizing, and verifying high-level user inference code that is expressed in terms of Gen's ADTs.

As mentioned above, Gen builds on the programmable inference paradigm in probabilistic programming [80] as exemplified by Venture [79, 78]. However, Gen's design differs significantly from that of Venture, which does not provide comparable ADTs for probabilistic models and execution traces. For example, while the runtime systems of other probabilistic programming languages including Venture perform automatic differentiation, they do not expose an operation to users that is analogous to Gen's 'gradient' trace ADT operation, which encapsulates the details of how automatic differentiation is performed, and presents a simple interface for obtaining gradients associated with an execution trace. Similarly, while Venture's runtime system internally performs incremental computation, it does not expose an operation to users that is analogous to Gen's 'update' trace ADT operation, which presents a simple interface for making incremental updates to a trace. The same is true of Gen's other core trace operations. Also Venture does not support multiple interoperable probabilistic programming languages, used a restricted domain-specific language for inference, and does not support many of the inference techniques supported by Gen that are important for efficient inference, including custom proposal distributions.

While this thesis gives mathematical descriptions of Gen's ADTs and shows how various inference algorithms can be decomposed into operations for the ADTs, it does not give formal semantics for Gen's probabilistic programming languages, or attempt to statically verify probabilistic inference code. Lew et al. [73] give the formal semantics for, and describes static verification of, a more restricted probabilistic modeling and inference system

that is embedded in Haskell and was based in part on Gen. Gen uses dynamic checks for some invariants in certain inference operations to detect user errors, but allows for arbitrary user code to be combined with Gen inference code, and allows users to use custom implementations of Gen's ADTs that would be difficult to verify.

The main programming languages contributions of this thesis include:

1. The design of Gen, which combines probabilistic programming languages for specifying models with data types for implementing inference algorithms in a general-purpose programming language.

2. New abstract data types for probabilistic programs and their execution traces that encapsulate the interpretation and compilation of probabilistic programming languages, and are sufficient for implementing a broad array of flexible Monte Carlo and variational inference algorithms.

3. The first probabilistic programming system with multiple interoperable probabilistic programming languages that use different compilers and strike different expressiveness-performance tradeoffs.

4. The first probabilistic programming system to support several inference techniques that are routinely used in practice including arbitrary reversible jump MCMC samplers and custom data-driven MCMC proposals expressed as probabilistic programs.

5. The first probabilistic programming language to support encapsulation of inference logic within modeling components via a novel construct called *internal proposals*.

6. An inference programming construct called *trace translators* that allows for inference programs to use multiple models of the same domain using a differentiable programming language for bijections between spaces of traces of two probabilistic programs.

This section introduces the key concepts behind some of these contributions for programming language researchers, and provides necessary background in probabilistic modeling and inference as needed.

1.2.1 Generative probabilistic models and probabilistic inference

First, we give an overview of the mathematical formalism that the thesis uses to describe probabilistic models and probabilistic inference. The full formalism is given in Chapter 2.

Probability distributions on dictionaries This thesis defines probabilistic inference as conditional probabilistic inference in a generative probabilistic model. We define a probabilistic model as a probability distribution p on dictionaries (i.e. finite associative arrays), denoted $\boldsymbol{\sigma}$ or $\boldsymbol{\tau}$ or $\boldsymbol{\upsilon}$, from some set of 'addresses' (i.e. keys) to values. We denote a literal dictionary that contains two addresses `a` and `b` with (Boolean) values `T` and `F` respectively, by $\boldsymbol{\tau} = \{\mathtt{a} \mapsto \mathtt{T}, \mathtt{b} \mapsto \mathtt{F}\}$. Each entry in one of these dictionaries encodes some piece of information about the state of the domain being modeled. The model p expresses our assumptions about the probable states of the domain, prior to observing any data.

Dictionaries form a flexible sample space because a given piece of information (a given address) may be present in some states but not in others. For example consider the model p defined below, which is a probability distribution over a set of six possible states:

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau})$
$\{\mathtt{a} \mapsto \mathtt{F}, \mathtt{c} \mapsto \mathtt{F}\}$	0.45
$\{\mathtt{a} \mapsto \mathtt{F}, \mathtt{c} \mapsto \mathtt{T}\}$	0.05
$\{\mathtt{a} \mapsto \mathtt{T}, \mathtt{b} \mapsto \mathtt{F}, \mathtt{c} \mapsto \mathtt{F}\}$	0.05
$\{\mathtt{a} \mapsto \mathtt{T}, \mathtt{b} \mapsto \mathtt{F}, \mathtt{c} \mapsto \mathtt{T}\}$	0.2
$\{\mathtt{a} \mapsto \mathtt{T}, \mathtt{b} \mapsto \mathtt{T}, \mathtt{c} \mapsto \mathtt{F}\}$	0.125
$\{\mathtt{a} \mapsto \mathtt{T}, \mathtt{b} \mapsto \mathtt{T}, \mathtt{c} \mapsto \mathtt{T}\}$	0.125

The left column lists the possible states, and the right column contains the probabilities for each state, which sum to one.[4] For this model the states contain entries with Boolean values (`T` or `F`). Four of the states contain an entry for address `b` and two do not.

Generative probabilistic models and conditioning The probabilistic models p considered in this thesis are *generative* because they specify normalized probability distributions on dictionaries that contain both observable and non-observable (or 'latent') entries. Observed data takes the form of a dictionary $\boldsymbol{\rho}$ that assigns values to some subset of the addresses. For example, we might obtain observed data $\boldsymbol{\rho} = \{\mathtt{c} \mapsto \mathtt{T}\}$. The goal of probabilistic inference is to answer queries about the probable values of unobserved addresses, in light of the observed data $\boldsymbol{\rho}$ and the model p that specifies how they are related. This is made precise by the *conditional distribution*, a probability distribution over unobserved dictionaries that is denoted $\boldsymbol{\sigma}$. The conditional distribution $p(\cdot|\boldsymbol{\rho})$ is defined by (i) marking each state $\boldsymbol{\tau}$ with whether it matches the observed data or not, and (ii) normalizing the probability over the unobserved part of all states that do match the observed data. We assume the observed addresses are present in all states, so for example, the address `b` could not be observed for the example model p defined earlier. For this p and $\boldsymbol{\rho} = \{\mathtt{c} \mapsto \mathtt{T}\}$, the conditional distribution is computed as follows:

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau})$	$\boldsymbol{\tau}$ matches $\boldsymbol{\rho}$
$\{\mathtt{a} \mapsto \mathtt{F}, \mathtt{c} \mapsto \mathtt{F}\}$	0.45	✗
$\{\mathtt{a} \mapsto \mathtt{T}, \mathtt{c} \mapsto \mathtt{T}\}$	0.05	✓
$\{\mathtt{a} \mapsto \mathtt{F}, \mathtt{b} \mapsto \mathtt{F}, \mathtt{c} \mapsto \mathtt{F}\}$	0.05	✗
$\{\{\mathtt{a} \mapsto \mathtt{T}, \mathtt{b} \mapsto \mathtt{F}, \mathtt{c} \mapsto \mathtt{T}\}$	0.2	✓
$\{\mathtt{a} \mapsto \mathtt{T}, \mathtt{b} \mapsto \mathtt{T}, \mathtt{c} \mapsto \mathtt{F}\}$	0.125	✗
$\{\mathtt{a} \mapsto \mathtt{T}, \mathtt{b} \mapsto \mathtt{T}, \mathtt{c} \mapsto \mathtt{T}\}$	0.125	✓

$\longrightarrow$

$\boldsymbol{\sigma}$	$p(\boldsymbol{\sigma}\|\boldsymbol{\rho})$
$\{\mathtt{a} \mapsto \mathtt{F}\}$	0.05/0.375
$\{\mathtt{a} \mapsto \mathtt{T}, \mathtt{b} \mapsto \mathtt{F}\}$	0.2/0.35
$\{\mathtt{a} \mapsto \mathtt{T}, \mathtt{b} \mapsto \mathtt{T}\}$	0.125/0.375

There are a number of queries that we may want to make given a probabilistic model p and the observed data $\boldsymbol{\rho}$. We may want to know the *expected value*, under the conditional distribution, of some *test function* g on the set of unobserved states ($\sum_{\boldsymbol{\sigma}} g(\boldsymbol{\sigma}) p(\boldsymbol{\sigma}|\boldsymbol{\rho})$). For

[4] The thesis includes a measure-theoretic mathematical framework, but in this section we assume that probability distributions are discrete to simplify the notation and make it more broadly accessible.

example, when g is an indicator function of an event, its expected value is the conditional probability of the event. For the conditional distribution defined above, and $g(\boldsymbol{\sigma}) := [\boldsymbol{\sigma}[\texttt{a}] = \texttt{T}]$, the expected value is:

$$p(\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}\}|\{\texttt{c} \mapsto \texttt{T}\}) + p(\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{T}\}|\{\texttt{c} \mapsto \texttt{T}\}) = 0.325/0.375 = 0.86\bar{6}$$

So the conditional probability that `a` is true, given that `c` is true, is about 0.87. Another common task is computing the *marginal likelihood* of the data, which is the sum of the probabilities of all states $\boldsymbol{\tau}$ that match the observed data. This is a quantitative measure of how well the model explains the data. For our example, the marginal likelihood is

$$p(\{\texttt{a} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{T}\}) + p(\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{T}\}) + p(\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{T}, \texttt{c} \mapsto \texttt{T}\}) = 0.375$$

Gen can help users solve these two types of tasks (and others), but for the remainder of this introduction we will define probabilistic inference as the process of evaluating the expected value of a test function g under the conditional distribution induced by a probabilistic model p and observed data $\boldsymbol{\rho}$.

1.2.2 Using probabilistic programming languages to express generative probabilistic models

The previous section introduced generative probabilistic models and conditioning using probability distributions represented as tables, but probability tables are an impractical medium for expressing probabilistic models because they scale exponentially in the number of addresses, and cannot represent continuous probability distributions. This thesis builds on prior work on probabilistic programming languages, and uses general-purpose executable programming languages with random choice to encode probabilistic models.

A probabilistic programming language Gen provides a probabilistic programming language called the Dynamic Modeling Language that extends the syntax of the Julia language [12]. The major syntactic extension to Julia is the addition of an extra expression type, called a *labeled random choice expression*, which has the form:

```
{address} ~ bernoulli(0.5)
```

The part of the expression to the left of $\sim$ and inside the braces encodes a dynamically computed address for a randomly chosen value, and the part of the expression to the right of $\sim$ encodes a probability distribution from which the value should be sampled. For example '`{:a} ~ bernoulli(0.5)`' samples a Boolean-valued Bernoulli random choice at address[5] `a` and '`{5} ~ bernoulli(0.5)`' samples a random choice from the same distribution, but at address `5`. There is a syntactic sugar that also assigns the value of the random choice to a variable in the program, and uses the variable name as the address:

`a ~ bernoulli(0.5)` is equivalent to `a = ({:a} ~ bernoulli(0.5))`

[5]The colon syntax is (e.g. `:a`) is syntax for a Julia symbol, or interned string.

Users construct a probabilistic program that defines their probabilistic model using a combination of these random choice expressions, Julia expressions, and Julia control flow. For example, consider the Dynamic Modeling Language program below:

```
@gen function burglary_model()
    burglary ~ bernoulli(0.01)
    if burglary
        disabled ~ bernoulli(0.1)
    else
        disabled = false
    end
    if !disabled
        alarm ~ bernoulli(if burglary 0.94 else 0.01 end)
    else
        alarm = false
    end
    calls ~ bernoulli(if alarm 0.70 else 0.05 end)
    return nothing
end
```

This program defines a simple probabilistic model of a scenario in which a burglar may be breaking in to one's home when one is away. There is a low probability (0.01) that there is a burglary. If there is a burglary, then the burglar may have disabled the alarm, with probability 0.1. If the alarm was not disabled, then there is some probability (0.94) that it is triggered. But there is also a small probability (0.01) that the alarm is triggered via a false positive when there is no burglary. If the alarm sounds, then there is a probability (0.70) that a neighbor calls you on the phone. But there is also a small probability (0.05) that the neighbor calls even if the alarm does not sound.

Trace-based semantics and generative functions The semantics of a program $\mathfrak{p}$ in this language is denoted $[\![\mathfrak{p}]\!]$ and has two parts. The first part is the probability distribution p on dictionaries from which the following process samples: Execute the program, but intercept random choice expressions, and record the randomly sampled value for each random choice in a dictionary $\boldsymbol{\tau}$ (called a *trace*) that maps addresses of random choices to their values; when the program finishes executing, return the dictionary $\boldsymbol{\tau}$. To be valid, the program must never attempt to record a value at the same address twice in an execution, and the program must finish executing with probability 1. The second part of the semantics is the function f that maps $\boldsymbol{\tau}$ to the return value of the program. In Gen, the return values of probabilistic programs exist to allow probabilistic programs to be composed by calling other probabilistic programs (e.g. via sequencing). In summary, probabilistic programs in Gen encode a mathematical object that contains a probability distribution p on dictionaries, and a function f from dictionaries to a return value.[6]. We call this type of mathematical object a *generative function*, denoted $\mathcal{P} = (p, f)$. Therefore, the semantic function $[\![\cdot]\!]$ of a probabilistic programming language in Gen maps the source code of a probabilistic program to a generative function. This thesis does not formally define the

[6]This is a simplified definition. It defined first in Chapter 2 and then extended in Chapter 4.

semantic function for Gen's probabilistic programming languages, which extend the Julia language. Instead, we define the semantic function for a simple toy language, and refer the interested reader to prior works on trace-based semantics for an understanding of how to define trace-based semantics for more complex languages [15, 73].

For probabilistic program source code given in this thesis, the name that is assigned to the function in the source code will denote the generative function that is derived from the source code via the semantic function. So, the semantics of the program above is a generative function `burglary_model` $= (p, f)$ where (`nothing` is a built-in Julia 'null' value):

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau})$	$f(\boldsymbol{\tau})$
$\{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{calls} \mapsto \texttt{F}\}$	$0.99 \cdot 0.99 \cdot 0.95$	`nothing`
$\{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{calls} \mapsto \texttt{T}\}$	$0.99 \cdot 0.99 \cdot 0.05$	`nothing`
$\{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{T}, \texttt{calls} \mapsto \texttt{F}\}$	$0.99 \cdot 0.01 \cdot 0.30$	`nothing`
$\{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{T}, \texttt{calls} \mapsto \texttt{T}\}$	$0.99 \cdot 0.01 \cdot 0.70$	`nothing`
$\{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{calls} \mapsto \texttt{F}\}$	$0.01 \cdot 0.9 \cdot 0.06 \cdot 0.95$	`nothing`
$\{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{calls} \mapsto \texttt{T}\}$	$0.01 \cdot 0.9 \cdot 0.06 \cdot 0.05$	`nothing`
$\{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{T}, \texttt{calls} \mapsto \texttt{F}\}$	$0.01 \cdot 0.9 \cdot 0.94 \cdot 0.30$	`nothing`
$\{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{T}, \texttt{calls} \mapsto \texttt{T}\}$	$0.01 \cdot 0.9 \cdot 0.94 \cdot 0.70$	`nothing`
$\{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{T}, \texttt{calls} \mapsto \texttt{F}\}$	$0.01 \cdot 0.1 \cdot 0.95$	`nothing`
$\{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{T}, \texttt{calls} \mapsto \texttt{T}\}$	$0.01 \cdot 0.1 \cdot 0.05$	`nothing`

Note that probabilistic programs in this language can also take arguments, in which case their probability distribution on dictionaries is parametrized by possible values of their arguments x (denoted $p(\cdot; x)$), and the function f maps arguments x and dictionaries $\boldsymbol{\tau}$ to return values. The approach to defining semantics via a probability distribution on traces is called *trace semantics*, and is well-covered formally in the probabilistic programming literature. The most important difference between the probabilistic programming languages used in Gen and most other probabilistic programming languages with trace-based semantics is that Gen's probabilistic programming languages use user-defined addresses for random choices (instead of the address being determined by its order in the execution [15] or its structural location in the abstract syntax tree [127]). It is important that the addresses of random choices in Gen are intuitive for the user, because in Gen, users write inference code that refers to specific random choices by their addresses.

Observed data and test functions are external to the probabilistic program Another departure from many recent probabilistic programming languages is that Gen's probabilistic programming languages do not include *observe* or *factor* statements. Therefore, programs always define normalized probability distributions p from which it is trivial to sample by running the program. Only by pairing a program with a dictionary $\boldsymbol{\rho}$ containing observed data can we define a conditional distribution. For example, we can represent the observation that we did received a phone call from a neighbor with the dictionary $\boldsymbol{\rho} = \{\texttt{calls} \mapsto \texttt{T}\}$. The conditional distribution induced by `burglary_model` together with

the observed data $\boldsymbol{\rho}$ is shown in the table below:

$\boldsymbol{\sigma}$	$p(\boldsymbol{\sigma}\|\boldsymbol{\rho})$
$\{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}\}$	0.7912
$\{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{T}\}$	0.1119
$\{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}\}$	0.0004
$\{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{T}\}$	0.0956
$\{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{T}\}$	0.0008

Note that in addition to the observed data, the test function g is also separate from the program that defines the generative model. For example, if we are interested in the conditional probability that there was a burglary, given that we received a phone call, the test function is $g(\boldsymbol{\sigma}) = [\boldsymbol{\sigma}[\texttt{burglary}]]$, which evaluates to 0 if $\boldsymbol{\sigma}[\texttt{burglary}] = \texttt{F}$ and 1 if $\boldsymbol{\sigma}[\texttt{burglary}] = \texttt{T}$. For this choice of g and $\boldsymbol{\rho} = \{\texttt{calls} \mapsto \texttt{T}\}$, the expected value is:

$$\begin{aligned}\sum_{\boldsymbol{\sigma}} g(\boldsymbol{\sigma}) p(\boldsymbol{\sigma}|\boldsymbol{\rho}) = \; & p(\{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}\}|\boldsymbol{\rho}) + \\ & + p(\{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{T}\}|\boldsymbol{\rho}) + \\ & + p(\{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{T}\}|\boldsymbol{\rho}) \\ = \; & 0.097\end{aligned}$$

The choices to define the observed data $\boldsymbol{\rho}$ and the test function g separately from the probabilistic program are motivated by modularity—the observed data and the test function frequently change for a given fixed model, and it is preferable to not have to modify the model code when the observed data or test function changes. Also, as we will see later, generative functions can be used to express auxiliary probability distributions that must support straightforward sampling, and Gen opts to use the same probabilistic programming languages for models and for these auxiliary distributions. Finally, Gen also uses data that is simulated from generative models as part of training components of inference algorithms. Excluding observe and factor statements from the modeling language ensures that it is always possible to generate simulated observed data simply by running the program.

Dictionaries	$\boldsymbol{\rho}, \boldsymbol{\sigma}, \boldsymbol{\tau}, \boldsymbol{\upsilon}$
Literal dictionaries	$\{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}\}$
Probability distributions	p, q
Instances of generative function ADT	$\mathcal{P} = (p, f), \mathcal{Q} = (q, f)$
Map from probabilistic programs to generative functions	$[\![\cdot]\!]$
Generative function ADT operations	SIMULATE, GENERATE
Instances of trace ADT	$\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau}), \mathbf{s} = (\mathcal{Q}, x, \boldsymbol{\sigma})$
Trace ADT operations	LOGPDF, CHOICES, UPDATE

Figure 1-3: Notation used in this introductory section

1.2.3 Abstract data types for generative functions and traces

This thesis is concerned largely with the design and implementation of an API for implementing Monte Carlo and variational probabilistic inference algorithms (Monte Carlo algorithms are the focus of the thesis). The design of Gen's API is based on the observation that these algorithms can be broken down into a small set of primitive operations whose semantics are derived from the trace-based semantics of probabilistic programs. The API consists of two abstract data types (ADTs)—one for generative functions and one for execution traces of generative functions. These ADTs encapsulate and automate low-level computations in inference algorithms, so that code that uses these ADTs is relatively high-level, abstract, resembles algorithm pseudocode, and has reduced surface area for bugs. The ADTs also provide a clean abstraction barrier that separates probabilistic programming language implementation and compiler architecture (which are internal to the implementation of the ADTs) from probabilistic inference algorithms (which are specified using the ADTs).

This section describes a simplified version of the generative function and trace ADTs, and the next section introduces how they can be used to perform probabilistic inference. The body of the thesis describes the full set of ADTs operations, and shows how they can be used for a broad set of inference algorithms. Note that *it is not necessary to understand approximate inference algorithms in order to understand and implement these ADTs.* This an important benefit of Gen's design, because it allows probabilistic programming language design and implementation to be decoupled from the design of inference algorithms.

In Gen, the compiler for a probabilistic programming language generates special generative function and trace ADTs for each probabilistic program. The data stored in a *generative function ADT* is a generative function $\mathcal{P} = (p, f)$, as defined above. The data stored in a *trace ADT* is a tuple $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$ where $\mathcal{P}$ is a generative function, x are arguments to the function, and the dictionary $\boldsymbol{\tau}$ stores the value of each random choice made during a possible execution of the generative function (i.e. $\boldsymbol{\tau}$ such that $p(\boldsymbol{\tau}; x) > 0$). We now introduce a subset of the operations supported by these ADTs.

Simulate operation The first operation supported by the generative function ADT is $\mathcal{P}.\textsc{simulate}(x)$, which takes arguments x to the generative function, samples a dictionary of random choices $\boldsymbol{\tau}$ according to the distribution p, and returns the resulting trace $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$. This operation allows us to sample execution traces from generative functions.

Example: For $\mathcal{P} =$ `burglary_model` a call to $\mathcal{P}.\textsc{simulate}(x)$ returns one of its 10 possible traces. It returns the trace $\mathbf{t} = (\mathcal{P}, x, \{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{calls} \mapsto \texttt{F}\})$ with probability $0.99 \cdot 0.99 \cdot 0.95$, and returns the trace $\mathbf{t} = (\mathcal{P}, x, \{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{calls} \mapsto \texttt{T}\})$ with probability $0.99 \cdot 0.99 \cdot 0.05$, and so on.

Generate operation The second generative function ADT operation, $\mathcal{P}.\textsc{generate}(x, \boldsymbol{\tau})$, also returns an execution trace $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$, but instead of sampling the random choices $\boldsymbol{\tau}$ according to p, it takes them as input. This operation may seem overly simple. It will be extended in the thesis with the ability to take a *partial* dictionary that only contains some of the choices and fill in the rest stochastically. We do not formalize this here to keep the

introduction accessible.

Example: For $\mathcal{P}$ = `burglary_model`, the call $\mathcal{P}$.GENERATE(x, {`burglary` $\mapsto$ `F`, `alarm` $\mapsto$ `F`, `calls` $\mapsto$ `F`}) returns the trace $\mathbf{t} = (\mathcal{P}, x, \{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{calls} \mapsto \texttt{F}\})$.

Logpdf operation The first operation supported by the trace ADT is $\mathbf{t}$.LOGPDF(), which returns the log probability $\log p(\boldsymbol{\tau}; x)$ that the random choices in the trace would have been sampled if x were the arguments to the generative function. This is typically the sum of log-probabilities for each random choice made.

Example: For $\mathcal{P}$ = `burglary_model` and $\mathbf{t} = (\mathcal{P}, x, \{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{calls} \mapsto \texttt{F}\})$, we have $\mathbf{t}$.LOGPDF() $= \log(0.99 \cdot 0.99 \cdot 0.95)$.

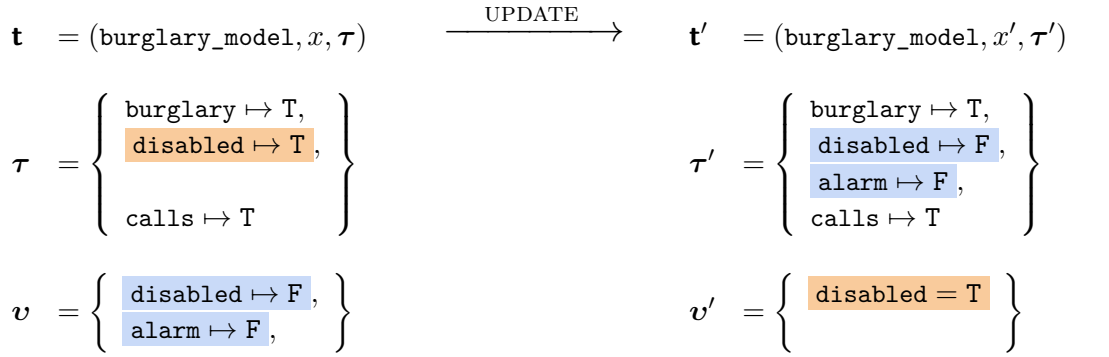
Choices operation The second operation supported by the trace ADT is $\mathbf{t}$.CHOICES(), which for $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$ returns the dictionary $\boldsymbol{\tau}$.

Example: For $\mathbf{t} = (\mathcal{P}, x, \{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{calls} \mapsto \texttt{F}\})$ we have $\mathbf{t}$.CHOICES() $= \{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{calls} \mapsto \texttt{F}\}$.

Update operation The third trace ADT operation is $\mathbf{t}$.UPDATE($x', \delta_X, \boldsymbol{\upsilon}$). This operation is more complex—it allows the arguments to, and the random choices made by, an execution trace to be modified. Suppose $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$. The first argument to this operation (x') provides new arguments to the generative function, which may be different from the arguments x that are stored in the initial execution trace $\mathbf{t}$. The second argument is a *change hint* δ_X that provides optional information about the difference between the original arguments x and the new arguments x' that allows an implementation to perform the operation more efficiently via incremental computation. The third argument $\boldsymbol{\upsilon}$ is a dictionary that contains entries for any addresses that are included in the previous trace but whose value should be changed, as well as entries for any addresses that were not included in the previous trace but should be included in the new trace. The operation returns $(\mathbf{t}', \boldsymbol{\upsilon}', \log w, \delta_Y)$ where $\mathbf{t}' = (\mathcal{P}, x', \boldsymbol{\tau}')$ is a new trace with choices $\boldsymbol{\tau}'$ constructed from $\boldsymbol{\tau}$ and $\boldsymbol{\upsilon}$. The other arguments are metadata about the change from $\mathbf{t}$ to $\mathbf{t}'$. In particular, $\log w$, is the log ratio of the new probability to the previous probability ($\log(p(\boldsymbol{\tau}'; x')/p(\boldsymbol{\tau}; x))$), and $\boldsymbol{\upsilon}'$ is the dictionary, that if passed to the update operation on $\mathbf{t}'$ would reverse the update and result in trace $\mathbf{t}$. Note that this operation does not mutate $\mathbf{t}$. Traces are immutable.

Example: Figure 1-4 illustrates the application of the update operation to a trace $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$ where $\mathcal{P}$ = `burglary_model` and $\boldsymbol{\tau} = \{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{T}, \texttt{calls} \mapsto \texttt{T}\}$. Note that this generative function takes no arguments, so both x and x' are the empty tuple. We will focus on how $\boldsymbol{\upsilon}$ determines the change in the choices from $\boldsymbol{\tau}$ to $\boldsymbol{\tau}'$, and the log weight $\log w$. On the left of the figure is the initial trace $\mathbf{t}$, and the argument $\boldsymbol{\upsilon}$ that specifies the change to the choices. The left side also shows the lines of code in probabilistic program that defines `burglary_model`, with lines that are visited and sample random choices

shown black. The dictionary $\boldsymbol{v}$ indicates that the value at address `disabled` should be set to false. As shown on the code listing on the right, this causes the control flow in the program to change, and there is now a random choice sampled at address `alarm`. Therefore, the dictionary $\boldsymbol{v}$ also contains an entry for address `alarm`. The resulting dictionary $\boldsymbol{\tau}'$ is shown on the right. The ratio $p(\boldsymbol{\tau}'; x')/p(\boldsymbol{\tau}; x)$ is also shown at the bottom. Because only some of the random choices were changed from $\boldsymbol{\tau}$ to $\boldsymbol{\tau}$, two of the factors cancel in this ratio.




```
@gen function burglary_model()
    burglary ~ bernoulli(0.01)
    if burglary
        disabled ~ bernoulli(0.1)
    else
        disabled = false
    end
    if !disabled
        alarm ~ bernoulli(..)
    else
        alarm = false
    end
    call ~ bernoulli(..)
end
```

```
@gen function burglary_model()
    burglary ~ bernoulli(0.01)
    if burglary
        disabled ~ bernoulli(0.1)
    else
        disabled = false
    end
    if !disabled
        alarm ~ bernoulli(..)
    else
      alarm = false
    end
    call ~ bernoulli(..)
end
```

$$\frac{p(\boldsymbol{\tau}'; x')}{p(\boldsymbol{\tau}; x)} = \frac{\cancel{0.01}}{\cancel{0.01}} \cdot \frac{0.9}{0.1} \cdot 0.06 \cdot \frac{\cancel{0.05}}{\cancel{0.05}}$$

Figure 1-4: Illustration of the 'update' operation of the trace ADT

1.2.4 Generating implementations of the abstract data types from the source code of probabilistic programs

Implementations of the generative function and trace ADTs in the previous section are statically compiled from probabilistic programs. Because the data types are abstract, their implementations can be changed without requiring a change to the inference that uses them (we will introduce inference code that uses these ADTs shortly in Section 1.2.6). Chapter 5 describes several approaches for generating generative function and trace ADTs. In

particular, it presents two probabilistic programming languages that are part of Gen—the *Dynamic Modeling Language* (DML) that we saw earlier, and the *Static Modeling Language* (SML). These two languages strike different tradeoffs between expressiveness and performance of their ADT implementations. The syntax of DML includes most of Julia's syntax, and supports standard Julia control flow constructs including recursion. But the high expressiveness of this language makes generating efficient implementations of the ADTs more difficult because statically analyzing arbitrary general-purpose Julia code is difficult. In contrast, SML uses a restricted set of control flow constructs, and is amenable to straightforward static analysis that is used to statically specialize the implementation of the ADT and achieve better performance. But while SML has better performance, it is less natural to learn for users that are accustomed to programming in languages like Julia and Python. Because DML and SML produce the same ADT, the same user inference code can be applied to a models defined in either language. This allows users to start using the more familiar DML probabilistic programming language for their model, and then migrate to the more complex SML language if and when they need better performance, without modifying their inference code. Gen has also been extended with *domain-specific* modeling languages that implement the generative function and trace ADTs, and are more restrictive and more performant than even SML. Decoupling the probabilistic programming language compiler from user inference code is a distinctive feature of Gen's design that defines new directions for future work on compiling probabilistic programs.

Two probabilistic programs with the same semantics Consider the DML program below, that defines a generative function `hmm_dynamic` that describes a Hidden Markov Model (HMM), a classic generative probabilistic model:

```
1  @gen function hmm_dynamic()
2      z = 1
3      for i in 1:1000
4          z = ({:steps=>i=>:z} ~ categorical(A[z,:])
5          {:steps=>i=>:y} ~ categorical(B[z,:])
6      end
7  end
```

The variables `A` and `B` are the 'transition matrix' $\boldsymbol{A} \in \mathbb{R}^{100\times100}$ and 'emission matrix' $\boldsymbol{B} \in \mathbb{R}^{100\times50}$ of the HMM, with entries denoted $a_{i,j}$ and $b_{i,j}$. Each row of `A` (denoted `A[i,:]`) and each row of `B` (denoted `A[i,:]`) are vectors that sum to one, and represent probability distributions on 'hidden states' and 'observed states', respectively. There are 100 possible values of each hidden state, 50 possible values for each observed state, and 1000 time steps. The addresses of the random choices representing the hidden states are (`:steps=>i=>:z`) for each `i` ranging from 1 to 1000, and the addresses of the random choices representing the observable states are (`:steps=>i=>:y`) for each `i` ranging from 1 to 1000. Therefore, this program makes a total of 2000 random choices. Note that `=>` is Julia syntax for constructing a pair (i.e. 'cons'), so these addresses are linked lists with three entries. Any Julia value can be used as an address in a DML program. This program uses Julia's `for` loop to iterate through the time steps and consecutively (i) sample the current hidden state

from a discrete probability distribution (using `categorical`) that depends on the previous hidden state and (ii) sample the current observed state from a discrete distribution that depends on the current hidden state.

The SML probabilistic program below defines a generative function called `hmm_static` that is semantically equivalent to `hmm_dynamic`. That is, it defines the same probability distribution on dictionaries, and the same return value function (both programs do not have a return value and therefore return `nothing`).

```
1  @gen (static) function hmm_static()
2      steps ~ Unfold(step)(1000, 1)
3  end
4
5  @gen (static) function step(i, z_prev)
6      z ~ categorical(A[z_prev,:])
7      y ~ categorical(B[z,:])
8      return z
9  end
```

We defer an explanation of the syntax of SML to Chapter 5. But briefly, SML does not permit use of Julia `for` loops. Instead, this program uses a special construct called `Unfold` to express the sequence over steps. The program is also factored into two parts—an inner part that expresses what happens during each step (`step`) and the outer part that applies `step` sequentially using `Unfold`. Like the DML program, this program samples random choices at addresses (`:steps=>i=>:z`) and (`:steps=>i=>:z`) for each `i` ranging from 1 to 1000. Indeed, we used addresses of this form in the DML version of the program specifically so that the two programs would sample at the same addresses.

Dynamic Modeling Language compiler The DML compiler handles a very expressive language, but generates ADT implementations that are asymptotically inefficient. This compiler uses a generic Julia dictionary data type to store the random choices in the the trace. Most of the trace ADT operations, including UPDATE, are implemented by transforming the body of the program into an executable Julia function by replacing each random choice expression with a call to an effect handler. This is a very simple compilation strategy that makes it straightforward for DML to support all of Julia's control flow operations. However, this approach means that running an UPDATE operation involves performing an end-to-end execution of the transformed probabilistic program. Therefore, all UPDATE operations scale linearly in the number of addresses in the input trace, even when the optimal scaling behavior is sublinear. In particular, for updates to traces that only modify the values of a few addresses at a time (i.e. $\boldsymbol{v}$ has only a few entries), the generated ADT implementations are asymptotically inefficient relative to an optimal implementation.

Static Modeling Language compiler The SML compiler first translates the probabilistic program into an intermediate representation based on static directed acyclic dependency graphs, and generates a Julia record type that is specialized to the specific random choices made by the program to store the the random choices in the trace. The code for the UPDATE trace operation is generated via static analysis of the intermediate representation.

Specifically, static analysis of the conditional independence properties, which can be read off the static dependence graph, is used to avoid unnecessary re-execution of fragments of the probabilistic program that are executed in the code generated by the DML compiler. The resulting implementations of the ADT operations are asymptotically efficient and have lower constant-factor overhead because they use a more efficient underlying data structure for the trace that is specialized to the probabilistic model.

Hand-compiling implementations of the ADTs It is also possible to implement the generative function and trace ADTs directly for a probabilistic model, without using a probabilistic programming language or compiler. This is more involved, but allows the implementation to be more heavily specialized to the model, which can result in better performance than with an ADT generated from a probabilistic program using a compiler. The ability to migrate one's model from a probabilistic program based ADT implementation to a hand-coded ADT implementation is a useful affordance for performance optimization of inference code, especially in performance-constrained application settings. Note that the inference code that uses the ADT does not need to be modified when migrating from an ADT implementation based on probabilistic programs to a hand-coded ADT implementation.

Trace ADT implementation	**Time per operation**
Compiled from Dynamic Modeling Language	6.78ms
Compiled from Static Modeling Language	15.4μs
Hand-coded	1.65μs

Table 1.2: Performance of different implementations of the same trace ADT

Performance of different ADT implementations We now illustrate the potential for performance differences between different implementations of the trace ADT. Suppose we are given a trace $\mathbf{t}$ of $\mathcal{P} =$ `hmm_dynamic` or $\mathcal{P} =$ `hmm_static` containing the following choices:

$$\boldsymbol{\tau} = \left\{ \begin{array}{l} \texttt{(:steps=>1=>:z)} \mapsto z_1, \ldots, \texttt{(:steps=>1000=>:z)} \mapsto z_{1000}, \\ \texttt{(:steps=>1=>:y)} \mapsto y_1, \ldots, \texttt{(:steps=>1000=>:y)} \mapsto y_{1000} \end{array} \right\}$$

For this model, the probability of a trace is: $p(\boldsymbol{\tau}) = \prod_{i=1}^{1000} a_{z_{i-1},z_i} b_{z_i,y_i}$ where $z_0 := 1$. Suppose we want to run an UPDATE operation on this trace, where we update the value of the 346th hidden state from z_{346} to z'_{346}. That is:

$$\mathbf{t}.\text{UPDATE}(z', \delta_X, \boldsymbol{\upsilon}) \text{ where } \boldsymbol{\upsilon} = \{\texttt{(:steps=>346=>:z)} \mapsto z'_{346}\}$$

the resulting trace $\mathbf{t}' = (\mathcal{P}, x', \boldsymbol{\tau}')$ has choices:

$$\boldsymbol{\tau}' = \left\{ \begin{array}{l} \texttt{(:steps=>1=>:z)} \mapsto z'_1, \ldots, \texttt{(:steps=>1000=>:z)} \mapsto z'_{1000}, \\ \texttt{(:steps=>1=>:y)} \mapsto y'_1, \ldots, \texttt{(:steps=>1000=>:y)} \mapsto y'_{1000} \end{array} \right\}$$

where $z'_i = z_i$ for all $i \neq 346$, and $y'_i = y_i$ for all i. Recall that one of the outputs of the update operation is $\log w$, which is:

$$\log w = \log(p(\boldsymbol{\tau}')/p(\boldsymbol{\tau})) = \left(\sum_{i=1}^{1000} \log a_{z'_{i-1},z'_i} + \log b_{z'_i,y'_i}\right) - \left(\sum_{i=1}^{1000} \log a_{z_{i-1},z_i} + \log b_{z_i,y_i}\right)$$

The code for this operation that is generated by the DML Language compiler computes $\log w$ by computing each of the individual terms and summing. However, for this model and this particular call to UPDATE, most terms in the left-hand sum cancel with a term in the right-hand sum, because most of the z_i and z'_i are equal, and all of the y_i and y'_i are equal. Therefore, the log weight can be simplified to:

$$\log w = \log a_{z_{t+1},z'_t} - \log a_{z_{t+1},z_t} + \log b_{y_t,z'_t} - \log b_{y_t,z_t}$$

The implementation of the update operation that is generated by the SML compiler only computes the four necessary terms because it uses static analysis to identify the opportunity for cancellation. The implementation of the operation for a reasonable hand-coded implementation of the ADT has the same asymptotic scaling behavior of the SML implementation, but can have lower constant-factor overhead because it is manually specialized to the model. Table 1.2 shows a comparison of the performance of the different ADT implementations for this update operation, measured by taking the median running time across 1000 runs. The ADT implementation generated by the SML compiler is almost 500x faster than the implementation from the DML compiler. The hand-coded implementation is approximately 8x faster than the implementation generated by the SML compiler.

Interoperable probabilistic programming languages Gen's ADTs for generative functions and execution traces are compositional in the following sense. Consider a generative function that is constructed by sequencing two other generative functions. The ADT for the composite generative function can be implemented using the ADT operations of the two respective constituent generative functions, without needing the source code of the constituent functions. Gen's probabilistic programming languages have a language construct for invoking generative functions, which may have been compiled from the same, or different, probabilistic programming language, or hand-compiled. The resulting data structures used to implement trace ADTs store instances of the trace ADTs for the callee generative functions, called *subtraces*. The compositionality of Gen's ADTs allows the user to write a complex model that uses different probabilistic programming languages for different parts of the model. Also, users can implement specialized instances of the ADTs themselves for highly performance-sensitive parts of the model, if necessary. The interoperability between multiple probabilistic programming languages, and the interoperability with hand-coded implementations of the ADTs, gives users flexible routes to incremental performance optimization, which is important for (typically computationally intensive) probabilistic inference implementations.

1.2.5 Approximate probabilistic inference algorithms

Before discussing how the generative function and trace ADTs can be used to implement inference algorithms, we first provide background on inference algorithms, using our mathematical framework of probability distributions on dictionaries. Recall that we defined probabilistic inference as the task of evaluating the conditional expectation of a test function under the conditional distribution induced by a model and observed data. For this task, the answer is a real number. To perform this type of task some systems use symbolic analysis and simplification to try to simplify the symbolic expression down to a number (e.g. [42]). Although this approach is exact, it suffers from a major limitation—it may not always be possible to simplify fully, and systems based on this approach often take a long and unpredictable time to return an answer, or they time-out, or they return expressions that are not fully simplified, and therefore are not directly useful. This is problematic for many applications of probabilistic inference, including in performance-constrained and online settings, where timeouts or failures are not acceptable.

Monte Carlo approximate inference algorithms The challenges with exact probabilistic inference explain why most recent algorithmic work in probabilistic inference is concerned with approximation algorithms, with *Monte Carlo* and *variational* inference being the primary approaches, each with an extensive literature spanning decades [104, 126]. Gen supports both Monte Carlo and variational inference approaches, but this thesis will focus on Gen's support for Monte Carlo probabilistic inference algorithms. The Monte Carlo algorithms supported by Gen are randomized algorithms that return an approximation to the conditional distribution in the form of a collection of weighted samples of unobserved dictionaries $\boldsymbol{\sigma}_1, \ldots, \boldsymbol{\sigma}_n$ with weights $w_1, \ldots, w_n$. These collections of samples can then be used to estimate the expected value of a test function g by taking a weighted average:

$$\sum_{i=1}^{n} w_i g(\boldsymbol{\sigma}_i) \approx \sum_{\boldsymbol{\sigma}} p(\boldsymbol{\sigma}|\boldsymbol{\rho}) g(\boldsymbol{\sigma}) \tag{1.1}$$

The estimate is on the left-hand side of $\approx$, and the true value is on the right-hand side. For example, suppose we are using model `burglary_model` and we observe $\boldsymbol{\rho} = \{\texttt{calls} \mapsto \texttt{T}\}$, and we want to know the probability of a burglary. That is, the expected value of the test function $g(\boldsymbol{\sigma}) = [\boldsymbol{\sigma}[\texttt{burglary}]]$ under the conditional distribution $p(\cdot|\boldsymbol{\rho})$. Suppose we ran a Monte Carlo algorithm that returned the following data:

$$\begin{aligned}
\boldsymbol{\sigma}_1 &= \{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}\} && w_1 = 0.83 \\
\boldsymbol{\sigma}_2 &= \{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}\} && w_2 = 0.06 \\
\boldsymbol{\sigma}_3 &= \{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}\} && w_3 = 0.06 \\
\boldsymbol{\sigma}_4 &= \{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{T}\} && w_4 = 0.05
\end{aligned}$$

Then, the estimate would be:

$$\sum_{i=1}^{n} w_i g(\boldsymbol{\sigma}_i) = 0 \cdot 0.83 + 1 \cdot 0.06 + 1 \cdot 0.06 + 1 \cdot 0.05 = 0.17$$

Note that the estimate is stochastic—if we run the algorithm again, we will get a different estimate. There are two ways that such approximate inference algorithms can be evaluated, which focus on the asymptotic and non-asymptotic properties of the algorithm, respectively.

Asymptotically exact Monte Carlo inference algorithms First, we can ask whether the algorithm is *asymptotically exact*. That is, does the output of the algorithm converge almost-surely to the true value, as the computation budget n is increased to infinity (there may be other parameters of the algorithm that factor into the computation budget, but the number of samples n is the parameter that is used to characterize the asymptotic behavior):

$$\sum_{i=1}^{n} w_i g(\boldsymbol{\sigma}_i) \xrightarrow{\text{a.s.}} \sum_{\boldsymbol{\sigma}} p(\boldsymbol{\sigma}|\boldsymbol{\rho}) g(\boldsymbol{\sigma}) \tag{1.2}$$

Monte Carlo inference algorithms are often designed by construction to be asymptotically exact, as this provides some degree of confidence in the algorithm, and reduces questions of its correctness to questions of computation budget. There are several standard templates for constructing asymptotically exact Monte Carlo inference algorithms, including Markov chain Monte Carlo [121], importance sampling, and sequential Monte Carlo [33]. Whether or not an algorithm is indeed asymptotically exact depends on how these templates are instantiated, but the necessary conditions can be reasoned about statically. This reasoning is currently typically done manually by practitioners, but automatic static verification that the implementation of an approximate inference algorithm is asymptotically exact is a promising and active area of research [55, 7, 112, 73]. Dynamic statistical tests for detecting bugs that cause a failure to be asymptotically exact are widely used [47]. The emphasis on the asymptotic exactness of algorithms in the Monte Carlo literature is not surprising since much of the probabilistic inference methodology was developed for use in statistics, which is often not heavily performance constrained. For example, in statistics, practitioners often aim to (i) write asymptotically exact inference implementations, and to (ii) use a large enough computation budget so that the algorithm has (hopefully) converged to the asymptote (e.g. "overnight" [17]). Heuristic statistical tests can be used to detect lack of convergence in some cases [23].

Non-asymptotic approximation error of Monte Carlo inference algorithms In more performance-constrained applications of probabilistic inference like robotics, it is natural to ask about the *non-asymptotic approximation error* of the algorithm for some finite computation budget. In these settings, an approximate algorithm is run on a test problem for some computation budget and the results are compared with a reference value v, which ideally is the true value but is more often an estimate produced by an algorithm with large computation budget (the 'gold-standard'). Evaluation involves computing the difference between the reference value and the values produced by an algorithm being evaluated, which is run m times to account for its variability because it is a randomized algorithm.

The *average error* across m runs is one numeric summary of the algorithm's error:

$$\frac{1}{m}\sum_{j=1}^{m}\left|\left(\sum_{i=1}^{n} g(\boldsymbol{\sigma}_{i,j})w_{i,j}\right) - v\right| \tag{1.3}$$

Here $\boldsymbol{\sigma}_{i,j}$ and $w_{i,j}$ are the ith sample and weight from the jth run of the algorithm. This approach relies on having trust in the gold-standard algorithm and its implementation, and trust that the evaluation results can be extrapolated from the test problem(s) to the real-world problem(s). This approach also depends on the function g: An algorithm that is accurate for one choice of g may be inaccurate for another. Developing more rigorous methods for estimating the non-asymptotic approximation error of probabilistic inference algorithms is an important area of ongoing research [26, 61].

One approach to implementing probabilistic inference is to use the asymptotically exact algorithm templates listed above, but set the computation budget to trade off computational cost with non-asymptotic approximation error. How these algorithm templates are parametrized determines the relationship between computation cost and approximation error. For example, while it is not difficult to construct an asymptotically exact Markov chain Monte Carlo (MCMC) algorithm based on a simple generic MCMC kernel, constructing one that has low approximation error when only run for 100 steps is much more challenging, and usually requires customizing the MCMC kernel to match the problem characteristics. Similarly, in importance sampling, any proposal distribution subject to mild conditions results in an asymptotically exact sampler, but poor choices of proposal distribution require an enormous and impractical number of samples for accurate results, while other choices require only modest numbers of samples [21]. Therefore, customizing Monte Carlo algorithm templates with elements like custom proposal distributions and custom kernels is essential to writing efficient inference algorithms, and practitioners of inference routinely employ these customizations.

Example: Self-normalized importance sampling One standard Monte Carlo algorithm template for approximate inference is *self-normalized importance sampling* [104]. This algorithm template is parametrized by (i) a probabilistic model p, (ii) observed data $\boldsymbol{\rho}$, (iii) a test function g, (iv) an auxiliary probability distribution q that is called the *proposal distribution*, and (v) a number of samples n to produce. Many other Monte Carlo inference algorithms also make use of auxiliary probability distributions. We now briefly describe how self-normalized importance sampling works, using probability distributions and test functions on dictionaries, as introduced in Section 1.2.1. Self-normalized importance sampling produces a weighted collection of samples $\boldsymbol{\sigma}_1, \ldots, \boldsymbol{\sigma}_n$ of the unobserved random choices, with weights $w_1, \ldots, w_n$. Each sample $\boldsymbol{\sigma}_i$ is sampled independently from the proposal distribution ($\boldsymbol{\sigma}_i \sim q$). After all the samples are collected, the weight for each sample is computed using the following formula:

$$w_i := \frac{p(\boldsymbol{\sigma}_i \oplus \boldsymbol{\rho})/q(\boldsymbol{\sigma}_i)}{\sum_{j=1}^{n} p(\boldsymbol{\sigma}_j \oplus \boldsymbol{\rho})/q(\boldsymbol{\sigma}_j)} \tag{1.4}$$

where $\boldsymbol{\sigma} \oplus \boldsymbol{\rho}$ denotes the dictionary formed by *merging* two dictionaries $\boldsymbol{\sigma}$ with $\boldsymbol{\rho}$ with disjoint sets of addresses. The weighted collection can then used to estimate the expected value of a test function using Equation (1.1). The proposal distribution q must satisfy one property: For every unobserved dictionary $\boldsymbol{\sigma}$ that has some nonzero probability under the conditional distribution, there needs to be a nonzero probability that it will be sampled from the proposal. That is,

$$p(\boldsymbol{\sigma}|\boldsymbol{\rho}) > 0 \implies q(\boldsymbol{\sigma}) > 0 \tag{1.5}$$

If this condition holds then the strong law of large numbers can be used to show that self-normalized importance sampling is asymptotically exact as n increases to infinity as in Equation (1.2). The intuition behind this algorithm is that if we could sample each $\boldsymbol{\sigma}_i$ from the conditional distribution $p(\cdot|\boldsymbol{\rho})$, then the samples could all be equally weighted with $w_i = 1/n$ (this is called *simple Monte Carlo*, but is not typically possible because we cannot example exactly from the conditional distribution in practice). The weights in Equation (1.4) correct for the fact that the the proposal distribution is not the same as the conditional distribution, by giving less weight to samples that are over-sampled by the proposal distribution relative to the conditional distribution and giving more weight to samples that are under-sampled by the proposal distribution. Because the probabilities used in the weight calculation are often very small for numerical stability, the weight calculation is typically performed in 'log-space' as follows:

$$m \leftarrow \max_i \{\log p(\boldsymbol{\sigma}_i \oplus \boldsymbol{\rho}) - \log q(\boldsymbol{\sigma}_i)\} \tag{1.6}$$

$$\ell \leftarrow m + \log \left(\sum_{i=1}^{n} \exp \left(\log p(\boldsymbol{\sigma}_i \oplus \boldsymbol{\rho}) - \log q(\boldsymbol{\sigma}_i) - m \right) \right) \tag{1.7}$$

$$w_i \leftarrow \exp \left(\log p(\boldsymbol{\sigma}_i \oplus \boldsymbol{\rho}) - \log q(\boldsymbol{\sigma}_i) - \ell \right) \text{ for } i = 1, \ldots, n \tag{1.8}$$

The next section will illustrate how self-normalized importance sampling can be implemented using the abstract data types from earlier using inference in `burglary_model` as an example, and how the choice of proposal distribution q affects the efficiency of self-normalized importance sampling algorithms.

1.2.6 Implementing inference algorithms with abstract data types

Users of Gen implement inference algorithms in a general-purpose host language using the abstract data types (ADTs) that were introduced in Section 1.2.3. In the Gen implementation described in this thesis, the host language is Julia. Users can implement inference algorithms that are either exact (in rare cases when this is feasible), asymptotically exact, or asymptotically inexact. Users can implement approximate algorithms that have either high or low non-asymptotic error. Gen does not perform any analysis of the user's Julia code that implements their inference algorithm. However, Gen does assist users in implementing asymptotically exact algorithms in several ways: Gen includes a library of Julia implementations of the asymptotically exact algorithm templates listed above (e.g. self-normalized importance sampling). While the properties of the algorithm depend on

how these templates are instantiated, implementing the templates for users reduces the surface area for bugs in user code. Gen also includes library functions for construct primitive MCMC kernels that have certain properties that are necessary (but not sufficient) for asymptotic exactness of MCMC algorithms, as well as a DSL for constructing composite kernels that are automatically instrumented with dynamic checks that detect common bugs in composite MCMC kernels. Finally, Gen's ADTs automate and encapsulate the low-level computations of probabilities, probability densities, derivatives, and samplers, which significantly reduces the code size and, as a result, the surface area for bugs in user code.

Implementing the self-normalized importance sampling template using Gen We now illustrate Gen's programming model using an example. The Julia code below implements the self-normalized importance sampling template using Gen's ADTs:

```
function importance_sampling(model::Gen.GenerativeFunction, observations,
                             proposal::Gen.GenerativeFunction, n::Int)
    traces = [nothing for i in 1:n]
    log_weights = [NaN for i in 1:n]
    for i in 1:n
        proposal_trace = Gen.simulate(proposal)
        all_choices = merge(observations, Gen.get_choices(proposal_trace))
        (traces[i], _) = Gen.generate(model, all_choices)
        log_weights[i] = Gen.logpdf(traces[i]) - Gen.logpdf(proposal_trace)
    end
    weights = exp.(log_weights .- Gen.logsumexp(log_weights))
    return (traces, weights)
end
```

This implementation is adapted from an implementation in Gen's Julia library, but there is no difference between inference code in Gen's library and code that users write—both make use of the same generative function and trace ADTs introduced in Section 1.2.3. We will now walk through this code, describing how it behaves when run on the generative model `burglary_model` to estimate the probability that there was a burglary ($\boldsymbol{\sigma}[\texttt{burglary}] = \texttt{T}$) given the observed data that a phone call was received ($\boldsymbol{\rho} = \{\texttt{calls} \mapsto \texttt{T}\}$). The true answer to this inference query was computed in Section 1.2.2.

First, consider the arguments to the function. The first argument is an instance of the generative function ADT, and defines the generative model:

```
model::Gen.GenerativeFunction
```

We will pass the the generative function `burglary_model` for this argument. The second argument is a dictionary containing the observed data:

```
observations
```

We will pass the dictionary $\boldsymbol{\rho} := \{\texttt{calls} \mapsto \texttt{T}\}$ for this argument. In the Julia implementation of Gen, we can construct this dictionary using the following code:

```
observations = Gen.choicemap()
observations[:calls] = true
```

The third argument is another instance of the generative function ADT that describes the proposal distribution:

```
proposal::Gen.GenerativeFunction
```

Recall that the proposal distribution is an auxiliary probability distribution (q) that is used by self-normalized importance sampling. Gen uses the *same representation* for generative models and auxiliary probability distributions like proposal distributions. That is, proposal distributions are expressed in the same probabilistic programming languages as generative models. For example, consider the probabilistic program below, which defines a generative function `proposal1` $= (q, f)$:

```
1  @gen function proposal1()
2      burglary ~ bernoulli(0.01)
3      if burglary
4          disabled ~ bernoulli(0.1)
5      else
6          disabled = false
7      end
8      if !disabled
9          alarm ~ bernoulli(if burglary 0.94 else 0.01 end)
10     end
11     return nothing
12 end
```

where q and f are given by:

$\boldsymbol{\sigma}$	$q(\boldsymbol{\sigma})$	$f(\boldsymbol{\sigma})$
$\{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}\}$	$0.99 \cdot 0.99$	`nothing`
$\{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{T}\}$	$0.99 \cdot 0.01$	`nothing`
$\{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}\}$	$0.01 \cdot 0.9 \cdot 0.06$	`nothing`
$\{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{T}\}$	$0.01 \cdot 0.9 \cdot 0.94$	`nothing`
$\{\texttt{burglary} \mapsto \texttt{T}, \texttt{disabled} \mapsto \texttt{T}\}$	$0.01 \cdot 0.1$	`nothing`

By comparing with the distribution p for `burglary_alarm`, we can verify that the distribution q satisfies the requirement for validity of importance sampling in Equation 1.5). The fourth and final argument to the importance sampling function is the number of samples (n). Now, examine the body of `importance_sampling`. Lines 3 and 4 initialize arrays to store traces of `model` and log weights $\log w_i$ for each of the n samples:

```
traces = [nothing for i in 1:n]
log_weights = [NaN for i in 1:n]
```

Next, we loop over each sample and populate these arrays. Line 6 samples a trace of the generative function `proposal` using the generative function ADT operation SIMULATE:

```
proposal_trace = Gen.simulate(proposal, ())
```

This line implements the ADT call `proposal`.SIMULATE(x) where $x = ()$. The empty tuple is passed for x because the proposal takes no arguments. The resulting value `proposal_trace` is an instance of Gen's trace ADT that contains data (`proposal`, $(), \boldsymbol{\sigma}$), where $\boldsymbol{\sigma}$ was sampled

from the proposal distribution ($\boldsymbol{\sigma} \sim q$). Line 7 first retrieves the dictionary $\boldsymbol{\sigma}$ from this trace via the CHOICES ADT operation, which is implemented in Julia with:

```
Gen.get_choices(proposal_choices)
```

Then, Line 7 merges the resulting dictionary with the observations dictionary to produce a dictionary called `all_choices`:

```
all_choices = merge(observations, Gen.get_choices(proposal_trace))
```

This corresponds to the mathematical operation $\boldsymbol{\sigma} \oplus \boldsymbol{\rho}$. Next, Line 8 uses the GENERATE ADT operation on the generative function `model`, passing in the dictionary ($\boldsymbol{\sigma} \oplus \boldsymbol{\rho}$) as $\boldsymbol{\tau}$:

```
(traces[i], _) = Gen.generate(model, all_choices)
```

The next line uses the trace ADT operation LOGPDF separately on the model trace and the proposal trace to compute $\log p(\boldsymbol{\sigma} \oplus \boldsymbol{\rho}) - \log q(\boldsymbol{\sigma})$:

```
log_weights[i] = Gen.logpdf(traces[i]) - Gen.logpdf(proposal_trace)
```

After the `for` loop has completed, the array `traces` contains traces of the model $\mathbf{t}_i = (\texttt{model}, (), \boldsymbol{\tau}_i)$ where $\boldsymbol{\tau}_i = \boldsymbol{\sigma}_i \oplus \boldsymbol{\rho}$ for each $i = 1, \ldots, n$. Next, Line 11 implements Equations (1.6-1.8).

```
weights = exp.(log_weights .- Gen.logsumexp(log_weights))
```

In particular, `Gen.logsumexp` implements Equations (1.7-1.8). Finally, on Line 12, the function returns the array of traces $[\mathbf{t}_1, \ldots, \mathbf{t}_2]$ and the array of weights $[w_1, \ldots, w_n]$:

```
return (traces, weights)
```

Applying self-normalized importance sampling in Gen Having defined the algorithm template for self-normalized importance sampling, we now apply it our inference problem involving `burglary_model` using the following Julia code. First, we implement the test function g, which indicates whether there was a burglary or not, in Julia:

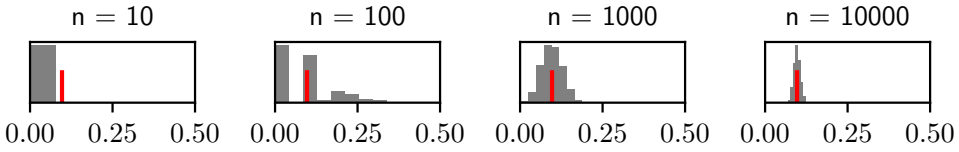
```
g(trace) = trace[:burglary]
```

Then, we run the importance sampling algorithm template, passing `burglary_model` as `model` and `proposal1` as `proposal`. This generates the weighted collection of traces, which we use to estimate of the expected value of the test function:

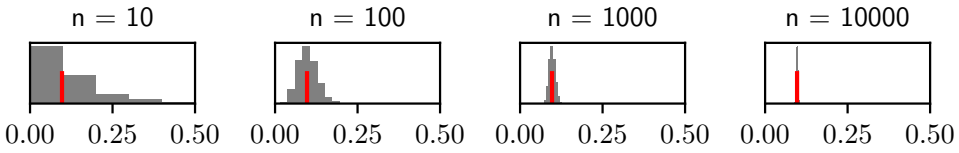
```
observations = Gen.choicemap()
observations[:calls] = true
n = 100
(traces, weights) = importance_sampling(burglary_model, observations, proposal1, n)
estimate = sum([g(trace) for trace in traces] .* weights)
```

This run uses $n = 100$ samples. Using the ground truth value computed earlier, we can evaluate the non-asymptotic approximation error of this algorithm. The algorithm was run $m = 1000$ times for each of 21 different values of n ranging from $n = 10$ to $n = 3000$. Histograms of the estimates for each value of n are shown in Figure 1-5a. They converge on the true value as expected from an asymptotically exact algorithm. The mean

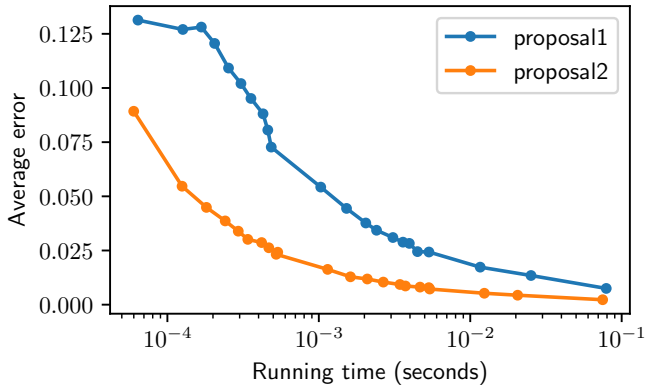
approximation error (Equation 1.3) as a function of running time is shown in the blue curve in Figure 1-5c. The running time for each n is the median across all m replicates.



(a) Histograms of estimates of the conditional probability of burglary via self-normalized importance sampling. The ground truth probability shown in red.



(b) Estimates of the conditional probability using an alternative proposal (`proposal2`).



(c) Average approximation error of self-normalized importance sampling using two different proposal distributions to estimated a conditional probability.

Figure 1-5: Approximation error of self-normalized importance sampling algorithms

Effect of proposal distribution on non-asymptotic approximation error Recall that the choice of proposal distribution q determines how many samples n are needed to achieve a given approximation error. We now illustrate this for our running example, by defining an alternative proposal distribution:

```
@gen function proposal2()
    burglary ~ bernoulli(0.5)
    if burglary
        disabled ~ bernoulli(0.5)
    else
        disabled = false
    end
    if !disabled
        alarm ~ bernoulli(if burglary 0.5 else 0.5 end)
    end
end
```

Figure 1-5b shows estimates from self-normalized importance sampling with this proposal distribution with different numbers of samples, and Figure 1-5c compares the approximation error to that of the algorithm using `proposal1`. Both proposals result in asymptotically exact algorithms but the non-asymptotic error decreases much more rapidly for the algorithm that uses `proposal2` than the algorithm that uses `proposal1` (note the log-scale on the x-axis). The number of samples in self-normalized importance sampling needed to achieve a given level of error depends on how close the proposal distribution is to the conditional distribution of interest [21], which is why practitioners of inference routinely employ custom proposals. The proposal distributions used for the running example are simple, but proposal distributions for real inference problems often use neural networks, heuristics, or even probabilistic inference in other models, with important consequences efficiency of the inference algorithm. The importance of the choice of proposal distributions and other auxiliary probability distributions motivates the use of expressive probabilistic programming languages to specify them. In addition to the self-normalized importance sampling construct introduced here, the thesis presents constructs that allow users to express custom proposal distributions within MCMC and sequential Monte Carlo algorithms, and to train proposal distributions to be more efficient using machine learning.

Chapter 2

Abstract Data Types for Inference: Generative Functions and Traces

This chapter proposes that algorithms for inference in generative models be implemented using an explicit software representation for generative models. We introduce two abstract data types for inference called *generative functions* and *traces*. Generative functions represent models and traces represent the values of latent and observed random variables in models. Representing models explicitly, and defining fixed data types for models and the values of their variables affords advantages that broaden the accessibility of probabilistic inference, improve productivity of users of probabilistic inference, and help to manage the complexity of inference algorithm implementations. First, explicitly writing the model makes modeling assumptions explicit and more easily checked and modified than if the algorithm is implemented by translating the model and algorithm directly into low-level numerical code. Second, the same common low-level computations on models are used for various inference algorithms and for various models, which presents an opportunity for these operations to be implemented once as part of a system and reused, reducing user implementation burden and frequency of bugs. Finally, by abstracting away the generation of data structures, density and gradient computations, incremental computation, and other implementation details of inference algorithms, the data types let users to focus on using their knowledge of the problem to design efficient inference strategies.

Generative functions are an abstract mathematical representation of probabilistic models that is expressive enough to permit models with random structures. Each random variable (or *random choice*) in a model is assigned a unique name (or *address*) by the modeler. To a first approximation, a generative function is a probability distribution on dictionaries that map addresses to their values. Users define generative functions using *modeling languages* like Gen's Dynamic Modeling Language (DML), which allow code in a general-purpose language (e.g. Julia) to be interleaved with sampling statements where random choices are made and labeled with an address. An important feature of generative functions is that they are *composable*—it is possible to construct generative functions from other generative functions using regular function-call syntax in a modeling language. To support such compositions, generative functions have arguments and a return value in addi-

tion to their random choices, and modeling languages allow users to construct a hierarchical address namespace for random choices that mirrors the call tree of generative functions.

Generative functions and traces expose a core set of operations that are sufficient for implementing a broad array of inference algorithms. This chapter introduces operations including simulating traces from a generative function (SIMULATE), generating a trace from arguments and values of random choices (GENERATE), making an incremental update to a trace (UPDATE), evaluating the probability of a trace (LOGPDF), and computing gradients of the probability of a trace with respect to the values of random choices and the arguments to the generative function (GRADIENTS). This chapter gives examples of generative functions expressed mathematically, as well as with corresponding Gen DML code. Chapter 3 shows how the operations provided by generative functions and traces can be used to implement inference algorithms at a high level of abstraction. Chapter 4 extends the generative function and trace data types with additional capabilities, and Chapter 5 discusses how generative functions are compiled from modeling language code.

2.1 An abstract formal representation for generative models

This section describes a mathematical representation for generative models called the *generative function*. The design of the generative function representation is based on several desiderata: First, the representation should be flexible enough to represent models that employ discrete and continuous random variables and structure uncertainty (i.e. the set of random variables sampled is itself random). Therefore, a representation based on joint distributions over a vector of random variables will not suffice. Second, the representation should be sufficient for implementing a wide variety of Monte Carlo and variational inference approaches to inference. Third, the representation should be *minimal*—it should not contain additional features than are needed to implement these algorithms. In Chapter 4, we will extend generative functions with the ability to encapsulate some limited inference logic within themselves. (Incrementally extending generative functions characterizing the added expressiveness introduced at each stage is intended to motivate each aspect of Gen's design, and to illuminate the type of tradeoffs between complexity, expressiveness, and performance that are inherent in probabilistic programming platform design more broadly.) Finally, it should be possible to translate from programs written in *probabilistic modeling languages* into their mathematical representation as generative functions. In this chapter, I show examples of models written in Gen's Dynamic Modeling Language (DML) alongside their mathematical representation as generative functions.

2.1.1 Random choices, addresses, and choice dictionaries

Probabilistic generative models encode a joint probability distribution over a set of random variables. Because we seek to represent generative models where the set of random variables is itself random, care in identifying these random variables is needed. To distinguish between the standard notion of random variable and the notion used in generative functions (and adopting nomenclature from probabilistic programming), we use the phrase *random choice* instead of 'random variable'. Random choices are identified by their *address*. We first

formalize the setting where random choices take values from a finite or countably infinite set (like discrete random variables), and relax this restriction later in the section.

Definition 2.1.1 (Address universe)**.** *Let A be a finite or countably infinite set of addresses. For each $a \in A$, let V_a denote the* domain *of a, where V_a is finite or countably finite. A pair (A, V), which defines a set of addresses and their domains, is called an* address universe*.*

Example: Suppose $A = \{a, b\}$, and $V_a = V_b = \{0, 1\}$. Then address universe (A, V) contains two addresses, and the domain of both addresses is $\{0, 1\}$.

Example: Suppose $A = \mathbb{N}$, $V_1 = \mathbb{N}$ and $V_i = \{\mathtt{T}, \mathtt{F}\}$ for $i \in \{2, 3, \ldots\}$. Then address universe (A, V) contains a countably infinite number of addresses, where address 1 has domain $\mathbb{N}$ and other addresses $(2, 3, \ldots)$ have domain $\{\mathtt{T}, \mathtt{F}\}$.

A probabilistic generative model is traditionally defined as a probability distribution on n-tuples $(x_1, \ldots, x_n) \in (V_1, \ldots, V_n)$ called a *joint probability distribution* for some fixed set of n random variables with domains $V_1, \ldots, V_n$. But joint probability distributions cannot represent models where the set of random variables is itself random. Therefore, we define a probabilistic generative models as a probability distributions on a richer class of objects, namely associative arrays or dictionaries. We call these objects *choice dictionaries*.

Definition 2.1.2 (Choice dictionary)**.** *A* choice dictionary *in address universe (A, V) is a map $\boldsymbol{\sigma} : A_{\boldsymbol{\sigma}} \to \cup_{a \in A} V_a$ where $A_{\boldsymbol{\sigma}}$ is a finite subset of A and where $\boldsymbol{\sigma}(a) \in V_a$ for all $a \in A_{\boldsymbol{\sigma}}$. Equivalently, $\boldsymbol{\sigma}$ is a finite set of pairs $\{(a_1, v_1), \ldots, (a_k, v_k)\}$ such that each $a_i \in A$ appears once and $v_i \in V_{a_i}$ for each i.*

We will use bold lowercase Greek letters (usually $\boldsymbol{\rho}, \boldsymbol{\sigma}, \boldsymbol{\tau}$ and $\boldsymbol{\upsilon}$) to denote choice dictionaries. We will use the syntax $\boldsymbol{\sigma}[a] := \boldsymbol{\sigma}(a)$ to denote the value stored at address a. We will sometimes represent choice dictionaries using the notation $\{a_1 \mapsto v_1, a_2 \mapsto v_2, \cdots\}$. For example, the choice dictionary $\boldsymbol{\sigma}$ with $\boldsymbol{\sigma}[a_1] = 0$ and $\boldsymbol{\sigma}[a_2] = 1$ is denoted $\{a_1 \mapsto 0, a_2 \mapsto 1\}$. Assume that there exists some address universe (A, V). For a set $B \subseteq A$ let $\boldsymbol{\sigma}|_B$ denote the *restriction* of $\boldsymbol{\sigma}$ to the set B; that is, the dictionary obtained by removing entries for all addresses not in B. We say that two choice dictionaries $\boldsymbol{\sigma}$ and $\boldsymbol{\tau}$ *agree* (denoted $\boldsymbol{\sigma} \sim \boldsymbol{\tau}$) if $\boldsymbol{\sigma}[a] = \boldsymbol{\tau}[a]$ for all $a \in A_{\boldsymbol{\sigma}} \cap A_{\boldsymbol{\tau}}$. For two *disjoint* choice dictionaries $\boldsymbol{\sigma}$ and $\boldsymbol{\tau}$ ($|A_{\boldsymbol{\sigma}} \cap A_{\boldsymbol{\tau}}| = 0$), let $\boldsymbol{\sigma} \oplus \boldsymbol{\tau}$ denote the *merge* of $\boldsymbol{\sigma}$ and $\boldsymbol{\tau}$. That is, $\boldsymbol{\rho} = \boldsymbol{\sigma} \oplus \boldsymbol{\tau}$ is a choice dictionary where $A_{\boldsymbol{\rho}} = A_{\boldsymbol{\sigma}} \cup A_{\boldsymbol{\tau}}$ and $\boldsymbol{\rho}[a] = \boldsymbol{\sigma}[a]$ for all $a \in A_{\boldsymbol{\rho}}$ and $\boldsymbol{\rho}[a] = \boldsymbol{\tau}[a]$ for all $a \in A_{\boldsymbol{\tau}}$.

For a finite set $B \subseteq A$ let $\mathcal{T}_B$ denote the set of all choice dictionaries $\boldsymbol{\sigma}$ such that $A_{\boldsymbol{\sigma}} = B$; that is, the set of all choice dictionaries that contain an entry for each address in B and only addresses in B. For a set $B \subseteq A$, let $\mathcal{T}_B^\star$ denote the set of all choice dictionaries that contain entries for some subset of the addresses in B:

$$\mathcal{T}_B^\star := \bigcup_{\substack{C \subseteq B \\ |C| < \infty}} \mathcal{T}_C$$

Example: For $A = \{a, b\}$ and $V_a = V_b = \{0, 1\}$, $\mathcal{T}_A^\star$ contains nine choice dictionaries:

$$\mathcal{T}_A^\star = \left\{ \begin{array}{l} \{\}, \\ \{a \mapsto 0\}, \{a \mapsto 1\}, \{b \mapsto 0\}, \{b \mapsto 1\}, \\ \{a \mapsto 0, b \mapsto 0\}, \{a \mapsto 0, b \mapsto 1\}, \{a \mapsto 1, b \mapsto 0\}, \{a \mapsto 1, b \mapsto 1\} \end{array} \right\}$$

Note that choice dictionaries are always finite, but there may be no upper bound on the number of addresses they may contain. For example, for $A = \mathbb{N}$ and $V_n = \{1\}$ for all $n \in \mathbb{N}$, there is a choice dictionary $\{1 \mapsto 1, 2 \mapsto 1, \ldots, k \mapsto 1\} \in \mathcal{T}_A^\star$ for each $k \in \mathbb{N}$. However, if each address has a countable domain, then the set of all choice dictionaries is countable.

Proposition 2.1.1. *$\mathcal{T}_A^\star$ is countable for all address universes (A, V).*

Proof. Recall that A is countable. Let C be a finite subset of A. Then, $\mathcal{T}_C$ is countable because it is isomorphic to the Cartesian product of a finite set of countable sets (V_a for each $a \in C$). Then, $\mathcal{T}_A^\star$ is countable because it is the countable union of countable sets $\mathcal{T}_C$. □

2.1.2 Probability distributions on choice dictionaries

Given an address universe (A, V) we denote probability distributions on choice dictionaries in $\mathcal{T}_A^\star$ by p, where $p(\boldsymbol{\tau})$ denotes the probability of choice dictionary $\boldsymbol{\tau} \in \mathcal{T}_A^\star$. Formally p is a map $p : \mathcal{T}_A^\star \to [0, 1]$ such that $\sum_{\boldsymbol{\tau} \in \mathcal{T}_A^\star} p(\boldsymbol{\tau}) = 1$. We denote the support of p by $\text{supp}(p) := \{\boldsymbol{\tau} \in \mathcal{T}_A^\star : p(\boldsymbol{\tau}) > 0\}$. We will sometimes denote probability distributions on choice dictionaries using tables, with a row for each dictionary in the support.

Example: Making a single random choice The table below defines a probability distribution p on choice dictionaries for $A = \{a\}$ and $V_1 = \{\texttt{T}, \texttt{F}\}$.

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau})$
$\{a \mapsto \texttt{T}\}$	0.3
$\{a \mapsto \texttt{F}\}$	0.7

Example: Making either one or two random choices For $A = \{a, b\}$ and $V_a = V_b = \{\texttt{T}, \texttt{F}\}$, the following probability distribution p cannot be expressed as a standard joint distribution. With probability 0.3 there is a single random choice with address a and with probability 0.7 there are two random choices, with addresses a and b:

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau})$
$\{a \mapsto \texttt{T}, b \mapsto \texttt{T}\}$	0.42
$\{a \mapsto \texttt{T}, b \mapsto \texttt{F}\}$	0.28
$\{a \mapsto \texttt{F}\}$	0.3

Example: Making an unbounded number of random choices Probability distributions on choice dictionaries can have countably infinite support, due to making an unbounded number of random choices and/or due to individual random choices having a countably infinite domain. For example, consider the distribution p_1 for $A = \mathbb{N}$ and $V_n = \{\texttt{T}, \texttt{F}\}$ for all $n \in \mathbb{N}$, and the distribution p_2 for $A = \{1\}$ and $V_a = \mathbb{N}$:

$\boldsymbol{\tau}$	$p_1(\boldsymbol{\tau})$
$\{1 \mapsto \texttt{F}\}$	0.5
$\{1 \mapsto \texttt{T}, 2 \mapsto \texttt{F}\}$	0.5^2
$\{1 \mapsto \texttt{T}, 2 \mapsto \texttt{T}, 3 \mapsto \texttt{F}\}$	0.5^3
...	...

$\boldsymbol{\tau}$	$p_2(\boldsymbol{\tau})$
$\{1 \mapsto 1\}$	0.5
$\{1 \mapsto 2\}$	0.5^2
$\{1 \mapsto 3\}$	0.5^3
...	...

Each of the example distributions given above satisfies the following property:

Definition 2.1.3 (Structured probability distribution on choice dictionaries)**.** *A probability distribution p on choice dictionaries is called* structured *if for all $\boldsymbol{\tau}, \boldsymbol{\tau}' \in \text{supp}(p)$ either $\boldsymbol{\tau} = \boldsymbol{\tau}'$ or $\boldsymbol{\tau}[a] \neq \boldsymbol{\tau}'[a]$ for some $a \in A_{\boldsymbol{\tau}} \cap A_{\boldsymbol{\tau}'}$. Equivalently, $\boldsymbol{\tau}, \boldsymbol{\tau}' \in \text{supp}(p)$ and $\boldsymbol{\tau} \sim \boldsymbol{\tau}'$ implies that $\boldsymbol{\tau} = \boldsymbol{\tau}'$.*

Example: Probability distribution on choice dictionaries that is not structured Let $A = \{a\}$ and $V_a = \{\texttt{T}, \texttt{F}\}$. The distribution p on choice dictionaries defined below is *not* structured. For $\boldsymbol{\tau}_1 = \{\}$ and $\boldsymbol{\tau}_2 = \{a \mapsto \texttt{T}\}$ we have $A_{\boldsymbol{\tau}_1} = \varnothing \neq \{a\} = A_{\boldsymbol{\tau}_2}$, and $\boldsymbol{\tau}_1$ and $\boldsymbol{\tau}_2$ have no addresses in common to which they assign different values.

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau})$
$\{\}$	0.5
$\{a \mapsto \texttt{T}\}$	0.25
$\{a \mapsto \texttt{F}\}$	0.25

Example: Probability distribution on choice dictionaries that is not structured Let $A = \mathbb{N}$ and $V_i = \{1\}$ for all $i \in \mathbb{N}$. The following is also *not* a structured probability distribution on choice dictionaries, because e.g. for $\boldsymbol{\tau}_1 = \{1 \mapsto 1\}$ and $\boldsymbol{\tau}_2 = \{1 \mapsto 1, 2 \mapsto 1\}$ we have $A_{\boldsymbol{\tau}_1} = \{1\} \neq \{1, 2\} = A_{\boldsymbol{\tau}_2}$ but $\boldsymbol{\tau}_1[1] = \boldsymbol{\tau}_2[1]$:

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau})$
$\{1 \mapsto 1\}$	0.5
$\{1 \mapsto 1, 2 \mapsto 1\}$	0.25
$\{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1\}$	0.125
...	...

The structured property allows differences in *structure* (the set of addresses) between two choice dictionaries in the support of a distribution p to be associated with a difference in the *value* of some shared random choice. We will use this property in Section 1.2.3.

2.1.3 Marginal likelihood, conditioning, and expectation

In standard generative model representations, a joint probability distribution on latent and observed random variables is conditioned on the observed values of the observed random variables to obtain a conditional distribution on the latent random variables. This section defines an analogous notion for conditioning for probability distributions on choice dictionaries. We assume that observed data is represented as a choice dictionary $\boldsymbol{\rho} \in \mathcal{T}_A^\star$. There are two types of events associated with a choice dictionary $\boldsymbol{\rho}$, denoted $E_{\boldsymbol{\rho}}$ and $E'_{\boldsymbol{\rho}}$, that result in two different notions of conditioning on $\boldsymbol{\rho}$:

$$E_{\boldsymbol{\rho}} = \{\boldsymbol{\tau} \in \mathcal{T}_A^\star : \boldsymbol{\tau} \sim \boldsymbol{\rho}\} \quad \text{and} \quad E'_{\boldsymbol{\rho}} = \{\boldsymbol{\tau} \in \mathcal{T}_A^\star : (A_{\boldsymbol{\tau}} \supseteq A_{\boldsymbol{\rho}}) \wedge (\boldsymbol{\tau} \sim \boldsymbol{\rho})\} \tag{2.1}$$

$E_{\boldsymbol{\rho}}$ requires that $\boldsymbol{\tau}$ agrees with $\boldsymbol{\rho}$ on all addresses that it shares with $\boldsymbol{\rho}$ but does not require that $\boldsymbol{\tau}$ contains all addresses in $\boldsymbol{\rho}$. $E'_{\boldsymbol{\rho}}$ does require that $\boldsymbol{\tau}$ contains all addresses in $\boldsymbol{\rho}$.

Example: Two types of events associated with an observed choice dictionary For $\boldsymbol{\rho} = \{2 \mapsto \mathtt{T}\}$, compare $E_{\boldsymbol{\rho}}$ and $E'_{\boldsymbol{\rho}}$ for the following collection of choice dictionaries $\boldsymbol{\tau}$:

$\boldsymbol{\tau}$	$\boldsymbol{\tau} \in E_{\boldsymbol{\rho}}$	$\boldsymbol{\tau} \in E'_{\boldsymbol{\rho}}$
$\{1 \mapsto \mathtt{F}\}$	✓	✗
$\{1 \mapsto \mathtt{T}, 2 \mapsto \mathtt{F}\}$	✗	✗
$\{1 \mapsto \mathtt{T}, 2 \mapsto \mathtt{T}, 3 \mapsto \mathtt{F}\}$	✓	✓

Note that the two events $E_{\boldsymbol{\rho}}$ and $E'_{\boldsymbol{\rho}}$ are equivalent if $\boldsymbol{\rho}$ has the following property.

Definition 2.1.4 (Existentially sound choice dictionary)**.** *A choice dictionary $\boldsymbol{\rho} \in \mathcal{T}_A^\star$ is* existentially sound *under distribution p if $\boldsymbol{\tau} \in \text{supp}(p)$ implies $A_{\boldsymbol{\rho}} \subseteq A_{\boldsymbol{\tau}}$.*

We will proceed with defining conditioning using the first type of event ($E_{\boldsymbol{\rho}}$) because it simplifies the formalism of internal proposals that will be introduced in Chapter 4. Throughout the thesis we assume that dictionaries $\boldsymbol{\rho}$ representing observed data are existentially sound, so there is no distinction between $E_{\boldsymbol{\rho}}$ and $E'_{\boldsymbol{\rho}}$. First we define the *marginal likelihood* of a choice dictionary $\boldsymbol{\sigma}$ as the probability of the event $E_{\boldsymbol{\sigma}}$.

Definition 2.1.5 (Marginal likelihood of a choice dictionary)**.** *For a probability distribution p on choice dictionaries and some $\boldsymbol{\sigma} \in \mathcal{T}_A^\star$ the* marginal likelihood *of $\boldsymbol{\sigma}$ under p is:*

$$\bar{p}(\boldsymbol{\sigma}) := \sum_{\boldsymbol{\tau} \in \mathcal{T}_A^\star} p(\boldsymbol{\tau})[\boldsymbol{\tau} \in E_{\boldsymbol{\sigma}}] = \sum_{\boldsymbol{\tau} \in \mathcal{T}_A^\star} p(\boldsymbol{\tau})[\boldsymbol{\tau} \sim \boldsymbol{\sigma}] = \sum_{B \subseteq A_{\boldsymbol{\sigma}}} \sum_{\boldsymbol{\upsilon} \in \mathcal{T}^\star_{A \setminus A_{\boldsymbol{\sigma}}}} p(\boldsymbol{\upsilon} \oplus (\boldsymbol{\sigma}|_B)) \tag{2.2}$$

where $[\cdot]$ denotes the indicator function.

If the distribution p is structured, then there is only one B with a nonzero term in Equation (2.2). If p is structured and $\boldsymbol{\sigma}$ is existentially sound under p, then the nonzero

term has $B = A_{\boldsymbol{\sigma}}$ and Equation (2.2) simplifies to:

$$\bar{p}(\boldsymbol{\sigma}) = \sum_{\boldsymbol{\upsilon} \in \mathcal{T}^{\star}_{A \setminus A_{\boldsymbol{\upsilon}}}} p(\boldsymbol{\upsilon} \oplus \boldsymbol{\sigma}) \tag{2.3}$$

Definition 2.1.6 (Conditional distribution on choice dictionaries)**.** *For a probability distribution on choice dictionaries p and a choice dictionary $\boldsymbol{\sigma} \in \mathcal{T}^{\star}_{A}$ such that $\bar{p}(\boldsymbol{\sigma}) > 0$, the* conditional distribution *induced by p and $\boldsymbol{\sigma}$ is the probability distribution $p(\cdot|\boldsymbol{\sigma}) : \mathcal{T}^{\star}_{A \setminus A_{\boldsymbol{\sigma}}} \to [0, 1]$ given by:*

$$p(\boldsymbol{\upsilon}|\boldsymbol{\sigma}) := \sum_{B \subseteq A_{\boldsymbol{\sigma}}} \frac{p(\boldsymbol{\upsilon} \oplus (\boldsymbol{\sigma}|_B))}{\bar{p}(\boldsymbol{\sigma})} \tag{2.4}$$

If p is structured then only one term in this sum is nonzero. If p is structured and $\boldsymbol{\sigma}$ is existentially sound, then this term has $B = A_{\boldsymbol{\sigma}}$ and Equation (2.4) simplifies to:

$$p(\boldsymbol{\upsilon}|\boldsymbol{\sigma}) = \frac{p(\boldsymbol{\upsilon} \oplus \boldsymbol{\sigma})}{\bar{p}(\boldsymbol{\sigma})} \tag{2.5}$$

Example: Consider the distribution p given by $p(\{n \mapsto 1, a \mapsto \mathtt{T}\}) := 0.5 \cdot 0.2$ and $p(\{n \mapsto 1, a \mapsto \mathtt{F}\}) := 0.5 \cdot 0.8$ and $p(\{n \mapsto i, a \mapsto \mathtt{T}\}) := 0.5^i \cdot 0.9$ and $p(\{n \mapsto i, a \mapsto \mathtt{F}\}) := 0.5^i \cdot 0.1$ for $i > 1$, and $p(\boldsymbol{\tau}) = 0$ otherwise. Let $\boldsymbol{\sigma}_1 := \{a \mapsto \mathtt{T}\}$ and $\boldsymbol{\sigma}_2 := \{a \mapsto \mathtt{F}\}$. Then $\bar{p}(\boldsymbol{\sigma}_1) = 0.55$ and $\bar{p}(\boldsymbol{\sigma}_2) = 0.45$, and the respective conditional distributions are:

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau})$	$\boldsymbol{\tau} \in E_{\boldsymbol{\sigma}_1}$	$\boldsymbol{\tau} \in E_{\boldsymbol{\sigma}_2}$
$\{n \mapsto 1, a \mapsto \mathtt{T}\}$	$0.5 \cdot 0.2$	✓	✗
$\{n \mapsto 1, a \mapsto \mathtt{F}\}$	$0.5 \cdot 0.8$	✗	✓
$\{n \mapsto 2, a \mapsto \mathtt{T}\}$	$0.5^2 \cdot 0.9$	✓	✗
$\{n \mapsto 2, a \mapsto \mathtt{F}\}$	$0.5^2 \cdot 0.1$	✗	✓
$\{n \mapsto 3, a \mapsto \mathtt{T}\}$	$0.5^3 \cdot 0.9$	✓	✗
$\{n \mapsto 3, a \mapsto \mathtt{F}\}$	$0.5^3 \cdot 0.1$	✗	✓
...	...	...	...

$\boldsymbol{\upsilon}$	$p(\boldsymbol{\upsilon}\|\boldsymbol{\sigma}_1)$	$p(\boldsymbol{\upsilon}\|\boldsymbol{\sigma}_2)$
$\{n \mapsto 1\}$	0.1818	0.8888
$\{n \mapsto 2\}$	0.40909	0.05555
$\{n \mapsto 3\}$	0.204545	0.027777
...	...	...

Example: Consider the distribution p below, and the conditional distribution $p(\cdot|\boldsymbol{\sigma})$ for $\boldsymbol{\sigma} := \{2 \mapsto \mathtt{T}\}$. The marginal likelihood is $\bar{p}(\boldsymbol{\sigma}) = 0.5 + \sum_{i=3}^{\infty} 0.5^i = 0.75$. Note that $\boldsymbol{\sigma}$ is not existentially sound. In particular, the address 2 is not in $\boldsymbol{\tau} = \{1 \mapsto \mathtt{F}\} \in \mathrm{supp}(p)$.

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau})$	$\boldsymbol{\tau} \in E_{\boldsymbol{\sigma}}$
$\{1 \mapsto \mathtt{F}\}$	0.5	✓
$\{1 \mapsto \mathtt{T}, 2 \mapsto \mathtt{F}\}$	0.5^2	✗
$\{1 \mapsto \mathtt{T}, 2 \mapsto \mathtt{T}, 3 \mapsto \mathtt{F}\}$	0.5^3	✓
$\{1 \mapsto \mathtt{T}, 2 \mapsto \mathtt{T}, 3 \mapsto \mathtt{T}, 4 \mapsto \mathtt{F}\}$	0.5^4	✓
...	...	

$\boldsymbol{\upsilon}$	$p(\boldsymbol{\upsilon}\|\boldsymbol{\sigma})$
$\{1 \mapsto \mathtt{F}\}$	$0.6\bar{6}$
$\{1 \mapsto \mathtt{T}\}$	0
$\{1 \mapsto \mathtt{T}, 3 \mapsto \mathtt{F}\}$	$0.1\bar{6}$
$\{1 \mapsto \mathtt{T}, 3 \mapsto \mathtt{T}, 4 \mapsto \mathtt{F}\}$	$0.08\bar{3}$
...	...

Recall that the conditional distribution $p(\cdot|\boldsymbol{\sigma})$ is not defined if $\bar{p}(\boldsymbol{\sigma}) = 0$. The following property guarantees that the conditional distribution is defined for any $\boldsymbol{\sigma} \in \mathcal{T}_A^\star$.

Definition 2.1.7 (Supportive probability distribution on choice dictionaries)**.** *For an address universe (A, V), a probability distribution on choice dictionaries p is called* supportive *if $\bar{p}(\boldsymbol{\sigma}) > 0$ for all $\boldsymbol{\sigma} \in \mathcal{T}_A^\star$.*

Example: A non-supportive probability distribution on choice dictionaries Let $A = \{a\}$. The following p is *not* supportive if $V_a = \{\texttt{T}, \texttt{F}\}$, but it is supportive if $V_a = \{\texttt{T}\}$.

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau})$
$\{a \mapsto \texttt{T}\}$	1

Definition 2.1.8 (Expectation of a function of a choice dictionary)**.** *We denote the expectation of a function $g : \mathcal{T}_A^\star \to \mathbb{R}$ with respect to a probability distribution p on $\mathcal{T}_A^\star$ by:*

$$\mathbb{E}_{\boldsymbol{\tau} \sim p}[g(\boldsymbol{\tau})] := \sum_{\boldsymbol{\tau} \in \mathcal{T}_A^\star} p(\boldsymbol{\tau}) g(\boldsymbol{\tau})$$

when the value of this sum is well-defined.

Note that even when the when the expectation is well-defined, it may be infinite.

2.1.4 Generalizing beyond discrete random choices

The previous section described discrete probability distributions on choice dictionaries. However, it is common to use continuous probability distributions (e.g. normal, beta) to construct generative models, which necessitates domains V_a for random choices that are uncountably infinite (e.g. $\mathbb{R}$ or $[0, 1]$). This section generalizes the notion of a probability distribution on choice dictionaries, replacing the probability mass function p with a probability density function with respect to an appropriate reference measure.

Definition 2.1.9 (Measure-theoretic address universe)**.** *A* measure-theoretic address universe *is a tuple (A, V, M) where A is a finite or countably infinite set of addresses and for each $a \in A$, V_a is a set called the* domain *of a that may be finite, countably infinite, or uncountably infinite, and $M_a = (\Sigma_a, \mu_a)$ where (V_a, Σ_a, μ_a) is a measure space and μ_a is a σ-finite measure.*

The sets of dictionaries $\mathcal{T}_B$ and $\mathcal{T}_B^\star$ for sets $B \subseteq A$ are defined analogously to how they were defined for the discrete address universes above. However, these are no longer necessarily countable sets.

Definition 2.1.10 (Reference measure on choice dictionaries)**.** *Let (A, V, M) be a measure-theoretic address universe. For each finite $B \subseteq A$, let $(\mathcal{T}_B, \Sigma_B)$ denote the product measurable space where $\mathcal{T}_B := \{(B, \mathbf{v}) : \mathbf{v} \in \times_{a \in B} V_a)\}$ and Σ_B is the σ-algebra generated by $\{\{(B, \times_{a \in B} C_a)\} : C_a \in \Sigma_a \text{ for all } a \in B\}$. Let μ_B denote the (unique, σ-finite) measure with $\mu_B(\{(B, \times_{a \in B} C_a)\}) = \prod_{a \in B} \mu_a(C_a)$. Let $\mathcal{T}_A^\star := \cup_{B \subseteq A, |B| < \infty} \mathcal{T}_B$, and let*

$\Sigma^\star_A := \{\cup_{B\subseteq A, |B|<\infty} C_B : C_B \in \Sigma_B \text{ for all } B\}$. *The* reference measure *induced by* (A, V, M) *is the measure* $\mu^\star_A$ *on the measurable space* $(\mathcal{T}^\star_A, \Sigma^\star_A)$*, defined by:*

$$\mu^\star_A(C) := \sum_{\substack{B\subseteq A\\ |B|<\infty}} \mu_B(C_B) \quad \text{for} \quad C = \bigcup_{\substack{B\subseteq A\\ |B|<\infty}} C_B$$

Example: Let $A := \{\mathtt{n}, \mathtt{y}\} \cup \{(\mathtt{x}, 1), (\mathtt{x}, 2), \ldots\}$ where `n`, `y`, and `x` are (single-character) strings[1], and $V_{\mathtt{n}} := \{1, 2, \ldots\}$, and $V_{\mathtt{y}} := \mathbb{R}$, and $V_{(\mathtt{x},i)} := \mathbb{R}$ for $i \in \{1, 2, \ldots\}$. Let $\Sigma_{\mathtt{n}}$ be the set of all subsets of $V_{\mathtt{n}}$ and $\mu_{\mathtt{n}}(C) := |C|$ (the counting measure). For $a \in \{\mathtt{y}\} \cup \{(\mathtt{x}, 1), (\mathtt{x}, 2), \ldots\}$, let Σ_a and μ_a denote the Borel σ-algebra and Lebesgue measure on $\mathbb{R}$, respectively. Let $C_1 := \{\{\mathtt{n} \mapsto 1, (\mathtt{x}, 1) \mapsto x_1\} : x_1 \in [\alpha_{11}, \beta_{11}]\}$, $C_2 := \{\{\mathtt{n} \mapsto 2, (\mathtt{x}, 1) \mapsto x_1, (\mathtt{x}, 2) \mapsto x_2\} : x_1 \in [\alpha_{21}, \beta_{21}], x_2 \in [\alpha_{22}, \beta_{22}]\}$, and $C_3 := \{\{\mathtt{n} \mapsto 3, (\mathtt{x}, 1) \mapsto x_1, (\mathtt{x}, 2) \mapsto x_2\} : x_1 \in [\alpha_{31}, \beta_{31}], x_2 \in [\alpha_{32}, \beta_{32}]\}$. Consider the set $C := C_1 \cup C_2 \cup C_3 \in \Sigma^\star_A$. The reference measure of this set is:

$$\begin{aligned}\mu^\star_A(C) &= \mu_{\{\mathtt{n},(\mathtt{x},1)\}}(C_1) + \mu_{\{\mathtt{n},(\mathtt{x},1),(\mathtt{x},2)\}}(C_2) + \mu_{\{\mathtt{n},(\mathtt{x},1),(\mathtt{x},2)\}}(C_3)\\ &= 1 \cdot (\beta_{11} - \alpha_{11}) + 1 \cdot (\beta_{21} - \alpha_{21}) \cdot (\beta_{22} - \alpha_{22}) + 1 \cdot (\beta_{31} - \alpha_{31}) \cdot (\beta_{32} - \alpha_{32})\end{aligned}$$

For a measure-theoretic address universe (A, V, M), we denote probability densities with respect to the reference measure $\mu^\star_A$ by p. Formally, p is a $\mu^\star_A$-measurable function $p : \mathcal{T}^\star_A \to [0, \infty)$ such that $\int_{\mathcal{T}^\star_A} p(\boldsymbol{\tau})\mu^\star_A(d\boldsymbol{\tau}) = 1$. We denote the set of dictionaries with nonzero density under p by $\text{supp}(p)$.

Example: Let (A, V, M) be the measure-theoretic address universe defined in the previous example. Consider the following density p with respect to $\mu^\star_A$:

$$p(\boldsymbol{\tau}) := \begin{cases} p_{\text{geom}(0.5)}(\boldsymbol{\tau}[\mathtt{n}]) \prod_{i=1}^{\boldsymbol{\tau}[\mathtt{n}]} p_{\text{norm}(0,1)}(\boldsymbol{\tau}[(\mathtt{x}, i)]) & \text{if } A_{\boldsymbol{\tau}} = \{\mathtt{n}, (\mathtt{x}, 1), \ldots, (\mathtt{x}, \boldsymbol{\tau}[\mathtt{n}])\} \\ 0 & \text{otherwise} \end{cases}$$

where $p_{\text{geom}(0.5)}$ is the probability mass function for a geometric distribution with success probability parameter 0.5, and $p_{\text{norm}(0,1)}$ is the probability density function for a normal distribution with mean 0 and standard deviation 1. A natural procedure for sampling a choice dictionary from the measure encoded the density p is to (i) first sample n from the geometric distribution, and then (ii) sample n times from the standard normal distribution.

Definition 2.1.11 (Well-behaved probability density on choice dictionaries)**.** *Note that for every* $B \subseteq A$ *there is a measure-theoretic address universe* $(A \setminus B, V, M)$*, consisting of choice dictionaries* $\boldsymbol{\upsilon} \in \mathcal{T}^\star_{A\setminus B}$*, with reference measure* $\mu^\star_{A\setminus B}$*. A probability density* p *on choice dictionaries is* well-behaved *if for every* $\boldsymbol{\sigma} \in \mathcal{T}^\star_A$*, the function* $\boldsymbol{\upsilon} \mapsto p(\boldsymbol{\upsilon} \oplus \boldsymbol{\sigma})$ *is measurable with respect to* $\mu^\star_{A\setminus A_{\boldsymbol{\sigma}}}$ *and* $\int_{\mathcal{T}^\star_{A\setminus A_{\boldsymbol{\sigma}}}} p(\boldsymbol{\upsilon} \oplus \boldsymbol{\sigma})\mu^\star_{A\setminus A_{\boldsymbol{\sigma}}}(d\boldsymbol{\upsilon}) < \infty$*.*

[1]We allow addresses to take any type of value. We will denote strings using fixed-width font. For example, `foo` is a string. The address $(\mathtt{x}, 2)$ is a tuple of a string and an integer.

We also define *structured* probability densities on choice dictionaries, and *existentially sound* choice dictionaries, using the earlier definitions for the discrete case without modification (Definition 2.1.3 and Definition 2.1.4), but where p is a probability density function instead of a probability mass function. We now define measure-theoretic notions of marginal likelihood and conditioning.

Definition 2.1.12 (Measure-theoretic marginal likelihood of a choice dictionary)**.** *For a well-behaved probability density on choice dictionaries p and a choice dictionary $\boldsymbol{\sigma} \in \mathcal{T}^\star_A$ the* marginal likelihood *of $\boldsymbol{\sigma}$ under p is:*

$$\bar{p}(\boldsymbol{\sigma}) := \sum_{B \subseteq A_{\boldsymbol{\sigma}}} \int_{\mathcal{T}^\star_{A \setminus A_{\boldsymbol{\sigma}}}} p(\boldsymbol{\upsilon} \oplus (\boldsymbol{\sigma}|_B)) \mu^\star_{A \setminus A_{\boldsymbol{\sigma}}}(d\boldsymbol{\upsilon}) \tag{2.6}$$

In the special case when all of the random choices are discrete, this is equivalent to the definition of the marginal likelihood given in the previous section. However, unlike in the case of discrete choices only, the measure-theoretic marginal likelihood is not a probability and may be greater than 1. If $\boldsymbol{\sigma}$ is existentially sound and p is structured, then there is at most one nonzero term in Equation (2.6), with $B = A_{\boldsymbol{\sigma}}$, and the measure-theoretic marginal likelihood of $\boldsymbol{\sigma}$ is:

$$\bar{p}(\boldsymbol{\sigma}) = \int_{\mathcal{T}^\star_{A \setminus A_{\boldsymbol{\sigma}}}} p(\boldsymbol{\upsilon} \oplus \boldsymbol{\sigma}) \mu^\star_{A \setminus A_{\boldsymbol{\sigma}}}(d\boldsymbol{\upsilon}) \tag{2.7}$$

Example: Consider an address universe with $A := \{\mathtt{n}\} \cup_{i=1}^{\infty} \{(\mathtt{x}, i)\}$, where $V_{\mathtt{n}} := \{1, 2, \ldots\}$ and $V_{(\mathtt{x},i)} := \mathbb{R}$ for all $i \in \{1, 2, \ldots\}$. Now consider the density $p(\boldsymbol{\tau})$ that is zero if $\mathtt{n} \notin A_{\boldsymbol{\tau}}$ or if $A_{\boldsymbol{\tau}} \neq \{\mathtt{n}, (\mathtt{x}, 1), \ldots, (\mathtt{x}, \boldsymbol{\tau}[\mathtt{n}])\}$, and is otherwise

$$p_{\text{geom}(0.5)}(\boldsymbol{\tau}[\mathtt{n}]) \cdot \left(\prod_{i=1}^{\boldsymbol{\tau}[\mathtt{n}]} p_{\text{norm}(0,1)}(\boldsymbol{\tau}[(\mathtt{x}, i)]) \right) \cdot p_{\text{norm}(m(\boldsymbol{\tau}),1)}(\boldsymbol{\tau}[\mathtt{y}]) \text{ where } m(\boldsymbol{\tau}) := \sum_{i=1}^{\boldsymbol{\tau}[\mathtt{n}]} \boldsymbol{\tau}[(\mathtt{x}, i)]$$

That is, there is a geometrically-distributed number of random choices $(\mathtt{x}, i)$ that are each independently and identically distributed, and a normally distributed random choice $\mathtt{y}$ whose mean is the sum of these. For $\boldsymbol{\sigma} := \{\mathtt{y} \mapsto 1.5\}$, the marginal likelihood is, where $B_n := \{\mathtt{n}, (\mathtt{x}, 1), \ldots, (\mathtt{x}, n)\}$:

$$\begin{aligned}
\bar{p}(\boldsymbol{\sigma}) &= \sum_{n=1}^{\infty} \int_{\mathcal{T}_{B_n}} p_{\text{geom}(0.5)}(\boldsymbol{\upsilon}[\mathtt{n}]) \prod_{i=1}^{\boldsymbol{\upsilon}[\mathtt{n}]} p_{\text{norm}(0,1)}(\boldsymbol{\upsilon}[(\mathtt{x}, i)]) p_{\text{norm}(m(\boldsymbol{\upsilon}),1)}(\boldsymbol{\sigma}[\mathtt{y}]) \mu_{B_n}(d\boldsymbol{\upsilon}) \\
&= \sum_{n=1}^{\infty} p_{\text{geom}(0.5)}(n) \int_{\mathbb{R}^n} \prod_{i=1}^{n} p_{\text{norm}(0,1)}(x_i) \cdot p_{\text{norm}\left(\sum_{i=1}^n x_i, 1\right)}(\boldsymbol{\sigma}[\mathtt{y}]) d\mathbf{x} \\
&= \sum_{n=1}^{\infty} 0.5^n \frac{1}{\sqrt{2\pi}\sqrt{(n+1)}\exp(0.5\boldsymbol{\sigma}[\mathtt{y}]^2/(n+1))} \approx 0.15573434
\end{aligned}$$

Definition 2.1.13 (Measure-theoretic conditional density)**.** *For a well-behaved probability density on choice dictionaries p and a choice dictionary $\boldsymbol{\sigma} \in \mathcal{T}^\star_A$ where $\bar{p}(\boldsymbol{\sigma}) > 0$, the*

conditional density *induced by p and $\boldsymbol{\sigma}$ is the following probability density on $\boldsymbol{\upsilon}$, with respect to the reference measure $\mu^{\star}_{A \setminus A_{\boldsymbol{\sigma}}}$:*

$$p(\boldsymbol{\upsilon}|\boldsymbol{\sigma}) := \sum_{B \subseteq A_{\boldsymbol{\sigma}}} \frac{p(\boldsymbol{\upsilon} \oplus (\boldsymbol{\sigma}|_B))}{\bar{p}(\boldsymbol{\sigma})} \tag{2.8}$$

When A only contains discrete random choices, this simplifies to Definition 2.1.6. If $\boldsymbol{\sigma}$ is existentially sound under p and p is structured, then the conditional density simplifies:

$$p(\boldsymbol{\upsilon}|\boldsymbol{\sigma}) = \frac{p(\boldsymbol{\upsilon} \oplus \boldsymbol{\sigma})}{\bar{p}(\boldsymbol{\sigma})} \tag{2.9}$$

Example: For the previous example, the conditional density $p(\cdot|\boldsymbol{\sigma})$ where $A_{\boldsymbol{\sigma}} = \{\mathtt{y}\}$ (that is, where only the address $\mathtt{y}$ is observed), is:

$$p(\boldsymbol{\upsilon}|\boldsymbol{\sigma}) = \frac{0.5^{\boldsymbol{\upsilon}[\mathtt{n}]} \cdot \prod_{i=1}^{\boldsymbol{\upsilon}[\mathtt{n}]} p_{\mathrm{norm}(0,1)}(\boldsymbol{\upsilon}[(\mathtt{x}, i)]) \cdot p_{\mathrm{norm}(m(\boldsymbol{\upsilon}),1)}(\boldsymbol{\sigma}[\mathtt{y}])}{\sum_{n=1}^{\infty} 0.5^n (2\pi)^{-1/2} (n+1)^{-1/2} \exp(-0.5\boldsymbol{\sigma}[\mathtt{y}]^2/(n+1))}$$

if $A_{\boldsymbol{\upsilon}} = \{\mathtt{n}, (\mathtt{x}, 1), \ldots, (\mathtt{x}, \boldsymbol{\upsilon}[\mathtt{n}])\}$, and $p(\boldsymbol{\upsilon}|\boldsymbol{\sigma}) = 0$ otherwise.

Proposition 2.1.2. *If p is a structured density on choice dictionaries, then for any $\boldsymbol{\sigma}$ such that $\bar{p}(\boldsymbol{\sigma}) > 0$, the density $p(\cdot|\boldsymbol{\sigma})$ is also structured.*

Proof. Suppose $p(\cdot|\boldsymbol{\sigma})$ is not structured. Then, there exists $\boldsymbol{\upsilon}_1, \boldsymbol{\upsilon}_2 \in \mathcal{T}^{\star}_{A \setminus A_{\boldsymbol{\sigma}}}$ such that $\boldsymbol{\upsilon}_1 \neq \boldsymbol{\upsilon}_2$ and $\boldsymbol{\upsilon}_1 \sim \boldsymbol{\upsilon}_2$ and $p(\boldsymbol{\upsilon}_1|\boldsymbol{\sigma}) > 0$ and $p(\boldsymbol{\upsilon}_2|\boldsymbol{\sigma}) > 0$. This implies that $p(\boldsymbol{\upsilon}_1 \oplus (\boldsymbol{\sigma}|_{B_1})) > 0$ and $p(\boldsymbol{\upsilon}_2 \oplus (\boldsymbol{\sigma}|_{B_2})) > 0$ for some B_1 and B_2. But then $\boldsymbol{\tau}_1 := \boldsymbol{\upsilon}_1 \oplus (\boldsymbol{\sigma}|_{B_1})$ and $\boldsymbol{\tau}_2 := \boldsymbol{\upsilon}_2 \oplus (\boldsymbol{\sigma}|_{B_2})$ satisfy $\boldsymbol{\tau}_1 \neq \boldsymbol{\tau}_2$ and $\boldsymbol{\tau}_1 \sim \boldsymbol{\tau}_2$ and $p(\boldsymbol{\tau}_1) > 0$ and $p(\boldsymbol{\tau}_2) > 0$, which is a contradiction since p is structured.

□

Proposition 2.1.3. *For a structured probability density p on choice dictionaries, and some $\boldsymbol{\sigma} \in \mathcal{T}^{\star}_{A}$, if $\boldsymbol{\tau}_1 = \boldsymbol{\sigma}|_{B_1} \in \mathrm{supp}(p)$ and $\boldsymbol{\tau}_2 = \boldsymbol{\sigma}|_{B_2} \in \mathrm{supp}(p)$ for some B_1, B_2 then $\boldsymbol{\tau}_1 = \boldsymbol{\tau}_2$.*

Proof. Suppose that $\boldsymbol{\tau}_1, \boldsymbol{\tau}_2 \in \mathrm{supp}(p)$ and $\boldsymbol{\tau}_1 = \boldsymbol{\sigma}|_{B_1}$ and $\boldsymbol{\tau}_2 = \boldsymbol{\sigma}|_{B_2}$, where $\boldsymbol{\tau}_1 \neq \boldsymbol{\tau}_2$. Note that $\boldsymbol{\tau}_1 \sim \boldsymbol{\tau}_2$. If $A_{\boldsymbol{\tau}_1} = A_{\boldsymbol{\tau}_2}$ then there must exist a such that $\boldsymbol{\tau}_1[a] \neq \boldsymbol{\tau}_2[a]$, which is a contradiction because $\boldsymbol{\tau}_1 \sim \boldsymbol{\tau}_2$. If $A_{\boldsymbol{\tau}_1} \neq A_{\boldsymbol{\tau}_2}$ then because p is structured, there must exist a such that $\boldsymbol{\tau}_1[a] \neq \boldsymbol{\tau}_2[a]$, which is a contradiction because $\boldsymbol{\tau}_1 \sim \boldsymbol{\tau}_2$. □

Now we adapt a few more definitions to the measure-theoretic setting.

Definition 2.1.14 (Supportive probability density on choice dictionaries)**.** *A well-behaved probability density p on choice dictionaries is* supportive *if $\bar{p}(\boldsymbol{\sigma}) > 0$ for all $\boldsymbol{\sigma} \in \mathcal{T}^{\star}_{A}$.*

Definition 2.1.15 (Expectation with respect to a probability density on choice dictionaries)**.** *For measure-theoretic address universe (A, V, M), we denote the expectation of a*

$\mu_A^\star$-measurable function $g : \mathcal{T}_A^\star \to \mathbb{R}$ with respect to a probability density p on $\mathcal{T}_A^\star$ by:

$$\mathbb{E}_{\boldsymbol{\tau} \sim p}[g(\boldsymbol{\tau})] := \int_{\mathcal{T}_A^\star} g(\boldsymbol{\tau}) p(\boldsymbol{\tau}) \mu_A^\star(d\boldsymbol{\tau})$$

when the value of the integral is well-defined.

2.1.5 Generative functions

Well-behaved and structured probability distributions on choice dictionaries will be the basis for our representation of generative models. However, we want our representation for generative models to be *composable*. That is, given two generative models, it should be possible to construct a third generative model that includes the random variables in both models. For example, we would like to sequence the two generative models, so that the distribution of the second generative model can depend on the values of random choices made by the first. To achieve this, we add a notion of *input* and *output* to our notion of well-behaved probability distribution on choice dictionaries. This gives us our initial definition of a *generative function*:

Definition 2.1.16 (Generative function)**.** *A* generative function *$\mathcal{P}$ in address universe (A, V, M) is a tuple $\mathcal{P} = (X, Y, p, f)$, with components as follows. X is the* argument type*. Y is the* return type*. For each $x \in X$, $p(\cdot; x) : \mathcal{T}_A^\star \to [0, 1]$ is a family of structured (Definition 2.1.3) and well-behaved (Definition 2.1.11) probability densities on $\mathcal{T}_A^\star$. Finally, $f : \{(x, \boldsymbol{\tau}) : \boldsymbol{\tau} \in \mathrm{supp}(p(\cdot; x))\} \to Y$ is the* return value function*.*

Example: A deterministic function A generative function can be built from a regular function by choosing p such that $p(\{\}; x) = 1$ for all x. For example, for $x \mapsto 2x$:

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau}; x)$	$f(x, \boldsymbol{\tau})$
$\{\}$	1	$2x$

Example: The distribution depends on the argument Let $X = (0, 1) \subset \mathbb{R}$ and $Y = [0, 1) \subset \mathbb{R}$. Then (X, Y, p, f) is a generative function where p and f are given by:

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau}; x)$	$f(x, \boldsymbol{\tau})$
$\{a \mapsto \mathtt{T}\}$	x	x
$\{a \mapsto \mathtt{F}\}$	$1 - x$	0

Example: Making different random choices depending on the argument Let $X = \{\mathtt{T}, \mathtt{F}\}$ and $Y = \{0, 1, 2, \ldots, \}$. Then (X, Y, p, f) is a generative function that only

makes random choices for input $x = \texttt{T}$ when p and f are given by:

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau}; \texttt{F})$	$f(\texttt{F}, \boldsymbol{\tau})$
$\{\}$	1	0
$\{\}$	1	0
$\{\}$	1	0
..	..	..

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau}; \texttt{T})$	$f(\texttt{T}, \boldsymbol{\tau})$
$\{n \mapsto 0\}$	0.5	0
$\{n \mapsto 1\}$	0.5^2	0
$\{n \mapsto 2\}$	0.5^3	0
..	..	..

Composing generative functions by sequencing Consider two generative functions $\mathcal{P}_1 = (X_1, Y_1, p_1, f_1)$ and $\mathcal{P}_2 = (X_2, Y_2, p_2, f_2)$. Suppose that the return type of the first generative function matches the argument type of the second generative function ($Y_1 = X_2$). Can we produce a third generative function $\mathcal{P}_3$ from $\mathcal{P}_1$ and $\mathcal{P}_2$ that samples from $\mathcal{P}_1$ and then samples from $\mathcal{P}_2$ given the return value of $\mathcal{P}_1$? This is straightforward if we can have two disjoint sets $A_1, A_2 \subset A$ such that $\mathcal{P}_1$ only ever makes random choices at addresses in A_1 and $\mathcal{P}_2$ only ever makes random choices at addresses in A_2. That is, if for all $x_1 \in X_1$, $\{a : \exists \boldsymbol{\tau} \in \text{supp}(p_1(\cdot; x_1)) \text{ s.t. } a \in A_{\boldsymbol{\tau}}\} \cap A_2 = \varnothing$ and for all $x_2 \in X_2$, $\{a : \exists \boldsymbol{\tau} \in \text{supp}(p_2(\cdot; x_2)) : a \in A_{\boldsymbol{\tau}}\} \cap A_1 = \varnothing$. Then we can construct a third generative function $\mathcal{P}_3 = (X_3, Y_3, p_3, f_3)$ defined by:

$$
\begin{aligned}
X_3 \quad & := X_1 \\
Y_3 \quad & := Y_2 \\
f_3(x, \boldsymbol{\tau}) \quad & := f_2(f_1(x, \boldsymbol{\tau}|_{A_1}), \boldsymbol{\tau}|_{A_2}) \\
p_3(\boldsymbol{\tau}; x) \quad & := p_1(\boldsymbol{\tau}|_{A_1}; x) p_2(\boldsymbol{\tau}|_{A_2}; f_1(x, \boldsymbol{\tau}|_{A_1}))
\end{aligned}
$$

Example: Sequencing generative functions with disjoint addresses Given two generative functions $\mathcal{P}_1$ and $\mathcal{P}_2$ given on the left, we construct $\mathcal{P}_3$ on the right using the construction above with $A_1 = \{n\}$ and $A_2 = \{x\}$:

$\boldsymbol{\tau}$	$p_1(\boldsymbol{\tau})$	$f_1(\boldsymbol{\tau})$
$\{n \mapsto 0\}$	0.5	1
$\{n \mapsto 1\}$	0.5^2	2
$\{n \mapsto 2\}$	0.5^3	3
..	..	..

$\boldsymbol{\tau}$	$p_2(\boldsymbol{\tau}; x)$	$f_2(x, \boldsymbol{\tau})$
$\{x \mapsto \texttt{T}\}$	$x/(x+1)$	0
$\{x \mapsto \texttt{F}\}$	$1/(x+1)$	0

$\boldsymbol{\tau}$	$p_3(\boldsymbol{\tau})$	$f_3(\boldsymbol{\tau})$
$\{n \mapsto 0, x \mapsto \texttt{T}\}$	$0.5 \cdot (1/2)$	0
$\{n \mapsto 0, x \mapsto \texttt{F}\}$	$0.5 \cdot (1/2)$	0
$\{n \mapsto 1, x \mapsto \texttt{T}\}$	$0.5^2 \cdot (2/3)$	0
$\{n \mapsto 1, x \mapsto \texttt{F}\}$	$0.5^2 \cdot (1/3)$	0
$\{n \mapsto 2, x \mapsto \texttt{T}\}$	$0.5^3 \cdot (3/4)$	0
$\{n \mapsto 2, x \mapsto \texttt{F}\}$	$0.5^3 \cdot (1/4)$	0
..	..	..

Example: Generative functions with overlapping addresses However, if generative functions $\mathcal{P}_1$ and $\mathcal{P}_2$ have overlapping addresses (x), then the construction fails:

$\boldsymbol{\tau}$	$p_1(\boldsymbol{\tau})$	$f_1(\boldsymbol{\tau})$
$\{x \mapsto 0\}$	0.5	1
$\{x \mapsto 1\}$	0.5^2	2
..	..	..

$\boldsymbol{\tau}$	$p_2(\boldsymbol{\tau}; x)$	$f_2(x, \boldsymbol{\tau})$
$\{x \mapsto \texttt{T}\}$	$x/(x+1)$	0
$\{x \mapsto \texttt{F}\}$	$1/(x+1)$	0

Address namespaces Like composition of regular functions and procedures in programming, composition of generative functions is most useful when it can be done safely without knowledge of the internals of the two generative functions. One general approach to guaranteeing that two generative functions $\mathcal{P}_1$ and $\mathcal{P}_2$ use disjoint addresses is to wrap them in generative functions $\mathcal{P}_1'$ and $\mathcal{P}_2'$ for which we can guarantee the disjoint address property. Specifically, given two distinct tokens k_1 and k_2, we construct $\mathcal{P}_1'$ by modifying each address a used by $\mathcal{P}_1$ into the tuple address (k_1, a). Let $\textsc{namespace}(\boldsymbol{\tau}, k)$ denote the dictionary $\boldsymbol{\tau}'$ with $\boldsymbol{\tau}'[(k, a)] = \boldsymbol{\tau}[a]$ for each $a \in A_{\boldsymbol{\tau}}$, and no other entries. Then we define $\mathcal{P}_1' := (X_1, Y_1, p_1', f_1')$ and $\mathcal{P}_2' := (X_2, Y_2, p_2', f_2')$ where:

$$
\begin{aligned}
p_1'(\boldsymbol{\tau}; x) &:= p_1(\textsc{namespace}(\boldsymbol{\tau}, k_1); x) \\
p_2'(\boldsymbol{\tau}; x) &:= p_2(\textsc{namespace}(\boldsymbol{\tau}, k_2); x) \\
f_1'(x, \boldsymbol{\tau}) &:= f_1(x, \{a \mapsto \boldsymbol{\tau}[(k_1, a)] : (k_1, a) \in A_{\boldsymbol{\tau}}\}) \\
f_2'(x, \boldsymbol{\tau}) &:= f_2(x, \{a \mapsto \boldsymbol{\tau}[(k_2, a)] : (k_2, a) \in A_{\boldsymbol{\tau}}\})
\end{aligned}
$$

Then, $\mathcal{P}_1'$ and $\mathcal{P}_2'$ can be safely composed because they use addresses in disjoint sets $A_1 = \{(k_1, a) : a \in A\}$ and $A_2 = \{(k_2, a) : a \in A\}$.

Example: Applying address namespaces k_1 and k_2 to the generative functions $\mathcal{P}_1$ and $\mathcal{P}_2$ defined above with overlapping address x allows them to be sequenced:

$\boldsymbol{\tau}$	$p_1'(\boldsymbol{\tau})$	$f_1'(\boldsymbol{\tau})$
$\{(k_1, x) \mapsto 0\}$	0.5	1
$\{(k_1, x) \mapsto 1\}$	0.5^2	2
$\{(k_1, x) \mapsto 2\}$	0.5^3	3
..	..	..

$\boldsymbol{\tau}$	$p_2'(\boldsymbol{\tau}; x)$	$f_2'(x, \boldsymbol{\tau})$
$\{(k_2, x) \mapsto \texttt{T}\}$	$x/(x+1)$	0
$\{(k_2, x) \mapsto \texttt{F}\}$	$1/(x+1)$	0

$\boldsymbol{\tau}$	$p_3(\boldsymbol{\tau})$	$f_3(\boldsymbol{\tau})$
$\{(k_1, x) \mapsto 0, (k_2, x) \mapsto \texttt{T}\}$	$0.5 \cdot (1/2)$	0
$\{(k_1, x) \mapsto 0, (k_2, x) \mapsto \texttt{F}\}$	$0.5 \cdot (1/2)$	0
$\{(k_1, x) \mapsto 1, (k_2, x) \mapsto \texttt{T}\}$	$0.5^2 \cdot (2/3)$	0
$\{(k_1, x) \mapsto 1, (k_2, x) \mapsto \texttt{F}\}$	$0.5^2 \cdot (1/3)$	0
$\{(k_1, x) \mapsto 2, (k_2, x) \mapsto \texttt{T}\}$	$0.5^3 \cdot (3/4)$	0
$\{(k_1, x) \mapsto 2, (k_2, x) \mapsto \texttt{F}\}$	$0.5^3 \cdot (1/4)$	0
..	..	..

Composing generative functions by branching We can also compose two generative functions $\mathcal{P}_1$ and $\mathcal{P}_2$ into a third generative function $\mathcal{P}_3 = (X_3, Y_3, p_3, f_3)$ that, depending on its argument, delegates to either $\mathcal{P}_1$ or $\mathcal{P}_2$. Note that unlike sequencing, this does not require that $\mathcal{P}_1$ and $\mathcal{P}_2$ use disjoint addresses. We define a new generative function $\mathcal{P}_3$ that takes three arguments (b, x_1, x_2) where if $b = 1$ then the distribution and return value will

match those of generative function $\mathcal{P}_1$, and if $b = 2$ then the distribution and return value will match those of $\mathcal{P}_2$:

$$\begin{aligned} X_3 &:= \{1, 2\} \times X_1 \times X_2 \\ Y_3 &:= Y_1 \cup Y_2 \\ p_3(\boldsymbol{\tau}; (b, x_1, x_2)) &:= \begin{cases} p_1(\boldsymbol{\tau}; x_1) & \text{if } b = 1 \\ p_2(\boldsymbol{\tau}; x_2) & \text{if } b = 2 \end{cases} \\ f_3((b, x_1, x_2), \boldsymbol{\tau}) &:= \begin{cases} f_1(x_1, \boldsymbol{\tau}) & \text{if } b = 1 \\ f_2(x_2, \boldsymbol{\tau}) & \text{if } b = 2 \end{cases} \end{aligned}$$

These two types of composition (sequencing and conditional delegation) are just two of the myriad ways that generative functions can be composed into more complex generative functions. Indeed, generative functions can be composed in all the ways that regular functions are routinely composed in functional programming languages. This is one motivation for using *probabilistic modeling languages*, like the one introduced in the next section.

2.2 Languages for defining generative functions

The previous section introduced a formal mathematical representation for generative models called generative functions. However, the probability distributions were described using either tables of choice dictionaries and associated probabilities, or expressions for the density. Often much more compact and intuitive representations of these probability distributions are available. We call these more compact representations *probabilistic modeling languages*, or 'modeling languages' for short. Modeling languages allow generative functions to be specified compactly in a way that makes the inherent structure in the generative function explicit to the modeler. Modeling languages also allow generative functions to be easily composed into more complex generative functions. This section describes one modeling language that is provided in the Julia implementation of Gen, called the Dynamic Modeling Language (DML), which is embedded in the Julia programming language. Other modeling languages will be introduced in Chapter 5, which shows how the clear separation between the generic mathematical representation of generative functions and how they are defined in specific modeling languages affords Gen extensibility that is important for performance.

Formally, a program in a modeling language (or any programming language) is a string $\mathtt{p}$. Each Gen modeling language has a *semantic function* that maps programs $\mathtt{p}$ to generative functions $\mathcal{P}$. Following convention from programming language theory, we denote application of the semantic function with $[\![.]\!]$ instead of $(.)$, so that $[\![\mathtt{p}]\!] = \mathcal{P}$. The semantic functions for practical modeling languages like DML are very complex, and this chapter does not define a formal semantics for DML. Instead this chapter includes selected instructive DML programs $\mathtt{p}$ alongside their corresponding generative function $\mathcal{P} = [\![\mathtt{p}]\!]$. To make the relationship between modeling language programs and their mathematical meaning as generative functions more concrete, this chapter does define a toy modeling language and its semantic function.

2.2.1 Gen Dynamic Modeling Language

The DML is a language for defining generative functions that is embedded in the Julia programming language. DML generative functions are defined using syntax based on Julia's own function definition syntax. For example, a generative function whose argument x is a pair $x = (x_1, x_2)$ is represented as:

```
@gen function (x1, x2)
    <body>
end
```

A deterministic generative function $\mathcal{P} = (X, Y, p, f)$ is one that makes no random choices (i.e. for which $\text{supp}(p(\cdot; x)) = \{\}$ for all $x \in X$). A DML function with no random choice expressions (described below) represents a regular (deterministic) function. For example, we can encode the deterministic generative function with $f(x) = 2x_1 + x_2$ as:

```
@gen function (x1, x2)
    return 2 * x1 + x2
end
```

p

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau}; x)$	$f(x, \boldsymbol{\tau})$
$\{\}$	1	$2x_1 + x_2$

$\mathcal{P} := [\![\mathtt{p}]\!]$

Random choice expressions In order to represent generative functions with choice dictionary distributions other than $p(\{\}; x) = 1$, DML functions use *labeled random choice expressions*, which are a syntax construct not present in the Julia language. Random choice expressions contain two sub-expressions: an *address expression*, and an expression that specifies a probability distribution, separated by '$\sim$':

```
{<address>} ~ <distribution>
```

Probability distribution expressions in DML take the form of a function application of a distribution family to arguments, which are Julia expressions that specify the parameters of the probability distribution. For example, the probability distribution expression below defines a geometric discrete probability distribution with success probability 0.5:

```
{<address>} ~ geometric(0.5)
```

The address expression a Julia expression that can depend on any variables in scope. We often use Julia *symbols* (interned strings) for addresses, which begin with a colon (e.g. `:a`), although other Julia values including e.g. integers, tuples, and strings, can be used as well:

```
{:a} ~ geometric(0.5)
```

The expression evaluates to the sampled value of the random choice. For example, the expression above evaluates to an integer that is greater than or equal to zero. We can then use this expression in other expressions. For example, the expression below evaluates to an integer that is greater than or equal to one:

```
({:a} ~ geometric(0.5)) + 1
```

Often, we want to assign a random choice directly to a variable in the program. There is a syntactic sugar for this case that automatically generates the address based on the variable name:

`a ~ geometric(0.5)` is equivalent to `a = ({:a} ~ geometric(0.5))`

We build the probability distribution p of the generative function by making several random choices, where the distribution of the later choices depends on the random value obtained for the earlier choices, e.g.:

```
a ~ geometric(0.5)
{:b} ~ bernoulli(a/(a+1))
```

Defining a generative function The definition of a generative function in DML includes the `@gen` macro, the arguments to the function, and the body of the function. Consider the following DML code $\mathfrak{p}$ on the left, and the generative function $\mathcal{P} := [\![\mathfrak{p}]\!]$ that it defines, on the right:

```
@gen function (x)
  a ~ geometric(x)
  prob = (a+1)/(a+2)
  return ({:b} ~ bernoulli(prob))
end
```

$\mathfrak{p}$

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau}; x)$	$f(x, \boldsymbol{\tau})$
$\{\mathtt{a} \mapsto 0, \mathtt{b} \mapsto \mathtt{T}\}$	$(1-x)^0 x \cdot \frac{1}{2}$	T
$\{\mathtt{a} \mapsto 0, \mathtt{b} \mapsto \mathtt{F}\}$	$(1-x)^0 x \cdot \frac{1}{2}$	F
$\{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto \mathtt{T}\}$	$(1-x)^1 x \cdot \frac{2}{3}$	T
$\{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto \mathtt{F}\}$	$(1-x)^1 x \cdot \frac{1}{3}$	F
..	..	..

$\mathcal{P} := [\![\mathfrak{p}]\!]$

Note that Julia symbol addresses of the form `:a` are denoted mathematically as `a`. The code above defines an anonymous generative function. We also can assign generative function to a variable, as shown below:

```
@gen function foo(x)
  a ~ geometric(x)
  return ({:b} ~ bernoulli(a/(a+1)))
end
```

Now, `foo` $:= \mathcal{P}$ where $\mathcal{P}$ is defined in the table above.

The density on choice dictionaries Informally, a choice dictionary $\boldsymbol{\tau}$ has $p(\boldsymbol{\tau}; x) > 0$ if there exists a returning execution of the DML function in which each random choice expression with address a is replaced with the value $\boldsymbol{\tau}[a]$ from the choice dictionary, and such that for every random choice expression encountered in the execution with address a, the probability that the expression evaluates to the value $\boldsymbol{\tau}[a]$ is nonzero. For such a choice dictionary, the probability $p(\boldsymbol{\tau}; x)$ is the product of the probabilities of each of the random choices encountered in the execution.

Control flow The language allows many of Julia's control flow constructs to be used in the body of DML functions. This includes including branches and loops. These constructs can be combined with random choice expressions:

```
if ({:a} ~ bernoulli(0.7))
    {:b} ~ bernoulli(0.6)
end
```

```
i = 1
while ({i} ~ bernoulli(0.5))
    i = i + 1
end
```

Invoking other generative functions DML generative functions can *invoke* other DML generative functions (as well as generative functions constructed via other means, as we will see later). Recall that in Section 2.1.5 we found that we could safely compose generative functions using sequencing by wrapping the generative functions in address namespaces, so that all addresses had the form (k, a) for some token k. DML functions can invoke other generative functions using this construction, using a syntax is similar to the random choice expression syntax. In place of the address expression, we place an *address namespace expression* that encodes the token k. Below is an example of a recursive DML function that uses this construct to make two calls, one with token $k_1 = \texttt{L}$ and the other with token $k_2 = \texttt{R}$. The generative function that it represents is shown on the right.

```
@gen function g()
  if ({:go} ~ bernoulli(0.2))
    n1 = ({:L} ~ g())
    n2 = ({:R} ~ g())
    return n1 + n2 + 1
  else
    return 1
  end
end
```

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau}; x)$	$f(x, \boldsymbol{\tau})$
$\{\texttt{go} \mapsto \texttt{F}\}$	0.8	1
$\left\{ \begin{array}{c} \texttt{go} \mapsto \texttt{T}, \\ (\texttt{L}, \texttt{go}) \mapsto \texttt{F}, (\texttt{R}, \texttt{go}) \mapsto \texttt{F} \end{array} \right\}$	$0.2 \cdot 0.8^2$	3
$\left\{ \begin{array}{c} \texttt{go} \mapsto \texttt{T}, \\ (\texttt{L}, \texttt{go}) \mapsto \texttt{T}, (\texttt{R}, \texttt{go}) \mapsto \texttt{F}, \\ (\texttt{L}, (\texttt{L}, \texttt{go})) \mapsto \texttt{F}, \\ (\texttt{L}, (\texttt{R}, \texttt{go})) \mapsto \texttt{F} \end{array} \right\}$	$0.2^2 \cdot 0.8^3$	5
..	..	..

It is also possible to invoke other DML without introducing an address namespace, although this is discouraged. This is done using the same syntax, with the token $*$ for the namespace expression. Below is an example DML function that has the same probability distribution on return values as the recursive DML function above, but uses different addresses:

```
@gen function g(i)
  if ({(i, :go)} ∼ bernoulli(0.2))
    n1 = ({*} ∼ g(i*2))
    n2 = ({*} ∼ g(i*2+1))
    return n1 + n2 + 1
  else
    return 1
  end
end
```

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau}; x)$	$f(x, \boldsymbol{\tau})$
$\{(1, \texttt{go}) \mapsto \texttt{F}\}$	0.8	1
$\left\{ \begin{array}{l} (1, \texttt{go}) \mapsto \texttt{T}, \\ (2, \texttt{go}) \mapsto \texttt{F}, \\ (3, \texttt{go}) \mapsto \texttt{F} \end{array} \right\}$	$0.2 \cdot 0.8^2$	3
$\left\{ \begin{array}{l} (1, \texttt{go}) \mapsto \texttt{T}, \\ (2, \texttt{go}) \mapsto \texttt{T}, \\ (3, \texttt{go}) \mapsto \texttt{F}, \\ (4, \texttt{go}) \mapsto \texttt{F}, \\ (5, \texttt{go}) \mapsto \texttt{F} \end{array} \right\}$	$0.2^2 \cdot 0.8^3$	5
..	..	..

Restrictions There are several properties of DML code that must be satisfied in order for the code to represent a generative function.

1. *Halts with probability 1.* For all values of its arguments, the function must halt with probability 1. For example, infinite while loops and infinite recursion are not permitted. Note that loops of unbounded length and recursion of unbounded depth are permitted (the recursive DML functions shown above are examples). Note that this property is not decidable in general. However, it can often be easily checked for the types of DML programs that are used in probabilistic modeling.[2]

2. *Addresses must be unique.* The function must never sample a random choice at the same address twice. This is possible to check statically in common cases.

3. *Restricted use of randomness outside of random choice expressions.* Random choices that are not part of a random choice expression (e.g. from calling Julia's `rand()` function directly) are valid only if they do not effect control flow or the support of future random choices. This requirement rules out cases that can cause generative functions to not have structured probability distributions (Definition 2.1.3).

4. *Restricted use of mutation.* The function may not mutate its arguments or any variables in its lexical scope that are not private to the function.

5. *DML functions cannot be passed to Julia higher-order functions.* Note that it is possible to implement higher-order functions using recursion (although in a later chapter we will discuss modeling constructs that play the role of higher-order functions, that are specialized for use with generative functions and provide better performance).

[2] For certain recursive programs like those above, simple linear algebra analyses [41] can be used to determine if the program satisfies this property.

Note that satisfying this list of restrictions does not alone guarantee that DML code does define a valid generative function. For example, the well-behaved property (Definition 2.1.11) does not necessarily hold.

An illustrative generative model The DML code below defines a generative model that uses an random number of random choices. We will use this model in the next chapter to illustrate inference procedures.

```
@gen function poly_model(x_coordinates)
    degree ~ uniform_discrete(0, 4)
    var ~ inv_gamma(1, 1)
    coefficients = [({(:c, i)} ~ normal(0, 1)) for i in 0:degree]
    for i=1:length(x_coordinates)
        x = x_coordinates[i]
        mu = coefficients' * x.^(0:degree)
        {(:y, i)} ~ normal(mu, sqrt(var))
    end
end
```

This generative function encodes a generative model of y-coordinates given a fixed vector of x-coordinates. The model assumes that there is a polynomial of unknown degree that maps x-coordinates to their corresponding y-coordinates, and that normally-distributed noise is added to the y-coordinates. The variance of the y-coordinates is unknown and has an inverse-gamma prior. The function first samples a random degree for a polynomial from a geometric distribution, at address `degree`. The degree takes values from $\mathbb{Z}_{\geq 0}$. Then, the function samples a value from an inverse gamma distribution at address `var`, taking values from $\mathbb{R}_{\geq 0}$. Then, the function samples values for each of the coefficients from a normal prior, at addresses of the form $(\mathtt{c}, i)$. Finally, the function samples the y-coordinates by looping over the x-coordinates, computing the value of the polynomial at each point, and adding normally-distributed noise. The y-coordinates are random choices with addresses of the form $(\mathtt{:y}, i)$. The addresses used by this generative function are $A = \{\mathtt{var}, \mathtt{degree}\} \cup \{(\mathtt{c}, i) : i \in \mathbb{Z}_{\geq 0}\} \cup \{(\mathtt{y}, i) : i \in \mathbb{Z}_{>0}\}$. The random choice with address `degree` is discrete with $V_a = \mathbb{Z}_{\geq 0}$ and the other choices are continuous. The address $a = \mathtt{var}$ has $V_a = \mathbb{R}_{>0}$ and $a = (\mathtt{c}, i)$ and $a = (\mathtt{y}, i)$ have $V_a = \mathbb{R}$, and all of these use the Borel σ-algebra and Lebesgue measure. The density with respect to the reference measure $\mu_A^\star$ is:

$$p(\boldsymbol{\tau}; x) = \begin{pmatrix} p_{\text{geom}(0.5)}(\boldsymbol{\tau}[\mathtt{degree}]) \cdot \\ p_{\text{invgamma}(1,1)}(\boldsymbol{\tau}[\mathtt{var}])) \cdot \\ \prod_{j=1}^{\boldsymbol{\tau}[\mathtt{degree}]} p_{\text{norm}(0,1)}(\boldsymbol{\tau}[(\mathtt{c}, j)]) \cdot \\ \prod_{i=1}^{n} p_{\text{norm}(m(\boldsymbol{\tau},i,x),1)}(\boldsymbol{\tau}[(\mathtt{y}, i)]) \end{pmatrix} \text{ where } m(\boldsymbol{\tau}, i, x) := \sum_{j=0}^{\boldsymbol{\tau}[\mathtt{degree}]} x_i^j \cdot \boldsymbol{\tau}[(\mathtt{c}, j)]$$

2.2.2 Formal semantics of a toy modeling language

Figure 2-1 defines a toy modeling language, and defines a semantic function that maps programs $\mathfrak{p}$ in this language to generative functions $\mathcal{P} = (X, Y, p, f)$. The language is a heavily restricted version of DML that does not permit mutation in the body of the

function, does not have loops, cannot call other generative functions, and only samples random choices from discrete distributions. Also, the addresses used in random choice expressions in programs must be *literal Julia symbols*, instead of arbitrary dynamic Julia values as in Gen's DML. The language includes 'if-else' expressions and 'let' expressions.

$$
\begin{array}{rl}
\textit{Constants} & c \in \mathbb{R} \\
\textit{Variables} & x \\
\textit{Address literals} & a \\
\textit{Primitive distributions} & d ::= \texttt{bernoulli} \mid \texttt{geometric}, .. \\
\textit{Primitive real-valued fns.} & f ::= .. \\
\textit{Primitive Boolean-valued fns.} & g ::= .. \\
\textit{Expressions} & E ::= \; c \mid x \mid \\
 & \qquad \{a\} \sim d(E) \mid \\
 & \qquad \text{let } x = E_1 \text{ in } E_2 \text{ end} \mid \\
 & \qquad \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \\
\textit{Function definition} & F ::= \text{@gen function}(x_1, x_2, ..) \; E \text{ end}
\end{array}
$$

(a) Syntax

$$
\begin{array}{rl}
\text{Addrs}[\![c]\!] & = \varnothing \\
\text{Addrs}[\![x]\!] & = \varnothing \\
\text{Addrs}[\![\{a\} \sim d(E)]\!] & = \{a\} \cup \text{Addrs}[\![E]\!] \;\; (\text{where } a \notin \text{Addrs}[\![E]\!]) \\
\text{Addrs}[\![\text{ let } x = E_1 \text{ in } E_2 \text{ end }]\!] & = \text{Addrs}[\![E_1]\!] \cup \text{Addrs}[\![E_2]\!] \\
\text{Addrs}[\![\text{ if } E_1 \text{ then } E_2 \text{ else } E_3]\!] & = \text{Addrs}[\![E_1]\!] \cup \text{Addrs}[\![E_2]\!] \cup \text{Addrs}[\![E_3]\!]
\end{array}
$$

$$
\begin{array}{rl}
\text{Val}[\![c]\!](\sigma)(\boldsymbol{\tau}) & = c \\
\text{Val}[\![x]\!](\sigma)(\boldsymbol{\tau}) & = \sigma[x] \\
\text{Val}[\![\{a\} \sim d(E)]\!](\sigma)(\boldsymbol{\tau}) & = \boldsymbol{\tau}[a] \\
\text{Val}[\![\text{ let } x = E_1 \text{ in } E_2 \text{ end }]\!](\sigma)(\boldsymbol{\tau}) & = \text{Val}[\![E_2]\!](\sigma[x \mapsto \text{Val}[\![E_1]\!](\sigma)(\boldsymbol{\tau})])(\boldsymbol{\tau}) \\
\text{Val}[\![\text{ if } E_1 \text{ then } E_2 \text{ else } E_3]\!](\sigma)(\boldsymbol{\tau}) & = \begin{cases} \text{Val}[\![E_2]\!](\sigma)(\boldsymbol{\tau}) & \text{if Val}[\![E_1]\!](\sigma)(\boldsymbol{\tau}) \\ \text{Val}[\![E_3]\!](\sigma)(\boldsymbol{\tau}) & \text{if } \neg\text{Val}[\![E_1]\!](\sigma)(\boldsymbol{\tau}) \end{cases}
\end{array}
$$

$$
\begin{array}{rl}
\text{Dist}[\![c]\!](\sigma)(\boldsymbol{\tau}) & = [\boldsymbol{\tau} = \{\}] \\
\text{Dist}[\![x]\!](\sigma)(\boldsymbol{\tau}) & = [\boldsymbol{\tau} = \{\}] \\
\text{Dist}[\![\{a\} \sim d(E)]\!](\sigma)(t) & = \begin{cases} p_d(\text{Val}_{[\![E]\!](\sigma)(\boldsymbol{\tau})})(v) & \text{if } \boldsymbol{\tau} = \{a \mapsto v\} \;\; \text{for some } v \\ 0 & \text{otherwise} \end{cases} \\
\text{Dist}[\![\text{ let } x = E_1 \text{ in } E_2 \text{ end }]\!](\sigma)(\boldsymbol{\tau}) & = \text{Dist}[\![E_1]\!](\sigma)(\boldsymbol{\tau}|_{\text{Addrs}[\![E_1]\!]}) \\
 & \quad \cdot \text{Dist}[\![E_2]\!](\sigma[x \mapsto \text{Val}[\![E_1]\!](\sigma)(\boldsymbol{\tau})])(\boldsymbol{\tau}|_{\text{Addrs}[\![E_2]\!]}) \\
\text{Dist}[\![\text{ if } E_1 \text{ then } E_2 \text{ else } E_3]\!](\sigma)(\boldsymbol{\tau}) & = \text{Dist}[\![E_1]\!](\sigma)(\boldsymbol{\tau}|_{\text{Addrs}[\![E_1]\!]}) \\
 & \quad \cdot \begin{cases} \text{Dist}[\![E_2]\!](\sigma)(\boldsymbol{\tau}|_{\text{Addrs}[\![E_2]\!]}) & \text{if Val}[\![E_1]\!](\sigma)(\boldsymbol{\tau}) \\ \text{Dist}[\![E_3]\!](\sigma)(\boldsymbol{\tau}|_{\text{Addrs}[\![E_3]\!]}) & \text{if } \neg\text{Val}[\![E_1]\!](\sigma)(\boldsymbol{\tau}) \end{cases}
\end{array}
$$

$$
\begin{array}{rl}
[\![\text{@gen function}(X_1, \ldots, X_n) \; E \text{ end}]\!] = (\mathbb{R}^n, \mathbb{R}, & \lambda x, \boldsymbol{\tau}.\text{Dist}[\![E]\!]([X_1 \mapsto x_1, \ldots, X_n \mapsto x_n], \boldsymbol{\tau}), \\
 & \lambda x, \boldsymbol{\tau}.\text{Val}[\![E]\!]([X_1 \mapsto x_1, \ldots, X_n \mapsto x_n])(\boldsymbol{\tau}))
\end{array}
$$

(b) Denotational semantics. Three auxiliary semantic functions ($\text{Addrs}[\![\cdot]\!]$, $\text{Val}[\![\cdot]\!]$ and $\text{Dist}[\![\cdot]\!]$) are used to define the main semantic function $[\![\cdot]\!]$, which maps modeling language source code to a generative function tuple. The $\text{Addrs}[\![\cdot]\!]$ function defines the set of addresses of random choices used in the source code. The $\text{Val}[\![\cdot]\!]$ function defines the value of an expression, evaluated using an environment σ and choice dictionary $\boldsymbol{\tau}$. The $\text{Dist}[\![\cdot]\!]$ function defines the probability distribution on choice dictionaries represented by an expression in environment σ.

Figure 2-1: Syntax and denotational semantics of toy probabilistic modeling language

2.3 Abstract data types for probabilistic inference

The previous section described an abstract mathematical representation for generative probabilistic models, called *generative functions*. This section bridges this abstract mathematical representation of generative models with the implementation of inference algorithms for these models. The key idea is that probabilistic inference algorithms can be broken down into a set of primitive operations whose semantics are based on the semantics of programs written in probabilistic modeling languages. In particular, given a generative function representing a generative probabilistic model, it is possible to implement a large array of algorithms for inference in that model using only a handful of operations whose mathematical meaning is derived from the generative function. In particular, we elevate generative functions into an *abstract data type* [75] (ADT). An ADT is a class of objects that stores data and supports operations that are defined abstractly and independently of what data structures are used internally to store the data or how the operations are implemented. The section also defines a second ADT called a *trace* that stores the latent state and observed data for a generative function.

Together, the generative function and trace ADTs encapsulate and automate the low-level computations in Monte Carlo and variational inference algorithms, including sampling, density evaluation, incremental updates, and gradients. As a result, the inference code that uses these ADTs is high-level, abstract, resembles algorithm pseudocode, and has reduced surface area for bugs. The ADTs also provide an abstraction barrier that separates the design and implementation of inference algorithms (which is done by the user) from the low-level computations (which are within the purview of the modeling language compiler).

This section describes a basic version of the generative function and trace ADTs. Then, Chapter 3 shows show a number of inference techniques can be implemented using these ADTs. Chapter 4 then extends the ADTs with additional features that allow more complex logic to be encapsulated within generative functions and traces. Chapter 5 describes techniques that modeling language compilers can use to implement these ADTs. These ADTs form the core of the Gen system [32, 25].

2.3.1 Generative function and trace ADTs

Recall that a generative function (Definition 2.1.16) is a tuple $\mathcal{P} = (X, Y, p, f)$. The tuple (X, Y, p, f) is the data stored in an instance of the *generative function ADT*, which is also denoted $\mathcal{P}$ ($\mathcal{Q}$ is also used). The data stored in an instance of the *trace ADT* is a tuple $(\mathcal{P}, x, \boldsymbol{\tau})$ where $\mathcal{P} = (X, Y, p, f)$ is a generative function, $x \in X$ are valid arguments to the generative function, and the $\boldsymbol{\tau}$ is a choice dictionary that stores the value of each random choice made during a possible execution of the generative function (that is, $\boldsymbol{\tau} \in \text{supp}(p(\cdot; x))$). Instances of the trace ADT are denoted $\mathbf{t}$ or $\mathbf{s}$, so e.g. $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$. We now describe the operations supported by these ADTs. We will illustrate several of these operations using the following generative function `burglary_model`:

```
@gen function burglary_model()
    burglary ~ bernoulli(0.01)
    if burglary
        disabled ~ bernoulli(0.1)
    else
        disabled = false
    end
    if !disabled
        alarm ~ bernoulli(if burglary 0.94 else 0.01 end)
    else
        alarm = false
    end
    calls ~ bernoulli(if alarm 0.70 else 0.05 end)
    return nothing
end
```

Note that this generative function takes no arguments, so the argument value x is an empty tuple in the examples below.

Simulate operation The first operation supported by the generative function ADT is $\mathcal{P}.\text{SIMULATE}(x)$, which takes arguments $x \in X$ to the generative function, then samples a dictionary of random choices $\boldsymbol{\tau}$ according to the distribution $p(\cdot; x)$, and returns the resulting trace $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$. This operation can be used to simulate paired unobservable and observable data from the prior distribution of a generative model. As will be shown in Chapter 3, it can also be used to sample from a proposal distribution, inference model, or variational approximation that is represented as a generative function.

Example: For $\mathcal{P} =$ `burglary_model` a call to $\mathcal{P}.\text{SIMULATE}(x)$ returns one of the 10 possible traces of $\mathcal{P}$. For example, it returns the trace $\mathbf{t} = (\mathcal{P}, x, \{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{call} \mapsto \texttt{F}\})$ with probability $0.99 \cdot 0.99 \cdot 0.95$, and returns the trace $\mathbf{t} = (\mathcal{P}, x, \{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{call} \mapsto \texttt{T}\})$ with probability $0.99 \cdot 0.99 \cdot 0.05$, and so on.

Generate operation The second generative function ADT operation, $\mathcal{P}.\text{GENERATE}(x, \boldsymbol{\sigma})$, is defined for $x \in X$ and $\boldsymbol{\sigma} \in \mathcal{T}_A^\star$. Like SIMULATE, this operation also returns an execution trace $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$, but instead of sampling the random choices $\boldsymbol{\tau}$ according to p, it constructs $\boldsymbol{\tau}$ deterministically via $\boldsymbol{\tau} := \boldsymbol{\sigma}|_B$ for some B (this $\boldsymbol{\tau}$ is unique by Proposition 2.1.3 since $p(\cdot; x)$ is structured). This operation serves as a deterministic constructor for traces. This operation is used in Chapter 3 to initialize Markov chains and compute importance weights. This operation is extended in Chapter 4 with the ability to take a *partial* dictionary that only contains some of the choices, filling in the rest stochastically.

Example: For $\mathcal{P} =$ `burglary_model`, the call $\mathcal{P}.\text{GENERATE}(x, \{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{call} \mapsto \texttt{F}\})$ returns the trace $\mathbf{t} = (\mathcal{P}, x, \{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{call} \mapsto \texttt{F}\})$. A

call $\mathcal{P}$.GENERATE($x, \{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{call} \mapsto \texttt{F}, \texttt{foo} \mapsto \texttt{F}\}$) that passes an extra random choice `foo` results in the same output as when the entry for `foo` is not provided.

Logpdf operation The first operation supported by the trace ADT is $\mathbf{t}$.LOGPDF() for $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$ and returns the log probability $\log p(\boldsymbol{\tau}; x)$ that the random choices in the trace would have been sampled if $x \in X$ were the arguments to the generative function. Note that since all traces $\mathbf{t}$ have $p(\boldsymbol{\tau}; x) > 0$, the value returned by LOGPDF is never $-\infty$. Chapter 3 uses this operation to compute importance weights and acceptance probabilities in various Monte Carlo inference algorithms. As will be described in Chapter 5, the value is typically precomputed and stored within the implementation of $\mathbf{t}$.

Example: For $\mathcal{P} =$ `burglary_model` and $\mathbf{t} = (\mathcal{P}, x, \{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{call} \mapsto \texttt{F}\})$, we have $\mathbf{t}$.LOGPDF() $= \log(0.99 \cdot 0.99 \cdot 0.95)$.

Choices operation The second operation supported by the trace ADT is $\mathbf{t}$.CHOICES(), which for $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$ returns the dictionary $\boldsymbol{\tau}$. This operation highlights the important difference between the traces and choice dictionaries—choice dictionaries are a simple data type that does not support any operations whose semantics is derived from a particular probabilistic model. Note that in the Gen implementation, syntactic sugar allows for the value at an address in $\boldsymbol{\tau}$ to be read from $\mathbf{t}$ concisely with $\mathbf{t}[a]$ instead of $\mathbf{t}$.CHOICES()$[a]$.

Example: For $\mathbf{t} = (\mathcal{P}, x, \{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{call} \mapsto \texttt{F}\})$ we have $\mathbf{t}$.CHOICES() $= \{\texttt{burglary} \mapsto \texttt{F}, \texttt{alarm} \mapsto \texttt{F}, \texttt{call} \mapsto \texttt{F}\}$.

The third operation supported by the trace ADT allows for a new trace to be constructed from a previous trace by changing the values of some of its random choices and its arguments. We now introduce definitions that form the basis of this operation's semantics.

Definition 2.3.1. *For structured probability density p on choice dictionaries, let $h_{\text{update}} : \text{supp}(p) \times \mathcal{T}_A^\star \to (\text{supp}(p) \times \mathcal{T}_A^\star) \cup \{\bot\}$ be defined by: $h_{\text{update}}(\boldsymbol{\tau}, \boldsymbol{\sigma}) := (\boldsymbol{\tau}', \boldsymbol{\tau}|_C)$ where $\boldsymbol{\tau}' := (\boldsymbol{\tau}|_B) \oplus \boldsymbol{\sigma}$ and $C := A^{\mathsf{C}}_{\boldsymbol{\tau}'} \cup A_{\boldsymbol{\sigma}}$ if there exists B such that $\boldsymbol{\tau}' \in \text{supp}(p)$ (by Proposition 2.1.3 $\boldsymbol{\tau}'$ is unique) and $h_{\text{update}}(\boldsymbol{\tau}, \boldsymbol{\sigma}) := \bot$ if there exists no such B.*

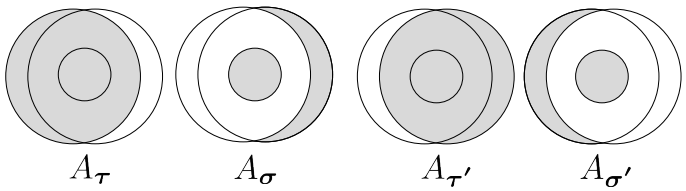
Example For the structured probability density p below, for $\boldsymbol{\tau} = \{a \mapsto \texttt{T}, b \mapsto \texttt{F}\}$ and $\boldsymbol{\sigma} = \{a \mapsto \texttt{F}\}$, we have $h_{\text{update}}(\boldsymbol{\tau}, \boldsymbol{\sigma}) = (\boldsymbol{\tau}', \boldsymbol{\sigma}')$ where $\boldsymbol{\tau}' = (\boldsymbol{\tau}|_B) \oplus \boldsymbol{\sigma} = \{a \mapsto \texttt{F}\} \in \text{supp}(p)$ (with $B = \{\}$) and $\boldsymbol{\sigma}' = \{a \mapsto \texttt{T}, b \mapsto \texttt{F}\}$. For $\boldsymbol{\tau} = \{a \mapsto \texttt{T}, b \mapsto \texttt{F}\}$ and $\boldsymbol{\sigma} = \{b \mapsto \texttt{T}\}$, we have $\boldsymbol{\tau}' = (\boldsymbol{\tau}|_B) \oplus \boldsymbol{\sigma} = \{a \mapsto \texttt{T}, b \mapsto \texttt{T}\} \in \text{supp}(p)$ (with $B = \{a\}$) and $\boldsymbol{\sigma}' = \{b \mapsto \texttt{F}\}$.

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau})$
$\{a \mapsto \texttt{T}, b \mapsto \texttt{T}\}$	0.42
$\{a \mapsto \texttt{T}, b \mapsto \texttt{F}\}$	0.28
$\{a \mapsto \texttt{F}\}$	0.3

Proposition 2.3.1. *If, for some pair of structured probability densities p_1 and p_2 on choice dictionaries, $(\boldsymbol{\tau}', \boldsymbol{\sigma}') = h^{(1)}_{\text{update}}(\boldsymbol{\tau}, \boldsymbol{\sigma})$ then $(\boldsymbol{\tau}, \boldsymbol{\sigma}) = h^{(2)}_{\text{update}}(\boldsymbol{\tau}', \boldsymbol{\sigma}')$, where $h^{(1)}_{\text{update}}$ and $h^{(2)}_{\text{update}}$ are defined with respect to p_1 and p_2 respectively.*

Proof. To show that $((\boldsymbol{\tau}' \oplus \boldsymbol{\sigma})|_{B_2}) \oplus \boldsymbol{\sigma}' = \boldsymbol{\tau}$ for some B_2, use $B_2 := (A_{\boldsymbol{\tau}'} \cap A_{\boldsymbol{\tau}}) \setminus A_{\boldsymbol{\sigma}}$. Next, $A_{\boldsymbol{\tau}'} \cap A_{\boldsymbol{\sigma}'} = A_{\boldsymbol{\tau}'} \cap (A_{\boldsymbol{\tau}} \cap (A^{\mathsf{c}}_{\boldsymbol{\tau}'} \cup A_{\boldsymbol{\sigma}})) = A_{\boldsymbol{\tau}} \cap (A_{\boldsymbol{\tau}'} \cap (A^{\mathsf{c}}_{\boldsymbol{\tau}'} \cup A_{\boldsymbol{\sigma}})) = A_{\boldsymbol{\tau}} \cap (A_{\boldsymbol{\tau}'} \cap A_{\boldsymbol{\sigma}}) = A_{\boldsymbol{\tau}} \cap A_{\boldsymbol{\sigma}}$. Then, $\boldsymbol{\tau}'|_{A^{\mathsf{c}}_{\boldsymbol{\tau}} \cup A_{\boldsymbol{\sigma}'}} = \boldsymbol{\tau}'|_{(A_{\boldsymbol{\tau}'} \setminus A_{\boldsymbol{\tau}}) \cup (A_{\boldsymbol{\tau}'} \cap A_{\boldsymbol{\sigma}'})} = \boldsymbol{\tau}'|_{(A_{\boldsymbol{\sigma}} \setminus A_{\boldsymbol{\tau}}) \cup (A_{\boldsymbol{\tau}} \cup A_{\boldsymbol{\sigma}})} = \boldsymbol{\tau}'|_{A_{\boldsymbol{\sigma}}} = \boldsymbol{\sigma}$. □

The relationships between the sets of addresses involved in $h_{\text{UPDATE}}(\boldsymbol{\tau}, \boldsymbol{\sigma}) = (\boldsymbol{\tau}', \boldsymbol{\sigma}')$ are shown below, where each set is shaded in gray.



The utility of h_{update} is that it lets us specify a new choice dictionary $\boldsymbol{\tau}' \in \text{supp}(p)$ by specifying only an *incremental change* $\boldsymbol{\sigma}$ from the previous choice dictionary $\boldsymbol{\tau}$. Note that $h_{\text{update}}(\boldsymbol{\tau}, \boldsymbol{\tau}') = (\boldsymbol{\tau}', \boldsymbol{\tau})$ for all $\boldsymbol{\tau}' \in \text{supp}(p)$. While we could always specify $\boldsymbol{\sigma} = \boldsymbol{\tau}'$, specifying only the parts of $\boldsymbol{\tau}'$ that are actually different from $\boldsymbol{\tau}$ makes it possible to incrementally compute quantities associated with the change from $\boldsymbol{\tau}$ to $\boldsymbol{\tau}'$.

Example Consider the probability density $p(\boldsymbol{\tau}) := p_{\text{norm}(0,1)}(\boldsymbol{\tau}[1]) \prod_{i=2}^{100} p_{\text{norm}(\boldsymbol{\tau}[i-1],1)}(\boldsymbol{\tau}[i])$ if $A_{\boldsymbol{\tau}} = \{1, \ldots, 100\}$ and 0 otherwise. For $\boldsymbol{\sigma} := \{55 \mapsto \beta\}$ for some $\beta \in \mathbb{R}$, $h_{\text{update}}(\boldsymbol{\tau}, \boldsymbol{\sigma}) = (\boldsymbol{\tau}', \boldsymbol{\sigma}')$ where $\boldsymbol{\tau}' = \{1 \mapsto \boldsymbol{\tau}[1], \ldots, 54 \mapsto \boldsymbol{\tau}[54], 55 \mapsto \beta, 56 \mapsto \boldsymbol{\tau}[56], \ldots, 100 \mapsto \boldsymbol{\tau}[100]\}$, and $\boldsymbol{\sigma}' = \{55 \mapsto \boldsymbol{\tau}[55]\}$. Consider the function $f(\boldsymbol{\tau}) := \sum_{i=1}^{100} \boldsymbol{\tau}[i]$. Note that $f(\boldsymbol{\tau}') = f(\boldsymbol{\tau}) + \boldsymbol{\sigma}[55] - \boldsymbol{\tau}[55]$. More generally, for $(\boldsymbol{\tau}', \boldsymbol{\sigma}') = h_{\text{update}}(\boldsymbol{\tau}, \boldsymbol{\sigma})$, and given a precomputed value for $f(\boldsymbol{\tau})$ it is possible to compute $f(\boldsymbol{\tau}')$ in $2|A_{\boldsymbol{\sigma}}|$ arithmetic operations with $f(\boldsymbol{\tau}') = f(\boldsymbol{\tau}) + \sum_{i \in A_{\boldsymbol{\sigma}}} \boldsymbol{\sigma}[i] - \sum_{i \in A_{\boldsymbol{\sigma}}} \boldsymbol{\tau}[i]$. While we could specify $\boldsymbol{\sigma} = \boldsymbol{\tau}'$, that would result in 200 operations whereas specifying $\boldsymbol{\sigma} = \{55 \mapsto \beta\}$ results in just 2 arithmetic operations. Similarly, consider computing the density ratio $p(\boldsymbol{\tau}')/p(\boldsymbol{\tau})$. For $\boldsymbol{\sigma} = \{i \mapsto \beta\}$ for some $1 < i < 100$, $p(\boldsymbol{\tau}')/p(\boldsymbol{\tau}) = (p_{\text{norm}(\boldsymbol{\tau}[i-1],1)}(\boldsymbol{\sigma}[i]) p_{\text{norm}(\boldsymbol{\sigma}[i],1)}(\boldsymbol{\tau}[i+1]))/(p_{\text{norm}(\boldsymbol{\tau}[i-1],1)}(\boldsymbol{\tau}[i]) p_{\text{norm}(\boldsymbol{\tau}[i],1)}(\boldsymbol{\tau}[i+1]))$, which uses 4 evaluations of the normal distribution density function, whereas computing $p(\boldsymbol{\tau}')$ from scratch alone requires 100 evaluations.

Recall that the densities $p(\cdot; x)$ of generative functions are structured for all $x \in X$. Therefore h_{update} is well-defined for all $p(\cdot; x)$. We will use h_{update} to define an operation that incrementally modifies a trace of a generative function. However, recall that a trace $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$ contains arguments x in addition to the choice dictionary $\boldsymbol{\tau}$. To allow for incremental computation in a setting when both the random choices and the arguments undergo an incremental modification, we introduce a notion of *change hint*.

Definition 2.3.2 (Change hint)**.** *For a set X and a set Δ_X containing at least elements $\top$ and $\bot$, let $\odot_X : X \times \Delta_X \to 2^X$ be a function where 2^X is the power set of X, and with*

values denoted $x \odot_X \delta_X$ *for* $\delta_X \in \Delta_X$*. Let* $(x \odot_X \bot) := X$ *and* $(x \odot_X \top) := \{x\}$ *for all* $x \in X$*. Then each element* $\delta_X \in \Delta_X$ *is called a* change hint*.*

Intuitively, a change hint δ_X may provide information about how a given value $x \in X$ has changed. Specifically, $\delta_X = \bot$ provides no information about the new value, and $\delta_X = \top$ indicates that there was no change, uniquely determining the new value.

Example Consider the set $X = \cup_{i=0}^{\infty}\{(i, \mathbf{x}) : \mathbf{x} \in \mathbb{R}^n\}$ of real-valued vectors. Let $\Delta_X := \{B : |B| < \infty, B \subseteq \{1, 2, \ldots, \}\}$. For each length-$n$ vector $x \in \mathbb{R}^n$, and each $B \in \Delta_X$ such that $B \subseteq \{1, \ldots, n\}$ let $x \odot_X B := \{x' \in \mathbb{R}^m : x'[i] = x[i] \text{ for all } i \in \{1, \ldots, \min(m, n)\} \setminus B\}$. For $x = [4.2, 5.3, 3.1, 1.1]$ and $B = \{2\}$, $x \odot_X B$ includes the vectors $[4.2, -5.3, 3.1, 1.1, 1.0]$ and $[4.2]$ but not the vector $[-4.2, 5.3, 3.1, 1.1]$.

Update operation The third trace ADT operation is $\mathbf{t}.\text{UPDATE}(x', \delta_X, \boldsymbol{\sigma})$. This operation allows the arguments to, and the random choices made by, an execution trace to be modified. Suppose $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$ and $\mathcal{P} = (X, Y, p, f)$. The first argument $(x' \in X)$ provides new arguments, which may be different from the arguments $x \in X$ that are stored in the initial execution trace $\mathbf{t}$. The second argument is a change hint $\delta_X \in \Delta_X$ such that $x' \in (x \odot_X \delta_X)$. The third argument $\boldsymbol{\sigma}$ is a dictionary that must satisfy $h_{\text{update}}(\boldsymbol{\tau}, \boldsymbol{\sigma}) \neq \bot$ for h_{update} defined with respect to $p(\cdot; x)$. The operation computes $\boldsymbol{\tau}'$ and $\boldsymbol{\sigma}'$ using $(\boldsymbol{\tau}', \boldsymbol{\sigma}') := h_{\text{update}}(\boldsymbol{\sigma}, \boldsymbol{\tau})$ and returns $(\mathbf{t}', \boldsymbol{\sigma}', \log w, \delta_Y)$ where $\mathbf{t}' = (\mathcal{P}, x', \boldsymbol{\tau}')$ is a new trace with choices $\boldsymbol{\tau}'$ constructed from $\boldsymbol{\tau}$ and $\boldsymbol{\sigma}$, and the remaining values are as follows: $\log w$ is the log ratio of the new probability to the previous probability $(\log(p(\boldsymbol{\tau}'; x')/p(\boldsymbol{\tau}; x)))$, δ_Y is a change hint for the return value of the generative function, where $f(x', \boldsymbol{\tau}') \in (f(x, \boldsymbol{\tau}) \odot_Y \delta_Y)$. Note that by Proposition 2.3.1, $\boldsymbol{\sigma}'$ is the dictionary, that if passed to the update operation on $\mathbf{t}'$ would reverse the update and result in trace $\mathbf{t}$.

Example: See Figure 1-4 in Section 1.2.3 for an example that involves changing the structure of the choice dictionary for the generative function `burglary_model`.

Example: A generative function that appends to a vector Consider a generative function $\mathcal{P} = (X, Y, p, f)$ where $X, Y := \cup_{i=0}^{\infty}\{(i, \mathbf{x}) : \mathbf{x} \in \mathbb{R}^n\}$ (the set of real-valued vectors) and $p(\{a \mapsto \beta\}; x) := p_{\text{norm}(0,1)}(\beta)$ for all $x \in X$ and $f((i, \mathbf{x}), \boldsymbol{\tau}) := (i, [x_1, \ldots, x_n, \boldsymbol{\tau}[a]])$. This generative function samples a value β from a standard normal distribution and appends it to the input vector to produce the output vector. Consider $\Delta_X = \Delta_Y$ and $\odot_X = \odot_Y$ for the set of real vectors as defined above. Consider $x = [4.2, 5.3, 3.3, 1.1]$ and $x' = [4.2, -5.3, 3.3, 1.1]$ and $\delta_X = \{2\}$ and $\boldsymbol{\sigma} = \{\}$. Then, $\mathbf{t}.\text{UPDATE}(x', \delta_X, \boldsymbol{\sigma})$ returns $(\mathbf{t}', \boldsymbol{\sigma}', 0, \delta_Y)$ where $\mathbf{t}' = (\mathcal{P}, x', \{\})$, and $\boldsymbol{\sigma}' = \{\}$ and where δ_Y satisfies $(i, [x_1, \ldots, x_n, \beta]) \in ((i, [x'_1, \ldots, x'_n, \beta]) \odot_Y \delta_Y)$. Examples of valid δ_Y include $\delta_Y = \{2\}$ and $\delta_Y = \bot$.

Example: Using update with a generative function that has fixed structure Consider the special case when the generative function has fixed structure; that is, for some $B \subseteq A$, $p(\boldsymbol{\tau}; x) > 0$ implies $A_{\boldsymbol{\tau}} = B$ for all $x \in X$. Then, $\boldsymbol{\sigma}$ simply contains new

values for some subset $A_{\boldsymbol{\sigma}} \subseteq B$, and $\boldsymbol{\sigma}'$ contains the previous values ($\boldsymbol{\sigma}' = \boldsymbol{\tau}|_{A_{\boldsymbol{\sigma}}}$).

Many probabilistic inference algorithms make use of gradients of probability densities with respect to parameter values. Consider the class of generative functions $\mathcal{P} = (X, Y, p, f)$ with real-valued arguments $X = \mathbb{R}^n$ where for all x and $\boldsymbol{\tau}$ where the support of the density on choice dictionaries does not change with x ($\text{supp}(p(\cdot; x)) = \text{supp}(p(\cdot; x'))$ for all $x, x' \in X$), and where for all $\boldsymbol{\tau}$ in the support, the function $x \mapsto \log p(\boldsymbol{\tau}; x)$ from $\mathbb{R}^n$ to $\mathbb{R}$ is differentiable. Suppose also that $Y = \mathbb{R}^m$ for some m and for each $\boldsymbol{\tau}$ in the support of the density, the function $x \mapsto f(x, \boldsymbol{\tau})$ from $\mathbb{R}^n$ to $\mathbb{R}^m$ is differentiable. Let the gradient of $\log p(\boldsymbol{\tau}; x)$ with respect to x be denoted $\nabla_x \log p(\boldsymbol{\tau}; x) \in \mathbb{R}^n$ and let the Jacobian of $x \mapsto f(x, \boldsymbol{\tau})$ be denoted $\boldsymbol{J}(x, \boldsymbol{\tau}) \in \mathbb{R}^{n \times m}$. Then consider a function

$$g(x, \boldsymbol{\tau}) := \log p(\boldsymbol{\tau}; x) + \ell(f(x, \boldsymbol{\tau})) \tag{2.10}$$

where $\ell : \mathbb{R}^m \to \mathbb{R}$ is some differentiable function. Then, the gradient of g with respect to x is, by the chain rule:

$$\nabla_x g(x, \boldsymbol{\tau}) = \nabla_x \log p(\boldsymbol{\tau}; x) + (\boldsymbol{J}(x, \boldsymbol{\tau}))\mathbf{v} \tag{2.11}$$

where $\mathbf{v} \in \mathbb{R}^n$ is $\nabla_y \ell(y)$ evaluated at $y = f(x, \boldsymbol{\tau})$. For generative functions satisfying the properties listed above, we can define an operation for the trace ADT that computes $\nabla_x g(x, \boldsymbol{\tau})$ given a trace $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$ and given a vector $\mathbf{v} \in \mathbb{R}^n$.

However, many inference algorithms also make use of gradients of probability densities with respect to the values of *random choices*. Consider generative functions satisfying the above requirements, as well as additional requirements for each $x \in X$: There must exist some set $\mathcal{B}_x \in 2^A$ such that for each $B \in \mathcal{B}_x$, each address $a \in B$ is continuous (that is, $V_a = \mathbb{R}$ and μ_a is the Lebesgue measure). The function from the set $\text{supp}(p(\cdot; x)) \cap \{\boldsymbol{\tau} \in \mathcal{T}_A^\star : B \subseteq A_{\boldsymbol{\tau}}\}$ to $\mathbb{R}$ given by $\boldsymbol{\tau} \mapsto \log p(\boldsymbol{\tau}; x)$ must be differentiable with respect to $\boldsymbol{\tau}[a]$ for each $a \in B$. For each $B \in \mathcal{B}_x$ the function from $\text{supp}(p(\cdot; x)) \cap \{\boldsymbol{\tau} \in \mathcal{T}_A^\star : B \subseteq A_{\boldsymbol{\tau}}\}$ to Y given by $\boldsymbol{\tau} \mapsto f(x, \boldsymbol{\tau})$ must be differentiable with respect to $\boldsymbol{\tau}[a]$ for each $a \in B$. Then, the partial derivative of $g(x, \boldsymbol{\tau})$ with respect to the value of a random choice in $\boldsymbol{\tau}$ for which $a \in B$ is:

$$\frac{\partial g(x, \boldsymbol{\tau})}{\partial \boldsymbol{\tau}[a]} = \frac{\partial \log p(\boldsymbol{\tau}; x)}{\partial \boldsymbol{\tau}[a]} + \sum_{i=1}^{m} \frac{\partial f(x, \boldsymbol{\tau})_i}{\partial \boldsymbol{\tau}[a]} v_i \tag{2.12}$$

where $\mathbf{v} \in \mathbb{R}^n$ is $\nabla_y \ell(y)$ evaluated at $y = f(x, \boldsymbol{\tau})$. Note that this formalism can be easily extended to handle multidimensional continuous random choices with $V_a = \mathbb{R}^k$ for $k > 1$, in which case $\partial g(x, \boldsymbol{\tau}) / \partial \boldsymbol{\tau}[a]$ is replaced with a gradient $\nabla_{\boldsymbol{\tau}[a]} g(x, \boldsymbol{\tau}) \in \mathbb{R}^k$.

Gradients The fourth trace ADT operation is $\mathbf{t}.\text{GRADIENT}(B, \mathbf{v})$ where $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$, $\mathcal{P}$ is a generative function satisfying the requirements above for n and m, $B \in \mathcal{B}_x$ and $\mathbf{v} \in \mathbb{R}^m$. The operation returns a tuple: $(\mathbf{u}, \boldsymbol{\gamma})$ where $\mathbf{u} := \nabla_x \log p(\boldsymbol{\tau}; x) + (\boldsymbol{J}(x, \boldsymbol{\tau}))\mathbf{v} \in \mathbb{R}^n$ and where $\boldsymbol{\gamma}$ is a choice dictionary with $A_{\boldsymbol{\gamma}} = B$ and $\boldsymbol{\gamma}[a] = (\partial \log p(\boldsymbol{\tau}; x) / \partial \boldsymbol{\tau}[a]) + \sum_{i=1}^{m} (\partial f(x, \boldsymbol{\tau})_i / \partial \boldsymbol{\tau}[a]) v_i$ for each $a \in B$. The set B is called the *selection*. This operation computes gradients of the log-density with respect to arguments and the value of selected

random choices. The input $\mathbf{v}$ is included to allow for compositional implementations of this operation that use reverse-mode automatic differentiation.

2.3.2 Implementing the ADT operations compositionally

The ADT operations listed above were designed so that each could be implemented compositionally. This allows an ADT for a generative function that is composed from other generative functions to be implemented using the ADT operations of the two respective constituent generative functions, without needing the source code of the constituent functions. We now demonstrate this compositionality of for one of the trace ADT operations.

Recall the construction from Section 2.1.5 of a generative function $\mathcal{P}_3 = (X_3, Y_3, p_3, f_3)$ from two other generative functions $\mathcal{P}_1 = (X_1, Y_1, p_1, f_1)$ and $\mathcal{P}_2 = (X_2, Y_2, p_2, f_2)$ by sequencing $\mathcal{P}_1$ followed by $\mathcal{P}_2$:

$$\begin{aligned} X_3 &:= X_1 \\ Y_3 &:= Y_2 \\ f_3(x, \boldsymbol{\tau}) &:= f_2(f_1(x, \boldsymbol{\tau}|_{A_1}), \boldsymbol{\tau}|_{A_2}) \\ p_3(\boldsymbol{\tau}; x) &:= p_1(\boldsymbol{\tau}|_{A_1}; x) p_2(\boldsymbol{\tau}|_{A_2}; f_1(x, \boldsymbol{\tau}|_{A_1})) \end{aligned}$$

Consider a trace $\mathbf{t}_3 = (\mathcal{P}_3, x_3, \boldsymbol{\tau}_3)$ and the trace $\mathbf{t}_1 := (\mathcal{P}_1, x_1, \boldsymbol{\tau}_1)$ defined by $x_1 := x_3$ and $\boldsymbol{\tau}_1 := (\boldsymbol{\tau}_3)|_{A_1}$ and the trace $\mathbf{t}_2 := (\mathcal{P}_2, x_2, \boldsymbol{\tau}_2)$ defined by $x_2 := f_1(x_3, (\boldsymbol{\tau}_3)|_{A_1})$ and $\boldsymbol{\tau}_2 := (\boldsymbol{\tau}_3)|_{A_2}$. Consider $x'_3 \in X_3$ and $\delta_{X3} \in \Delta_X$ such that $x'_3 \in (x_3 \odot \delta_{X3})$ and $\boldsymbol{\sigma}_3$ such that $h^{(3)}_{\text{update}}(\boldsymbol{\tau}_3, \boldsymbol{\sigma}_3) \neq \perp$, where $h^{(3)}_{\text{update}}$ is defined with respect to the density $p(\cdot; x'_3)$. Then the behavior of $\mathbf{t}_3.\text{UPDATE}(x'_3, \delta_{X3}, \boldsymbol{\sigma}_3)$ is defined. Consider $\mathbf{t}'_2, \log w_2, \boldsymbol{\sigma}'_2, \delta_{Y2}, \mathbf{t}'_2, \log w_2, \boldsymbol{\sigma}'_2, \delta_{Y2}$ given by:

$$\begin{aligned} (\mathbf{t}'_1, \log w_1, \boldsymbol{\sigma}'_1, \delta_{Y1}) &= \mathbf{t}_1.\text{UPDATE}(x'_3, \delta_{X3}, (\boldsymbol{\sigma}_3)|_{A_1}) \quad \text{where } \mathbf{t}'_1 = (\mathcal{P}_1, x'_1, \boldsymbol{\tau}'_1) \\ (\mathbf{t}'_2, \log w_2, \boldsymbol{\sigma}'_2, \delta_{Y2}) &= \mathbf{t}_2.\text{UPDATE}(f_1(x'_3, (\boldsymbol{\tau}_3)|_{A_1}), \delta_{Y1}, (\boldsymbol{\sigma}_3)|_{A_2}) \quad \text{where } \mathbf{t}'_2 = (\mathcal{P}_2, x'_2, \boldsymbol{\tau}'_2) \end{aligned}$$

Proposition 2.3.2. *Given the conditions above, a valid value of* $\boldsymbol{t}_3.\text{UPDATE}(x'_3, \delta_{X3}, \boldsymbol{\sigma}_3)$ *is* $(\boldsymbol{t}'_3, \log w_3, \boldsymbol{\sigma}'_3, \delta_{Y3})$ *where* $\boldsymbol{t}'_3 = (\mathcal{P}_3, x'_1, \boldsymbol{\tau}'_1 \oplus \boldsymbol{\tau}'_2)$, $\log w_3 = \log w_1 + \log w_2$, $\boldsymbol{\sigma}'_3 = \boldsymbol{\sigma}'_1 \oplus \boldsymbol{\sigma}'_2$, *and* $\delta_{Y3} = \delta_{Y2}$. *Note that the valid value of* $\boldsymbol{t}_3.\text{UPDATE}(x'_3, \delta_{X3}, \boldsymbol{\sigma}_3)$ *would be unique, except for the change hint that is returned, for which there may be multiple valid values (e.g.* $\perp$ *is always a valid change hint to return).*

Proof. We prove the claim $\log w_3 = \log w_1 + \log w_2$. $\log w_3 = \log p_3(\boldsymbol{\tau}'_1 \oplus \boldsymbol{\tau}'_2; x'_3) - \log p_3(\boldsymbol{\tau}_1 \oplus \boldsymbol{\tau}_2; x'_3) = \log p_1(\boldsymbol{\tau}'_1; x'_1) + \log p_2(\boldsymbol{\tau}'_2; x'_2) - \log p_1(\boldsymbol{\tau}_1; x_1) - \log p_2(\boldsymbol{\tau}_2; x_2) = (\log p_1(\boldsymbol{\tau}'_1; x'_1) - \log p_1(\boldsymbol{\tau}_1; x_1)) - (\log p_2(\boldsymbol{\tau}'_2; x'_2) - \log p_2(\boldsymbol{\tau}_2; x_2)) = \log w_1 + \log w_2$. □

Chapter 5 discusses implementing the generative function and trace ADTs in more detail. In particular Chapter 5 discusses how implementations of these ADTs can be generated from modeling language source code.

2.4 Related work

Semantics of probabilistic programming languages. Numerous efforts have been made to define semantics for universal probabilistic programming languages and inference algorithms for these languages [15, 55, 115]. We do not define the denotational semantics for Gen's modeling languages. Instead, we provide a definition of an abstract mathematical object (generative functions) that makes the addresses of random choices explicit, on top of which Gen's trace data type is built. The probability densities and reference measures used in the definition of a generative function are similar to those used in prior work [15], except that we use arbitrary addresses that have individual base measures.

Incremental computation and change hints The UPDATE trace operation is designed to be implemented using compositional incremental computation schemes via change hints. Note that UPDATE requires computation of not just the new return value (as in traditional deterministic computation) but also the ratio of probability densities of the new and old random choices and the new choice dictionary. Incremental computation has a long history [97, 2]. Change hints in Gen are similar to over-approximations of *change structures* [18]. The addresses of the random choices passed in as constraints to UPDATE play a similar role as the memory addresses in self-adjusting computation based on memory locations [2]. Gen does not prescribe how modeling language implementations use or compute change hints; different approaches may be better suited for different modeling languages. Automatically generating code that computes change hints for code in general-purpose programming languages is an interesting area of future work.

Compositional automatic differentiation Numerous systems support compositional automatic differentiation, where users write custom operators and provide gradient computations designed for reverse-mode automatic differentiation based on the chain rule [1, 94]. The GRADIENT operator is Gen is unique because it distinguishes between differentiation with respect to arguments to the computation, and differentiation with respect to the values of random choices in the trace based on a dynamic selection including over addresses that may only exist dynamically, depending on control flow.

Addresses for random choices and generative function calls Unlike the modeling languages of Church [51], Venture [79] and most other probabilistic programming systems, Gen's dynamic modeling language allows users to assign specific addresses to random choices for fine-grained control over traces when implementing inference algorithms. The Pyro system [13] also introduced individual addresses to random choices independently and concurrently, although the entire execution uses the same address namespace in Pyro, whereas Gen allows users to assign an address for a function call. This reflects a difference in architecture—Gen's modeling languages can invoke any generative function that implements Gen's abstract data types; the implementation details of the callee and the random choices that is makes are encapsulated.

Chapter 3

Implementing Inference Using Generative Functions and Traces

A variety of algorithms for probabilistic inference and learning can be implemented using the tools of the previous chapter—generative functions and traces. This chapter gives procedures in pseudocode for implementing inference algorithms and building blocks for inference algorithms using generative functions and traces. Despite the fact that these procedures operate on flexible model representations derived from probabilistic programs, they are short and lay bare the mathematical structure of inference algorithms because the implementation details are abstracted away by the generative function and trace abstract data types. Such details include how the trace data structure is implemented, how probabilities or gradients are computed, and how conditional independence in the model is exploited by the modeling language compiler to make the abstract data type operations more efficient.

Users of Gen implement their own inference algorithms in a general-purpose programming language using a Gen API that provides generative function and trace data types, together with embedded probabilistic modeling languages for defining generative functions. At the time of this writing, the Gen implementation uses Julia as the general-purpose host language, but Gen's architecture and the inference pseudocode given in this chapter could be implemented in other host languages as well. The pseudocode procedures provided in this chapter can be implemented by users in Julia. In some cases these procedures have also been implemented within Gen's inference library, which contains reusable inference logic built on top of the abstract data types. The chapter includes pedagogical examples that are implemented in Julia. The examples use Gen's probabilistic modeling languages to express generative functions, Gen's Julia API, and in some cases Gen's inference library.

The inference procedures defined in this chapter are in many cases composable with one another. Examples of composability include (i) composing MCMC kernels together into more complex MCMC kernels, (ii) composing particle filtering with MCMC kernels (where the MCMC kernels play the role of rejuvenation moves), and (iii) composing learning and inference by training proposal distributions for use in Monte Carlo inference algorithms on data simulated from a generative model. The composability is due in part to the fact that all of the procedures are implemented with the same abstract data types. The ability to

implement an open-ended set of inference algorithms from primitive building blocks in a general-purpose programming language while using probabilistic programming languages to explicitly express probabilistic models is a distinctive feature of Gen.

The chapter also introduces several new programming constructs for Monte Carlo algorithms, including domain-specific languages for composing MCMC kernels and for expressing deterministic transformations of traces. One construct, called a *trace translator*, is a versatile building block for implementing inference algorithms based on two or more probabilistic models of the same domain that use different latent representations.

3.1 Simple Monte Carlo with traces

Monte Carlo inference algorithms [104] are randomized approximation algorithms that are used to estimate properties of probability distributions that are difficult to compute symbolically. The most basic Monte Carlo algorithm, sometimes called *simple Monte Carlo*, involves (i) approximating a probability distribution by a collection of samples drawn from the distribution, and (ii) using this approximation to estimate expectations of test functions.

Consider a probability distribution p on choice dictionaries, and a test function $g : \mathcal{T}_A^\star \to \mathbb{R}$ whose expectation ($\mathbb{E}_{\boldsymbol{\tau}\sim p}[g(\boldsymbol{\tau})]$, assuming it exists) we want to estimate. Simple Monte Carlo approximates the distribution by a collection $(\boldsymbol{\tau}^{(1)}, \ldots \boldsymbol{\tau}^{(n)})$ where each $\boldsymbol{\tau}^{(i)}$ is independently and identically distributed according to p, and averages the value of the test function across the samples in the collection:

$$\mathbb{E}_{\boldsymbol{\tau}\sim p}[g(\boldsymbol{\tau})] \approx \frac{1}{n}\sum_{i=1}^{n} g(\boldsymbol{\tau}^{(i)}) \quad \text{for} \quad \boldsymbol{\tau}^{(i)} \overset{iid}{\sim} p \tag{3.1}$$

In particular, the estimate converges almost surely to the expectation as n increases. When the function takes the form $g(\boldsymbol{\tau}) := [\boldsymbol{\tau} \in E]$ for some event $E \subseteq \mathcal{T}_A^\star$, Equation (3.1) estimates the probability of the event E. Note that simple Monte Carlo requires the ability to sample from the probability distribution, which is often not possible for the conditional distributions arising in probabilistic inference. The subsequent sections in this chapter give more sophisticated approximation algorithms that do support conditional distributions.

Simple Monte Carlo with traces As described in the previous chapter, a generative function $\mathcal{P}$ defines a probability distribution p on choice dictionaries $\boldsymbol{\tau}$. The first step in employing simple Monte Carlo to infer an expectation under a probability distribution p is to construct a generative function $\mathcal{P}$ called the 'model' whose distribution on choice dictionaries is p, typically by writing a program in a probabilistic modeling language. To sample from this distribution we use $\mathcal{P}$.SIMULATE, which returns a trace $\mathbf{t}$ that wraps the sampled choice dictionary $\boldsymbol{\tau} \sim p$ with additional metadata and operations that are useful for implementing more sophisticated inference algorithms. The simple Monte Carlo procedure (Algorithm 1) does not make use of these additional features of traces, and just retrieves the sampled choice dictionary with $\mathbf{t}$.CHOICES(). The extra capabilities of traces are important for the more sophisticated algorithms appearing later in this chapter. In this chapter, the arguments to the generative function $\mathcal{P}$ that represents the model often play no role in the

algorithm and can be treated as constant. To reduce complexity of the notation, in such cases we denote the arguments to $\mathcal{P}$ with an underscore (_). For example, $\mathcal{P}$.SIMULATE(_) invokes SIMULATE on $\mathcal{P}$, which may or may not take arguments.

Algorithm 1 Simple Monte Carlo with traces

procedure SIMPLE-MONTE-CARLO($\mathcal{P}$, g, n)
  for $i \leftarrow 1 \ldots n$ **do**
    $\mathbf{t}^{(i)} \leftarrow \mathcal{P}$.SIMULATE(_)
  end for
  return $\frac{1}{n} \sum_{i=1}^{n} g(\mathbf{t}^{(i)}$.CHOICES())
end procedure

Example: Estimating the probability of an event Consider the generative function $\mathcal{P} :=$ `poly_model` defined below that encodes a generative model of y-coordinates generated from a random polynomial of random degree:

```
@gen function poly_model(x_coordinates)
    degree ~ uniform_discrete(0, 4)
    var ~ inv_gamma(1, 1)
    coefficients = [({(:c, i)} ~ normal(0, 1)) for i in 0:degree]
    for i=1:length(x_coordinates)
        x = x_coordinates[i]
        mu = coefficients' * x.^(0:degree)
        {(:y, i)} ~ normal(mu, sqrt(var))
    end
end
```

Suppose we want to estimate the probability that the second y-coordinate is greater than the first y-coordinate, for some given x-coordinates. Then $g(\boldsymbol{\tau}) := [\boldsymbol{\tau}[(\mathrm{y}, 2)] > \boldsymbol{\tau}[(\mathrm{y}, 1)]$. To estimate this probability using Algorithm 1, we first assemble the collection of ($n = 100$) traces by simulating from the generative function. Julia code to do this, using the version of Gen at time of this writing, is shown below:

```
x_coordinates = [0.0, 1.0, 2.0, 3.0]
n = 100
traces = [Gen.simulate(poly_model, (x_coordinates,)) for i in 1:n]
```

The Julia function call '`Gen.simulate(poly_model, (xs,))`' implements $\mathcal{P}$.SIMULATE(x) where x is the tuple of arguments to the generative function (which in this case contains a single element `x_coordinates`). Then, we implement the test function as a Julia function and compute its average value across the traces:

```
g(trace) = trace[(:y, 1)] > trace[(:y, 2)]
estimate = sum([g(trace) for trace in traces]) / n
```

Note that as a syntactic sugar, traces in Gen allow direct access to the values of random choices by their address, using square brackets. That is, the syntax `trace`[a] where `trace` represents trace $\mathbf{t}$ evaluates to $\mathbf{t}$.CHOICES()[a]. For example in the code above, each value

`trace` represents $\mathbf{t}^{(i)} = (\mathcal{P}, x, \boldsymbol{\tau}^{(i)})$ where $\boldsymbol{\tau}^{(i)}$ is a choice dictionary, and `trace[(y, 1)]` evaluates to $\boldsymbol{\tau}^{(i)}[(\mathtt{y}, 1)]$. Running the Julia code above gives a simple Monte Carlo estimate of the desired probability. Figure 3-1 shows histograms of `estimate` collected from 1000 executions of this code, for different settings of n. The true probability is ≈ 0.5. Increasing n reduces the variance in the estimates as expected.

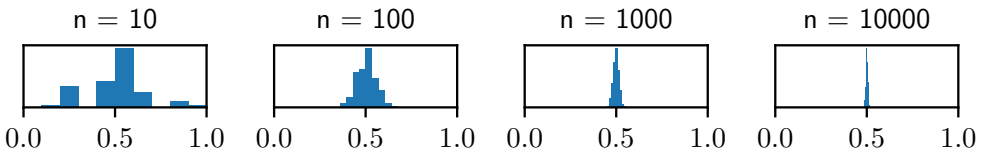


Figure 3-1: Convergence of simple Monte Carlo implemented with traces

Note that the test function whose expectation we estimate is defined *externally* to the generative function $\mathcal{P}$. It is also possible to define the return value of $\mathcal{P}$ to be $g(\boldsymbol{\tau})$, in which case the values $g(\boldsymbol{\tau}^{(i)})$ in Algorithm 1 are computed during SIMULATE, and are accessible from each trace via the RETVAL trace operation. However, this is undesirable because it reduces modularity of the code—modifying the test function would require modifying the model code. Also, encoding the test function separately from the model code allows the same collection of traces to be reused with many test functions.

3.2 Importance sampling with traces

Using simple Monte Carlo we estimated a property of a distribution p using samples from that distribution. More sophisticated Monte Carlo methods involve sampling from other probability distributions called *proposal distributions* that are denoted q. Importance sampling [104] is the simplest class of Monte Carlo methods that uses proposal distributions.

This section describes how to implement importance sampling algorithms using generative functions and traces. A key idea is that proposal distributions are represented in the same way as model distributions—as generative functions. Users write their proposal distributions using the same probabilistic modeling languages used to write generative models. Because proposals are defined using expressive modeling languages, a variety of different types of proposals are possible, including proposals based on the prior distribution, data-driven proposals, algorithmic proposals, simulator-based proposals, and proposals based on neural networks that are fully or partially learned from data. Proposal generative functions can be used for models with stochastic structure. In this section, generative functions representing a model are denoted $\mathcal{P}$, generative functions representing a proposal are denoted $\mathcal{Q}$, and the probability density on choice dictionaries for $\mathcal{Q}$ is denoted q.

The two design parameters of an importance sampling algorithm are the number of samples n and the proposal distribution q. Increasing n reduces error while increasing computational cost, and the choice of proposal distribution determines the efficiency of the algorithm (i.e. the error for a given n). Tailoring the proposal distribution to the

inference problem can vastly improve the efficiency of the algorithm. This motivates our use of flexible probabilistic modeling languages to express proposal distributions.

3.2.1 Regular importance sampling

Like simple Monte Carlo, regular importance sampling [104] is used to estimate expectations of a test function g under a distribution p. However, instead of sampling each $\boldsymbol{\tau}^{(i)}$ from p, we sample each $\boldsymbol{\tau}^{(i)}$ from q, and correct the estimate by 'weighting' each sample by an *importance weight* $p(\boldsymbol{\tau}^{(i)})/q(\boldsymbol{\tau}^{(i)})$, and then averaging the weighted values of the test function:

$$\mathbb{E}_{\boldsymbol{\tau}\sim p}[g(\boldsymbol{\tau})] \approx \frac{1}{n}\sum_{i=1}^{n} g(\boldsymbol{\tau}^{(i)})w^{(i)} \text{ where } w^{(i)} := \frac{p(\boldsymbol{\tau}^{(i)})}{q(\boldsymbol{\tau}^{(i)})} \text{ for } \boldsymbol{\tau}^{(i)} \overset{iid}{\sim} q \tag{3.2}$$

Like simple Monte Carlo, this estimator is unbiased and converges to the desired expectation as n increases. This procedure is often used when it is possible to sample from p, but simple Monte Carlo would give a high-variance estimate because g has large magnitude in regions of the state space that have low probability under p. For example, g may be the indicator function for an event that has low probability under p. In this setting, it is possible to choose q that reduces the variance of the estimator below that of simple Monte Carlo.

Algorithm 2 Regular importance sampling with traces

```
procedure IMPORTANCE-SAMPLING(P, Q, g, n)
    for i ← 1...n do
        s ← Q.SIMULATE(_)
        σ ← s.CHOICES()
        t^(i) ← P.GENERATE(_, σ)
        w^(i) ← exp(t^(i).LOGPDF() − s.LOGPDF())
    end for
    return (1/n) Σ_{i=1}^{n} w^(i) · g(t^(i).CHOICES())
end procedure
```

Algorithm 2 shows an implementation of regular importance sampling using a generative function $\mathcal{P}$ that encodes the distribution p, and a generative function $\mathcal{Q}$ that encodes the proposal distribution q. This algorithm associates each random choice sampled in $\mathcal{Q}$ with the random choice sampled in $\mathcal{P}$ that has the same address. In order to be a valid proposal distribution for use with $\mathcal{P}$, $\mathcal{Q}$ must use the same address universe as $\mathcal{P}$. In the measure-theoretic setting, this implies that $\mathcal{P}$ and $\mathcal{Q}$ must use the same reference measure for each address. For example, $\mathcal{P}$ cannot sample a discrete random choice at some address a while $\mathcal{Q}$ samples a continuous random choice. $\mathcal{Q}$ and $\mathcal{P}$ must also satisfy the following condition:

$$p(\boldsymbol{\tau}) > 0 \implies q(\boldsymbol{\tau}) > 0 \tag{3.3}$$

In particular, if $\mathcal{P}$ employs stochastic control flow, then $\mathcal{Q}$ must also employ stochastic control flow (although the Julia language constructs used to implement the control flow

need not be the same). Users of Gen must reason about the relationships between the sets of possible choice dictionaries $\boldsymbol{\tau}$ for the two generative functions when designing proposals. The current Gen implementation does not automatically verify that Equation (3.3) holds.

Example: Importance sampling and stochastic structure Consider the model $\mathcal{P} :=$ `p_inf_squares` below, and the function $g(\boldsymbol{\tau}) := \sum_{i=1}^{\infty}([\boldsymbol{\tau}[(\texttt{go}, i)]] \cdot \boldsymbol{\tau}[(\texttt{x}, i)]^2)$. We use Algorithm 2 to estimate the value of the expectation:

$$\mathbb{E}_{\boldsymbol{\tau} \sim p}[g(\boldsymbol{\tau})] = \sum_{i=1}^{\infty}(0.5)^i \sum_{j=1}^{i-1}(1 + j^2) = 7.0$$

(This example is chosen because it is possible to compute the expectation symbolically; in general this is not possible). We use proposal distribution $\mathcal{Q} :=$ `q_inf_squares` below:

```
@gen function p_inf_squares()
  i = 1; xtot = 0
  while ({(:go, i)} ~ bernoulli(0.5))
    xtot += ({(:x, i)} ~ normal(i, 1))^2
    i += 1
  end
  return xtot
end
```

```
@gen function q_inf_squares()
  i = 1
  while ({(:go, i)} ~ bernoulli(0.7))
    {(:x, i)} ~ normal(i, 1)
    i += 1
  end
end
```

Julia code implementing this estimator using the current version of Gen is:

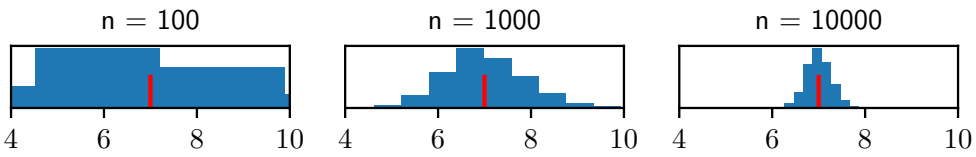
```
n = 1000
estimate = 0.0
for i in 1:n
    q_trace = Gen.simulate(q_inf_squares, ())
    (p_trace, _) = Gen.generate(p_inf_squares, (), Gen.get_choices(q_trace))
    w = exp(Gen.get_score(p_trace) - Gen.get_score(q_trace))
    estimate += w * Gen.get_retval(p_trace) # or estimate += g(p_trace)
end
estimate = estimate / n
```

Note that in this code, the test function g is computed within the body of the generative function $\mathcal{P}$, and stored in the return value of its trace $\mathbf{t}$, which is accessed using `Gen.get_retval`. It is also possible to instead implement the the test function separately in a Julia function `g` that reads the trace:

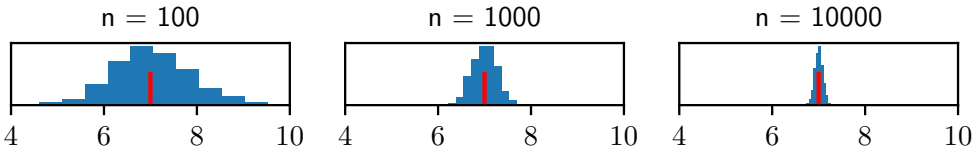
```
@gen function p_inf_squares()
    i = 1
    while ({(:go, i)} ~ bernoulli(0.5))
        {(:x, i)} ~ normal(i, 1)
        i += 1
    end
end
```

```
function g(trace)
    i = 1; xtot = 0
    while trace[(:go, i)]
        xtot += (trace[(:x, i)])^2
        i += 1
    end
    return xtot
end
```

Specifying g as part of the generative function has the benefit of reducing the total amount of code, but has the downside of tying the test function to the model code, so that the model code needs to be modified if the test function is modified. Figure 3-2 shows the results of running this code 1000 times, for different settings of n, and compares the results to those of a simple Monte Carlo estimator. The true value of the expectation (7.0) is shown in red. Increasing n reduces the variance in the estimates as expected. The importance sampling estimator is more accurate for a given n than the simple Monte Carlo estimator. Intuitively, this is because $\mathcal{Q}$ allocates more probability mass than $\mathcal{P}$ to traces for which $g(\boldsymbol{\tau})$ is large by increasing the likelihood that larger i are sampled.



(a) Simple Monte Carlo (Algorithm 1)



(b) Importance sampling (Algorithm 2)

Figure 3-2: Comparing accuracy of importance sampling and simple Monte Carlo

3.2.2 Self-normalized importance sampling

Regular importance sampling (Algorithm 2) can be used when distribution of interest p is the unconditioned distribution on choice dictionaries of a generative function. But typically in probabilistic inference the distribution of interest is a conditional distribution $p(\cdot|\boldsymbol{\rho})$ arising from conditioning such a distribution on observed data $\boldsymbol{\rho}$. In this chapter, we will assume that the observed data $\boldsymbol{\rho}$ are existentially sound and have nonzero marginal likeli-

hood ($\bar{p}(\boldsymbol{\rho}) > 0$). We will denote the latent choice dictionary by $\boldsymbol{\sigma}$. In this setting it is not typically possible to evaluate the target density $p(\boldsymbol{\sigma}|\boldsymbol{\rho}) := p(\boldsymbol{\sigma} \oplus \boldsymbol{\rho})/\bar{p}(\boldsymbol{\rho})$, and therefore it is not possible to use Algorithm 2. However, it is possible to evaluate $p(\boldsymbol{\sigma} \oplus \boldsymbol{\rho})$ and use an estimate of the marginal likelihood $\bar{p}(\boldsymbol{\rho})$ to approximate the importance weight used by regular importance sampling. This is the approach taken by *self-normalized importance sampling* [104], which uses a collection of samples from the proposal q to construct an estimate of the marginal likelihood, and weights these same samples by approximate importance weights obtained using the estimated marginal likelihood instead of $\bar{p}(\boldsymbol{\rho})$:

$$\mathbb{E}_{\boldsymbol{\sigma} \sim p(\cdot|\boldsymbol{\rho})}[g(\boldsymbol{\tau})] \approx \frac{1}{n} \sum_{i=1}^{n} g(\boldsymbol{\sigma}^{(i)}) w^{(i)} \quad \text{for} \quad w^{(i)} := \frac{p(\boldsymbol{\sigma}^{(i)} \oplus \boldsymbol{\rho})/\hat{z}}{q(\boldsymbol{\sigma}^{(i)})}, \; \boldsymbol{\sigma}^{(i)} \overset{iid}{\sim} q \tag{3.4}$$

where the estimate of the marginal likelihood is:

$$\hat{z} := \frac{1}{n} \sum_{j=1}^{n} \frac{p(\boldsymbol{\sigma}^{(j)} \oplus \boldsymbol{\rho})}{q(\boldsymbol{\sigma}^{(j)})} \tag{3.5}$$

Unlike regular importance sampling, this procedure is not in general unbiased, but it still converges asymptotically to the correct value. Note that self-normalized importance sampling can actually be more accurate than regular importance sampling even in the absence of conditioning [104] (e.g. with $\boldsymbol{\rho} = \{\}$).

Algorithm 3 shows an implementation of self-normalized importance sampling using generative functions and traces. Note that the sum of weights $w^{(i)}$ is 1. It is convenient to treat the weighted collection of traces $\{(\mathbf{t}^{(i)}, w^{(i)})\}_{i=1}^{n}$ as an approximation to the conditional distribution $p(\cdot|\boldsymbol{\rho})$, independently of the test function we are trying to estimate. For example, we can use the same weighted collection to estimate expectations of various test functions. Also, as we will see in Section 3.5, the weighted collection of traces can be subjected to further inference operations within an inference algorithm. Therefore, instead of taking a test function g as input and returning the estimated expectation as output, Algorithm 3 returns a weighted collection of traces that approximates a given conditional distribution $p(\cdot|\boldsymbol{\rho})$. It also returns the log marginal likelihood estimate $\log \hat{z}$, because this quantity is of intrinsic interest for model comparison. Note that each trace $\mathbf{t}^{(i)}$ contains both the latent choices $\boldsymbol{\sigma}^{(i)}$ and the observed choices $\boldsymbol{\rho}$, which are the same across all traces.

Since Algorithm 3 produces an approximation to the conditional distribution $p(\cdot|\boldsymbol{\rho})$ instead of an estimate of an expectation of some test function g, the proposal $\mathcal{Q}$ should be chosen to be accurate for various test functions. A general design principle for proposals that are not specialized to some test function g is to make the proposal distribution q as close as possible to the conditional distribution $p(\cdot|\boldsymbol{\rho})$. In particular, the number of samples that is necessary and sufficient to achieve a particular expected estimation error grows exponentially in the difference between target and proposal distributions, as measured by Kullback-Leibler divergence from target to proposal [21].

The requirement on the proposal generative function $\mathcal{Q}$ to be valid for self-normalized importance sampling is similar to that for regular importance sampling, but instead of

Algorithm 3 Self-normalized importance sampling with traces

procedure SELF-NORM-IMPORTANCE-SAMPLING($\mathcal{P}$, $\mathcal{Q}$, $\boldsymbol{\rho}$, n)
  for $i \leftarrow 1 \ldots n$ **do**
    $\mathbf{s} \leftarrow \mathcal{Q}$.SIMULATE(_)
    $\boldsymbol{\tau} \leftarrow \mathbf{s}$.CHOICES() $\oplus$ $\boldsymbol{\rho}$
    $\mathbf{t}^{(i)} \leftarrow \mathcal{P}$.GENERATE(_, $\boldsymbol{\tau}$)
    $\log \tilde{w}^{(i)} \leftarrow \mathbf{t}^{(i)}$.LOGPDF() $- \mathbf{s}$.LOGPDF()
  end for
  $((w^{(1)}, \ldots, w^{(n)}), \log \widehat{z}) \leftarrow$ NORMALIZE($\log \tilde{w}^{(1)}, \ldots, \log \tilde{w}^{(n)}$)
  return $\left(\{(\mathbf{t}^{(1)}, w^{(1)}), \ldots, (\mathbf{t}^{(n)}, w^{(n)})\}, \log \widehat{z}\right)$
end procedure
procedure NORMALIZE($\ell^{(1)}, \ldots, \ell^{(n)}$)
  $\bar{\ell} \leftarrow \max_j \ell^{(j)} + \log \left(\sum_{i=1}^{n} \exp(\ell^{(i)} - \max_j \ell^{(j)})\right)$
  for $i \leftarrow 1 \ldots n$ **do**
    $w^{(i)} \leftarrow \exp(\ell^{(i)} - \bar{\ell})$
  end for
  return $\left([w^{(1)}, \ldots, w^{(n)}], \bar{\ell}\right)$
end procedure

covering the support of p, the support of the q must cover the support of $p(\cdot|\boldsymbol{\rho})$:

$$p(\boldsymbol{\sigma} \oplus \boldsymbol{\rho}) > 0 \implies q(\boldsymbol{\sigma}) > 0 \tag{3.6}$$

Intuitively, the proposal 'fills in' values for the latent random choices in the model's trace by sampling its own random choices $\boldsymbol{\sigma}$ at the same addresses that are sampled in the model. These choices are then merged with the observed data $\boldsymbol{\rho}$ to construct the choice dictionary for the model $\boldsymbol{\tau}$. Note that if $p(\boldsymbol{\sigma}^{(i)} \oplus \boldsymbol{\rho}) = 0$ where $\boldsymbol{\sigma}^{(i)} :=$ $\mathbf{s}$.CHOICES() for all i in Algorithm 3 then $\log \tilde{w}^{(i)} = -\infty$ for all i, and it is not possible to compute a log marginal likelihood estimate or normalized weights $w^{(i)}$, and the procedure will error. While the probability of this event decreases to zero as n increases, to ensure the algorithm always successfully returns, the following converse support requirement is added:

$$q(\boldsymbol{\sigma}) > 0 \implies p(\boldsymbol{\sigma} \oplus \boldsymbol{\rho}) > 0 \tag{3.7}$$

Example: Potential proposals for self-normalized importance sampling Consider the generative function $\mathcal{P} :=$ `p_self_norm` defined below, and two generative functions $\mathcal{Q}_1 :=$ `q1_self_norm` and $\mathcal{Q}_2 :=$ `q2_self_norm` that encode potential proposals for use in Algorithm 3. For observations $\boldsymbol{\rho} := \{\texttt{c} \mapsto \texttt{T}\}$, $\mathcal{Q}_1$ is a valid importance sampling proposal for $\mathcal{P}$ and $\boldsymbol{\sigma}$ but $\mathcal{Q}_2$ is not because it never produces choice dictionaries containing address `b`. For example, for $\boldsymbol{\sigma} := \{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}\}$ we have $p(\boldsymbol{\sigma} \oplus \boldsymbol{\rho}) > 0$ but $q_2(\boldsymbol{\sigma}) = 0$.

```
@gen function p_self_norm()
  a ~ bernoulli(0.1)
  b ~ bernoulli(0.2)
  c ~ bernoulli((a && b) ? 0.9 : 0.1)
end
```

```
@gen function q1_self_norm()
  a ~ bernoulli(0.5)
  b ~ bernoulli(0.5)
end

@gen function q2_self_norm()
  a ~ bernoulli(0.5)
end
```

Example: Using the prior distribution as a proposal A simple way of constructing a generative function $\mathcal{Q}$ that is a valid proposal some pair of model $\mathcal{P}$ and observed data $\boldsymbol{\rho}$ is to write $\mathcal{Q}$ so that it samples from the *prior distribution* on the latent random choices. When $\mathcal{P}$ is written in a probabilistic modeling language like Gen's DML, it is often possible to obtain such a $\mathcal{Q}$ by simply removing the lines of code that sample the observed data from the model code. For example the following generative function $\mathcal{Q}$:= `q_is_prior` defines a valid proposal for the polynomial curve model $\mathcal{P}$:= `poly_model` (with lines for observed random choices commented out):

```
@gen function q_is_prior(x_coordinates)
    degree ~ uniform_discrete(0, 4)
    var ~ inv_gamma(1, 1)
    coefficients = [({(:c, i)} ~ normal(0, 1)) for i in 0:degree]
    # deleted lines that sample observed addresses (:y, i)
    # ..
end
```

Note that if the proposal instead iterated over '`i in 1:degree`' (skipping the constant-term coefficient at `i = 0`), then it would become invalid because all choice dictionaries $\boldsymbol{\sigma}$ that contain address $(\mathtt{c}, 0)$ would have $q(\boldsymbol{\sigma}) = 0$. Chapter 4 shows how the generative function and trace abstract data types can be extended so that simple proposals like this can be automatically generated by the modeling language compiler.

Example: A data-driven proposal distribution Although a proposal based on the prior distribution is easy to construct it is unlikely to result in an efficient importance sampling algorithm, because the prior distribution is usually not similar to the target distribution $p(\cdot|\boldsymbol{\rho})$. In order to better match the target distribution, it is necessary for the proposal to take into account the observed data $\boldsymbol{\rho}$. Such proposals are sometimes called *data-driven* [125]. One way of constructing data-driven proposals is to use a heuristic to estimate the mode of the target distribution (or one of its conditional distributions) and to sample values near the estimate of the mode, but with noise added. For example, consider conditional inference in $\mathcal{P}$:= `poly_model` given observations of the form $\boldsymbol{\rho} = \{(\mathtt{y}, 1) \mapsto y_1, \ldots (\mathtt{y}, 5) \mapsto y_5\}$ where the length of `x_coordinates` is 5. If we knew the degree of the polynomial, then we could use the least squares fit of a polynomial of that

degree as a heuristic estimate of the mode of the distribution on coefficients given data and degree. The generative function $\mathcal{Q}$:= `q_is_data_driven` below defines a proposal distribution that samples the degree and noise from their prior distributions, but runs least squares consecutively to estimate each coefficient in sequence. We propose each coefficient from a Cauchy distributions centered at the least squares estimate, with scale parameters that have been tuned for this model (see Section 3.3 for techniques for tuning parameters of proposal distributions).

```
@gen function q_is_data_driven(x_coords, y_coords)
    scales = [0.395, 0.242, 0.088, 0.020, 0.007]
    n = length(x_coords)
    @assert n == length(y_coords)
    degree ~ uniform_discrete(0, 4)
    coeffs = [NaN for i in 0:degree]
    predicted = [0.0 for i in 1:n]
    for i in 0:degree
        residuals = y_coords .- predicted # elementwise subtraction
        # fit a polynomial to residuals with coefficients 0..i-1 fixed to zero
        est_coeffs = least_squares(x_coords, residuals, degree, min_degree=i)
        coeffs[i+1] = ({(:c, i)} ~ cauchy(est_coeffs[1], scales[i+1]))
        predicted = [coeffs' * x.^(0:i) for x in x_coords]
    end
    residuals = y_coords .- predicted
    var ~ inv_gamma(1 + n/2, 1 + 0.5 * residuals' * residuals)
end
```

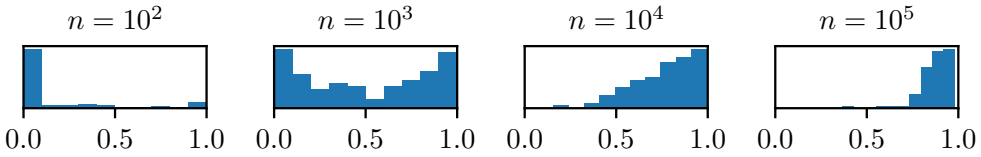
Note that this proposal takes both the input data (`x_coords`) and the observed data `y_coords`) as arguments. In general, proposal generative functions are allowed to take arbitrary arguments. Also, crucially, the proposal is able to run least-squares Julia code. The ability to run general-purpose code as part of a proposal distribution is a key motivator for Gen's use of flexible probabilistic modeling languages to express proposal distributions. The proposal finishes by sampling the variance from its conditional distribution, which was derived using conjugate analysis.

Suppose our goal is to estimate the conditional probability that the degree is three for some data set. The Julia code below uses Gen's inference library implementation of Algorithm 3, with the data-driven proposal $\mathcal{Q}$. Note that the inference library implementation returns log-weights $\log w^{(i)}$ instead of $w^{(i)}$.

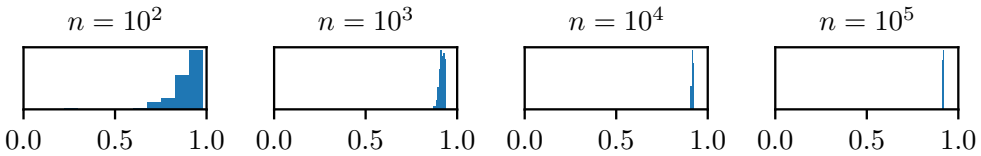
```
observations = Gen.choicemap() # initialize empty choice dictionary
for i in 1:length(y_coords)
    observations[(:y, i)] = y_coords[i]
end
(traces, log_weights) = Gen.importance_sampling(
        poly_model, (x_coords,), observations,
        q_is_data_driven, (x_coords, y_coords), n)
weights = exp.(log_weights)
estimate = sum(weights .* map(g, traces))
```

Figure 3-3 shows the results of running this code 100 times, for different settings of n,

and compares the results with that of the same code, but with the less efficient proposal `q_is_prior`. Increasing n reduces the variance in the estimates as expected. The importance sampling estimator that uses the data-driven proposal is much more accurate for given n than the estimator using the prior as the proposal (the true value is near 0.9). Note that many other probabilistic programming systems with universal modeling languages *exclusively* support importance sampling proposals based on the prior distribution [130, 40].



(a) Estimates from self-normalized importance sampling with a prior proposal.



(b) Estimates from self-normalized importance sampling with a data-driven proposal.

Figure 3-3: Comparing self-normalized importance sampling using different proposals

3.3 Training proposal distributions on simulated data

Manually devising efficient proposal distributions for use in self-normalized importance sampling and other Monte Carlo algorithms can be difficult. The process of constructing an efficient proposal generative function $\mathcal{Q}$ can be partially automated by training numerical parameters in $\mathcal{Q}$ using supervised learning on training data that is generated from the generative model $\mathcal{P}$. The general strategy of training an proposal or inference computation on data simulated from a generative model has a long history in the machine learning and inference literature [58, 87]. The strategy can be motivated by mathematical relationship between maximum likelihood learning of discriminative models and Kullback-Leibler (KL) divergence. Consider a generative model distribution $p(\boldsymbol{\sigma} \oplus \boldsymbol{\rho})$ where $\boldsymbol{\sigma}$ contains latent choices and $\boldsymbol{\rho}$ contains observed random choices. We can train a family of proposals distributions $q(\cdot; \boldsymbol{\rho}, \theta)$, where $\boldsymbol{\rho}$ is observed data, and θ are numerical parameters of the proposal program, to maximize the expectation of the log-likelihood $\log q(\boldsymbol{\sigma}; \boldsymbol{\rho}, \theta)$, where the latent and observed data $\boldsymbol{\sigma}$ and $\boldsymbol{\rho}$ are jointly sampled from p:

$$\max_{\theta} \mathbb{E}_{(\boldsymbol{\sigma} \oplus \boldsymbol{\rho}) \sim p} \left[\log q(\boldsymbol{\sigma}; \boldsymbol{\rho}, \theta) \right] \tag{3.8}$$

This is equivalent to solving the following optimization problem:

$$\min_{\theta} \mathbb{E}_{\boldsymbol{\rho} \sim p} \left[\mathrm{D}_{\mathrm{KL}}(p(\cdot|\boldsymbol{\rho}) || q(\cdot; \boldsymbol{\rho}, \theta)) \right] \tag{3.9}$$

The equivalence has been noted several times in recent literature [16, 70, 31, 71]. Because the KL divergence $\mathrm{D}_{\mathrm{KL}}(p(\cdot|\boldsymbol{\rho})||q(\cdot; \boldsymbol{\rho}, \theta))$ governs the efficiency of a proposal distribution [21], the equivalence between Equation (3.8) and Equation (3.9) implies that Monte Carlo proposals can be tuned using supervised learning on data simulated from the generative model. The optimization problem in Equation (3.8) can be solved using stochastic gradient techniques with gradient estimates obtained from minibatches of simulated data.

This training procedure is implemented naturally using Gen's abstract data types for generative functions and traces. Algorithm 4 gives a procedure for training numerical parameters θ of a generative function $\mathcal{Q}$ that encodes a proposal distribution, on data generated from a generative function $\mathcal{P}$ that encodes a generative model. The procedure assumes that the numerical parameters θ are arguments to the generative function $\mathcal{Q}$, and it obtains gradients of $\log q(\boldsymbol{\sigma}; \boldsymbol{\rho}, \theta)$ with respect to θ using the GRADIENTS trace operation. The same procedure can be used to train parameters of generative functions for use as proposals within the Metropolis-Hastings algorithms that will be described in Section 3.4.2 and the particle filtering algorithms that will be described in Section 3.5. Also note that with small adjustments this procedure can be used for learning of discriminative generative functions $\mathcal{Q}$ directly from real data, or from combinations of real and simulated data.

Algorithm 4 Training parameters of a proposal on data generated from a model

procedure SIMULATED-DATA-TRAIN-SGD($\mathcal{P}$, $\mathcal{Q}$, θ_0, α, k, m)
  for $i \leftarrow 1 \ldots k$ **do**
    $\mathbf{u} \leftarrow \mathbf{0}$
    for $j \leftarrow 1 \ldots m$ **do**
      $\mathbf{t} \leftarrow \mathcal{P}$.SIMULATE(_)
      $(\boldsymbol{\sigma} \oplus \boldsymbol{\rho}) \leftarrow \mathbf{t}$.CHOICES() ▷ Separate out latents $\boldsymbol{\sigma}$ and observations $\boldsymbol{\rho}$
      $\mathbf{s} \leftarrow \mathcal{Q}$.GENERATE($(\boldsymbol{\rho}, \theta_{i-1}), \boldsymbol{\sigma}$)
      $((\_, \mathbf{v}), \_) \leftarrow \mathbf{s}$.GRADIENT($\{\}, \mathbf{0}$) ▷ $\mathbf{v} = \nabla_\theta \log q(\boldsymbol{\sigma}; \boldsymbol{\rho}, \theta)$ at $\theta = \theta_{i-1}$
      $\mathbf{u} \leftarrow \mathbf{u} + (1/m) \cdot \mathbf{v}$
    end for
    $\theta_i \leftarrow \theta_{i-1} + \alpha \cdot \mathbf{u}$
  end for
  return θ_k
end procedure

Example: Automatically tuning the stochasticity of a heuristic-based proposal Consider the proposal $\mathcal{Q} :=$ `q_is_data_driven` above. This data-driven proposal uses least-squares to determine the mode of the (Cauchy) sampling distribution for each coefficient. The numerical parameters '`scales`' govern how stochastic the sampling distribution of each coefficient should be. Intuitively, if the scale parameters are too large, then the proposal

will often propose samples with very low posterior probability, and the importance sampling algorithm will require many particles in order for an significant number to lie in regions of appreciable posterior probability. The algorithm may give overly uncertain estimates unless the number of particles is very large. If the scale parameters are too small, then the regions of the parameter space will not be sampled frequently enough, and the algorithm may give overconfident estimates. The code below implements Algorithm 4 in Julia, using the version of Gen at the time of this writing:

```
n = length(x_coords)
theta = [10.0, 10.0, 10.0, 10.0, 10.0]
for iter in 1:500 # iterations of stochastic gradient descent
   grad_accumulator = [0.0, 0.0, 0.0, 0.0, 0.0]
   for i in 1:100 # minibatch size
      p_trace = Gen.simulate(poly_model, (x_coords,))
      q_args = (x_coords, [p_trace[(:y, i)] for i in 1:n], log_scales)
      latents = Gen.complement(Gen.select([(:y, i) for i in 1:n]...))
      latent_choices = Gen.get_selected(Gen.get_choices(p_trace), latents)
      (q_trace, _) = Gen.generate(q_is_data_driven, q_args, latent_choices)
      ((_,_,grad),_) = Gen.choice_gradients(q_trace, Gen.select())
      grad_accumulator .+= grad / 100
   end
   theta .+= (grad_accumulator * 0.1)
end
```

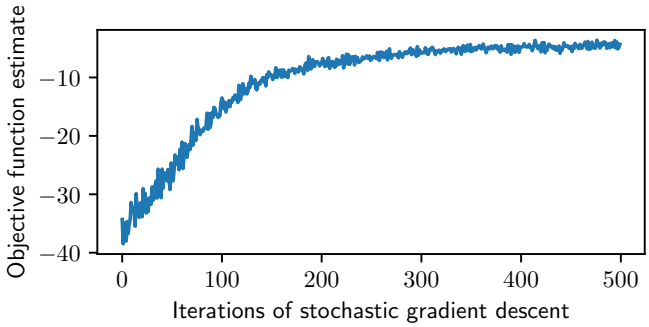
This algorithm uses `Gen.choice_gradients`, which implements the GRADIENT abstract data type operation. Note that the program `q_is_data_driven` was modified to add a third argument `log_scales`. We use the log of the scale vector as the argument so that optimization can be performed over an unconstrained space. Also, we label the argument with '`grad`' to indicate that gradients with respect to this argument will be required:

```
@gen function q_is_data_driven(xs, ys, (grad)(log_scales))
    scales = exp.(log_scales)
    ..
end
```

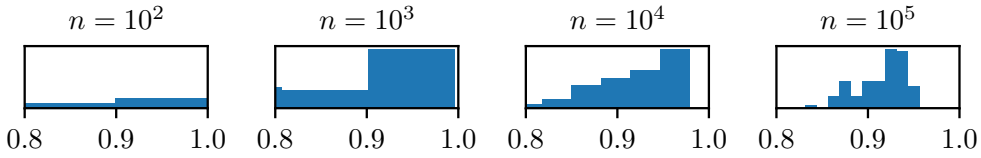
Not that Gen also has more specialized gradient operations that can be used when optimizing large parameter arrays. The code above is chosen for simplicity, and to use only the subset of Gen's abstract data type operations that were introduced in Chapter 2.3.

Figure 3-4a shows the estimated objective function over the course of training these parameters using Algorithm 4, where data is simulated from the generative model $\mathcal{P} :=$ `poly_model`. Figure 3-4b shows the estimates of the conditional probability that the degree of the polynomial is three, obtained using self-normalized importance sampling with proposal `q_is_data_driven` without training (all scale parameters were set heuristically to 0.5 for all coefficients based on qualitative accuracy on some simulated data sets) and after training (the values for the trained scale parameters are shown in the body of `q_is_data_driven`. Each of the two algorithms was run 100 times each for four different numbers of particles ranging from $n = 10^2$ to $n = 10^5$ and the estimated probabilities from the 100 runs were aggregated histograms. While both algorithms appear to converge to

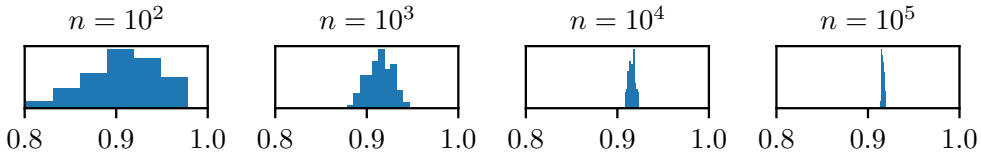
the same value as n grows, the algorithm after training requires orders of magnitude fewer particles to achieve comparable levels of accuracy.



(a) Objective function estimates over the course of parameter training.



(b) Conditional probability estimates using self-normalized importance sampling with a data-driven proposal, with heuristically chosen scale parameters, for different numbers of particles n.



(c) Conditional probability estimates using self-normalized importance sampling with a data-driven proposal, after training the scale parameters on simulated data, for different numbers of particles n.

Figure 3-4: Training the parameters of a data-driven importance-sampling proposal

3.4 Markov chain Monte Carlo with traces

Markov chain Monte Carlo (MCMC) is a flexible framework for constructing algorithms that sample approximately from target probability distributions that are defined by an unnormalized density function [104], and conditional distributions induced by generative models in particular. In the setting of generative models, MCMC algorithms initialize a state that contains the values of latent random variables, and then repeatedly apply a stochastic *kernel* to this state, producing a new state. This section shows how to implement MCMC algorithms using Gen's generative function and trace abstract data types. The section includes (i) a programming construct for primitive MCMC kernels based on Metropolis-Hastings

with arbitrary proposal distributions encoded as generative functions, (ii) a programming construct for Hamiltonian Monte Carlo, and (iii) a language for composing MCMC kernels into more complex kernels. This section focuses primarily on constructing kernels that are *stationary* with respect to the target distribution; Gen does not aid users in evaluating the ergodicity of their MCMC kernels.

3.4.1 MCMC with the trace abstract data type

MCMC is implemented in Gen using traces (Section 2.3) to store the state of the latent and observed random choices. In this framework, an MCMC kernel is a procedure KERN takes as input a trace $\mathbf{t}$ of the model $\mathcal{P}$ and returns as output a new trace $\mathbf{t}'$ of $\mathcal{P}$. Let $\boldsymbol{\rho}$ denote an existentially sound choice dictionary of observed data with positive marginal likelihood ($\bar{p}(\boldsymbol{\rho}) > 0$), so that the conditional distribution $p(\cdot|\boldsymbol{\rho})$ is well-defined. An MCMC algorithm targeting $p(\cdot|\boldsymbol{\rho})$ generates a sequence of traces $\mathbf{t}_0, \mathbf{t}_1, \ldots, \mathbf{t}_n$ such that each trace contains the observed choices $\boldsymbol{\rho}$ (which are the same across all iterations i) as well as the latent choices ($\boldsymbol{v}_i$, which change across iterations):

$$\mathbf{t}_i.\text{CHOICES}() = \boldsymbol{v}_i \oplus \boldsymbol{\rho} \text{ for all } i = 1, \ldots, n$$

This thesis describes several constructions for MCMC kernels that are stationary with respect to a given distribution $p(\cdot|\boldsymbol{\rho})$, but the following template for constructing MCMC algorithms is the same regardless of the kernel used. First, we obtain the initial trace $\mathbf{t}_0$ for an MCMC chain using

$$\mathbf{t}_0 \leftarrow \mathcal{P}.\text{GENERATE}(\_, \boldsymbol{v}_0 \oplus \boldsymbol{\rho})$$

where $\boldsymbol{v}_0$ is some choice dictionary such that $p(\boldsymbol{v}_0|\boldsymbol{\rho}) > 0$ (recall that '_' denotes the arguments to $\mathcal{P}$, which are treated as constant for the purposes of this section). Here, $\boldsymbol{v}_0$ encodes the initial values for the latent random choices. We may obtain $\boldsymbol{v}_0$ by simulating from a generative function $\mathcal{Q}_0$:

$$\begin{aligned}&\mathbf{s} \leftarrow \mathcal{Q}_0.\text{SIMULATE}()\\ &\boldsymbol{v}_0 \leftarrow \mathbf{s}.\text{CHOICES}()\end{aligned}$$

After the initial trace $\mathbf{t}_0$ is obtained, an MCMC kernel KERN is repeatedly applied to the trace n times to generate the MCMC chain (where in general, KERN is stochastic):

$$\begin{aligned}&\textbf{for } i \leftarrow 1 \ldots n \textbf{ do}\\ &\quad \mathbf{t}_i \leftarrow \text{KERN}(\mathbf{t}_{i-1})\\ &\textbf{end for}\end{aligned}$$

Then, the collection of traces $\{\mathbf{t}_i\}_{i=b}^n$ after some number of 'burn-in' iterations b can be treated as a particle approximation to $p(\cdot|\boldsymbol{\rho})$, assuming that KERN is ergodic and n is large. Users of Gen implement MCMC algorithms by writing code in a general-purpose programming language (Julia) that implements the simple template above. Other probabilistic programming systems that support MCMC place the initialization and loop over iterations within their internal inference engine implementation. Gen's approach makes

the algorithm's high-level structure explicit in user code, which allows users to debug and instrument the algorithm without having to understand the inference engine or compiler architecture of a probabilistic programming system.

The remainder of this section gives constructions for kernels KERN that are stationary with respect to the target distribution $p(\cdot|\boldsymbol{\rho})$, including a class of primitive kernels based on Metropolis-Hastings and a class of composite kernels that combine other kernels using loops, branching, and random mixtures. Section 3.7 gives another more general class construction for MCMC kernels on traces that can express any kernel in the reversible jump MCMC framework [53].

3.4.2 Metropolis-Hastings using generative functions as proposals

Metropolis-Hastings (MH) [22] is a general technique for constructing kernels with the correct stationary distribution that involves first constructing a *proposal kernel*, which is a transition kernel that does not in general have the correct stationary distribution. The MH kernel invokes the proposal kernel and either returns the proposed state (this is called *accepting*) or returns the previous state (this is called *rejecting*). The decision to accept or reject is stochastic. The probability of accepting a proposed state is constructed using a standard formula to ensure the correct stationary distribution for the MH kernel.

Implementing a Metropolis-Hastings kernel from scratch requires (i) a data structure to store the latent state, (ii) a sampler for the proposal, (iii) an evaluator for the density of the proposal, and (iv) an evaluator for the density of the unnormalized target distribution. Gen's construct for Metropolis-Hastings (MH-KERNEL, Algorithm 5) automates these implementation details, while still allowing the user to hand-design their proposal. The construct is expressive enough to allow for kernels that change the structure (e.g. control flow) of the model's trace. This construct is based on the following key ideas:

1. A Metropolis-Hastings proposal is represented as a generative function $\mathcal{Q}$ that takes in the current trace $\mathbf{t}$ of the model generative function $\mathcal{P}$ as an argument. Like the model generative function, the proposal generative function $\mathcal{Q}$ is defined using a probabilistic modeling language like Gen's Dynamic Modeling Language.

2. $\mathcal{Q}$ determines how the trace of the model should be changed by making random choices at the *same addresses* used by the model $\mathcal{P}$. The proposal's choices $\boldsymbol{\sigma}$ are used to replace the values at some addresses in the previous model trace and may change the structure (control flow) of the model trace, in which case the choices are also used to fill in values for any newly introduced addresses.

3. Any choices in the previous model trace $\mathbf{t}$ that are not replaced by choices in $\boldsymbol{\sigma}$, and are not removed from the trace as a result of control flow changes, are automatically retained in the new model trace $\mathbf{t}'$.

In particular, for a current model trace $\mathbf{t}$, the proposal $\mathcal{Q}$ is simulated (SIMULATE), and the resulting choices $\boldsymbol{\sigma}$ are passed as constraints to $\mathbf{t}$.UPDATE, which returns the proposed trace $\mathbf{t}'$ and metadata ($\log w$ and $\boldsymbol{\sigma}'$) that is used to compute the acceptance probability.

Algorithm 5 Metropolis-Hastings kernel using a generative function as the proposal

procedure MH-KERNEL$_{\mathcal{Q},\boldsymbol{\rho}}(\mathbf{t})$
  $\mathbf{s} \leftarrow \mathcal{Q}$.SIMULATE$(\mathbf{t})$
  $\boldsymbol{\sigma} \leftarrow \mathbf{s}$.CHOICES$()$
  $(\mathbf{t}', \log w, \boldsymbol{\sigma}', \_) \leftarrow \mathbf{t}$.UPDATE$(_, \top, \boldsymbol{\sigma})$
  assert $|A_{\boldsymbol{\rho}} \cap A_{\boldsymbol{\sigma}'}| = 0$
  $\mathbf{s}' \leftarrow \mathcal{Q}$.GENERATE$(\mathbf{t}', \boldsymbol{\sigma}')$
  $\alpha \leftarrow \min\{1, \exp(\log w - \mathbf{s}'$.LOGPDF$() + \mathbf{s}$.LOGPDF$())\}$
  $r \sim \text{Uniform}(0, 1)$
  if $r \leq \alpha$ **then return** $\mathbf{t}'$ **else return** $\mathbf{t}$
end procedure

In order for MH-KERNEL$_{\mathcal{Q},\boldsymbol{\rho}}$ to be stationary with respect to the target distribution $p(\cdot|\boldsymbol{\rho})$, the proposal generative function $\mathcal{Q}$ must satisfy certain requirements with respect to the model generative function $\mathcal{P}$ and the observations $\boldsymbol{\rho}$. First, it must take an argument of the form $\mathbf{t}$, where $\mathbf{t}$ is a trace of $\mathcal{P}$. Second, it must never make a random choice at an observed address $a \in A_{\boldsymbol{\rho}}$:

$$q(\boldsymbol{\sigma}; \mathbf{t}) > 0 \implies |A_{\boldsymbol{\sigma}} \cap A_{\boldsymbol{\rho}}| = 0$$

Third, for all $(\mathbf{t}, \boldsymbol{v}, \boldsymbol{\sigma})$ where $p(\boldsymbol{v}|\boldsymbol{\rho}) > 0$ and $\mathbf{t}$.CHOICES$() = \boldsymbol{v} \oplus \boldsymbol{\rho}$ and $q(\boldsymbol{\sigma}; \mathbf{t}) > 0$, there must exist some $\mathbf{t}'$ and $B \subseteq A_{\boldsymbol{v}}$ such that $\mathbf{t}'$.CHOICES$() = \boldsymbol{\sigma} \oplus (\boldsymbol{v}|_B) \oplus \boldsymbol{\rho}$ and $p(\boldsymbol{\sigma} \oplus (\boldsymbol{v}|_B) \oplus \boldsymbol{\rho}) > 0$ and $q(\boldsymbol{\sigma}'; \mathbf{t}') > 0$ where $\boldsymbol{\sigma}' := \boldsymbol{v}|_{B^\mathtt{c}}$. The third requirement means that (i) every choice dictionary $\boldsymbol{\sigma}$ that can be sampled from $\mathcal{Q}$ given input $\mathbf{t}$ can be used to construct a new choice dictionary $(\boldsymbol{\sigma} \oplus (\boldsymbol{v}|_B) \oplus \boldsymbol{\rho})$ that has nonzero density under the model, by merging $\boldsymbol{\sigma}$ with some subset B of the previous latent choices $(\boldsymbol{v}|_B)$ and the observations $(\boldsymbol{\rho})$, and that (ii) for every possible proposed transition from $\boldsymbol{v}$ to $\boldsymbol{v}'$, it is possible to reverse this transition using some proposed choices $\boldsymbol{\sigma}'$ that could be sampled from $\mathcal{Q}$ given the new trace $\mathbf{t}'$ as input.

Example: Invalid Metropolis-Hastings proposals Consider the model $\mathcal{P}$:= `p_mh` below, and $\boldsymbol{\rho} := \{c \mapsto \mathtt{T}\}$. The conditional distribution $p(\cdot|\boldsymbol{\rho})$ is shown to the right.

```
@gen function p_mh()
    z = ({:a} ~ bernoulli(0.5))
    if z
        z = z && ({:b} ~ bernoulli(0.5))
    end
    c ~ bernoulli(z ? 0.9 : 0.1)
end
```

$\boldsymbol{v}$	$p(\boldsymbol{v}\|\boldsymbol{\rho})$
$\{\mathtt{a} \mapsto \mathtt{F}\}$	0.167
$\{\mathtt{a} \mapsto \mathtt{T}, \mathtt{b} \mapsto \mathtt{F}\}$	0.083
$\{\mathtt{a} \mapsto \mathtt{T}, \mathtt{b} \mapsto \mathtt{T}\}$	0.750

Consider three potential proposals $\mathcal{Q}_1$:= `q1_mh`, $\mathcal{Q}_2$:= `q2_mh` and $\mathcal{Q}_3$:= `q3_mh` defined below. Each is invalid for a different reason. $\mathcal{Q}_1$ violates the second requirement because it samples at address `c`. $\mathcal{Q}_2$ violates the third requirement because for $\boldsymbol{v} = \{\mathtt{a} \mapsto \mathtt{F}\}$ and $\boldsymbol{\sigma} = \{\mathtt{a} \mapsto \mathtt{T}\}$, there is no new trace $\mathbf{t}'$ of $\mathcal{P}$ that can be constructed, because a value for address `b` is not included in $\boldsymbol{\sigma}$. $\mathcal{Q}_3$ violates the third requirement because for

$\boldsymbol{v} = \{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}\}$ and $\boldsymbol{\sigma} = \{\texttt{a} \mapsto \texttt{F}\}$, the proposed choices that would reverse the move are $\boldsymbol{\sigma}' = \{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}\}$ but $q(\boldsymbol{\sigma}'; \mathbf{t}') = 0$.

```
@gen function q1_mh(t)
    c ~ bernoulli(0.5)
end
```

```
@gen function q2_mh(t)
    a ~ bernoulli(0.5)
end
```

```
@gen function q3_mh(t)
    a ~ bernoulli(0.0)
end
```

Example: Valid Metropolis-Hastings proposal Consider generative function $\mathcal{Q} :=$ `q4_mh` below. $\mathcal{Q}$ is a valid proposal for the conditional distribution above. The proposal proposes to switch branches (from $\texttt{a} \mapsto \texttt{T}$ to $\texttt{a} \mapsto \texttt{F}$ or vice-versa) with probability 0.9. If the previous model trace $\mathbf{t}$ had $\texttt{a} \mapsto \texttt{T}$ and it does not switch branches, then it retains the previous value for `b`. If the previous model trace had $\texttt{a} \mapsto \texttt{F}$ and it does switch branches (setting $\texttt{a} \mapsto \texttt{T}$) then it samples a fresh value for `b`.

```
@gen function q4_mh(t)
  if t[:a]
    a ~ bernoulli(0.1)
  else
    if ({:a} ~ bernoulli(0.9))
      b ~ bernoulli(0.5)
    end
  end
end
```

Below, for every possible previous latent choice dictionary $\boldsymbol{v}$ and every possible proposed choice dictionary $\boldsymbol{\sigma}$, we show the set of retained choices B and the resulting proposed latent choice dictionary $\boldsymbol{v}'$ and the proposed choice dictionary $\boldsymbol{\sigma}'$ that would reverse the move:

$\boldsymbol{v}$	$\boldsymbol{\sigma}$	B	$\boldsymbol{v}'$	$\boldsymbol{\sigma}'$
$\{\texttt{a} \mapsto \texttt{F}\}$	$\{\texttt{a} \mapsto \texttt{F}\}$	$\{\}$	$\{\texttt{a} \mapsto \texttt{F}\}$	$\{\texttt{a} \mapsto \texttt{F}\}$
$\{\texttt{a} \mapsto \texttt{F}\}$	$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}\}$	$\{\}$	$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}\}$	$\{\texttt{a} \mapsto \texttt{F}\}$
$\{\texttt{a} \mapsto \texttt{F}\}$	$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{T}\}$	$\{\}$	$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{T}\}$	$\{\texttt{a} \mapsto \texttt{F}\}$

$\boldsymbol{v}$	$\boldsymbol{\sigma}$	B	$\boldsymbol{v}'$	$\boldsymbol{\sigma}'$
$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}\}$	$\{\texttt{a} \mapsto \texttt{T}\}$	$\{\texttt{b}\}$	$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}\}$	$\{\texttt{a} \mapsto \texttt{T}\}$
$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}\}$	$\{\texttt{a} \mapsto \texttt{F}\}$	$\{\}$	$\{\texttt{a} \mapsto \texttt{F}\}$	$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}\}$

$\boldsymbol{v}$	$\boldsymbol{\sigma}$	B	$\boldsymbol{v}'$	$\boldsymbol{\sigma}'$
$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{T}\}$	$\{\texttt{a} \mapsto \texttt{T}\}$	$\{\texttt{b}\}$	$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{T}\}$	$\{\texttt{a} \mapsto \texttt{T}\}$
$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{T}\}$	$\{\texttt{a} \mapsto \texttt{F}\}$	$\{\}$	$\{\texttt{a} \mapsto \texttt{F}\}$	$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{T}\}$

Example: Random walk proposal Proposals based on applying a random walk to a single random variable are a simple and common (but generally inefficient) type of Metropolis-Hastings proposal. This is a simple class of proposals that never alters the structure of the trace. Recall the generative model $\mathcal{P} :=$ `poly_model`. The proposal $\mathcal{Q} :=$ `q_mh_random_walk` below performs a random walk on one of the coefficients of the

polynomial in this model. The proposal reads the previous value of the coefficient from the model trace at address $(\texttt{c}, i)$, and then samples the new value from a normal distribution centered at the previous value, at the *same address*.

```
@gen function q_mh_random_walk(trace, i::Int)
    {(:c, i)} ~ normal(trace[(:c, i)], 0.05)
end
```

Note that the generative function $\mathcal{Q}$ takes a second argument (`i`) in addition to the model trace ($\mathbf{t}$). The second argument indicates which coefficient should be proposed to. Proposal generative functions $\mathcal{Q}$ that take additional arguments besides the previous model trace actually determine a parametrized *family* of kernels $\{\text{MH-KERNEL}_{\mathcal{Q},\boldsymbol{\rho},x}\}_{x \in X}$, where x indexes the possible values of the additional arguments to $\mathcal{Q}$.

Example: Data-driven proposal Random walk proposals on individual random choices are relatively inefficient because they only operate on one choice at a time, and they can take a long time to move to regions of higher probability (especially if the standard deviation of the random walk is too small or too large). Random walk proposals are not informed by the the observed data. Instead of proposing a new value for the ith coefficient from a random walk around its previous value, we can estimate the value of the coefficient using a least-squares fit to the residuals resulting from setting the ith coefficient to zero:

$$\min_{c_i} \left\| \left(\mathbf{y} - \sum_{j \neq i} c_j [x_1^j, \ldots, x_n^j]^\top \right) - c_i [x_1^i, \ldots, x_n^i]^\top \right\|_2$$

where $\mathbf{x}$ and $\mathbf{y}$ are the input x-coordinates and the observed y-coordinates, respectively. The data-driven proposal below solves the optimization problem above to find $\widehat{c}_i$ and then proposes a new value at address $(\texttt{c}, i)$ by sampling from a Cauchy distribution centered at $\widehat{c}_i$. Unlike the random walk proposal, this proposal is capable of proposing large jumps in the latent space.

```
@gen function q_mh_data_driven(trace, x_coords, y_coords, i)
    n = length(x_coords)
    degree = trace[:degree]
    coeffs = [trace[(:c, i)] for i in 0:degree]

    # least-squares for coefficient i with other coefficients fixed
    coeffs[i+1] = 0.0
    residuals = y_coords .- [coeffs' * x.^(0:degree) for x in x_coords]
    x_vec = [x^i for x in x_coords]
    coefficient_guess = (x_vec' * residuals) / sqrt(x_vec' * x_vec)

    # sample around the estimated coefficient
    {(:c, i)} ~ cauchy(coefficient_guess, 0.5)
end
```

Example: Gibbs sampling Gibbs sampling is equivalent to a Metropolis-Hastings kernel where the proposal is the conditional distribution on the proposed-to addresses. In particular, the MH kernel constructed from the proposal below is equivalent to Gibbs sampling move on the variance:

```
@gen function q_var_gibbs(trace, x_coords, y_coords)
    n = length(x_coords)
    coeffs = [trace[(:c, i)] for i in 0:trace[:degree]]
    residuals = y_coords .- [coeffs' * x.^(0:trace[:degree]) for x in x_coords]
    var ~ inv_gamma(1 + n/2, 1 + 0.5 * residuals' * residuals)
end
```

Note that this move will always be accepted. The conditional distribution was derived by hand, which is possible because the inverse-gamma distribution is the conjugate prior on the variance for a normal likelihood.

Example: Structure-changing proposal The previous two proposals did not alter the structure of the model trace. For the polynomial curve model, the structure of the trace is determined by the value at address `degree`. The proposal below samples a new degree. If the new degree is less than or equal to the previous degree, then it does not sample any additional random choices— it simply removes coefficients from the polynomial. If the new degree is greater than the previous degree, then it samples values for the newly introduced coefficients near zero.

```
@gen function q_mh_structure(trace)
    prev_degree = trace[:degree]
    new_degree = ({:degree} ~ uniform_discrete(0, 4))
    for i in prev_degree+1:new_degree
        {(:c, i)} ~ cauchy(0.0, 0.4)
    end
end
```

Note that if the proposal did not sample values for the newly introduced coefficients, then it would not be a valid proposal, because it would not be possible to construct the new trace $\mathbf{t}'$ without an assignment to these addresses.

Using the Gen inference library implementation of Metropolis-Hastings The Gen inference library contains an implementation of MH-KERNEL in the Julia function `Gen.mh`. This function returns a tuple containing the trace and metadata. The code below shows how to implement an MCMC algorithm in Julia using Gen's inference library. In particular, we construct a MCMC chain that repeatedly samples the MH kernel constructed from the proposal $\mathcal{Q} :=$ `q_mh_structure`.

```
traces = []
kernel(trace) = Gen.mh(trace, q_mh_structure, ())[1]
trace = init_trace
for i=1:n
    trace = kernel(trace)
    push!(traces, trace)
end
```

Figure 3-5shows the results of running this code with each of the different proposal generative functions $\mathcal{Q}$ defined above, starting with the same initial trace of $\mathcal{P} :=$ `poly_model`. The results emphasize that none of these kernels is individually sufficient for an efficient MCMC algorithm. The values at five addresses are plotted for the first 100 iterations. The kernel constructed from `q_mh_random_walk` with `i` $= 0$ makes small perturbations to the constant-term coefficient. The kernel constructed from `q_mh_data_driven` with `i` $= 0$ makes larger perturbations to the constant-term coefficient. The kernel constructed from `q_var_gibbs` samples from the conditional distribution on variance with other parameters fixed. The kernel constructed from `q_mh_structure` makes changes to the degree. Only the kernel based on the structure-changing move is ergodic, but it is very inefficient because its proposals for new coefficient values are uninformed by both the data and the previous values. In contrast, the data-driven proposal gives a kernel that can give rough estimates of the coefficients for a fixed degree, but cannot estimate the degree itself, and cannot efficiently fine-tune the value of any given coefficient. The kernel based on the random walk proposal is capable of fine-tuning the value of individual coefficients, but cannot estimate the degree or the value of the other coefficients. Section 3.4.4 will show how to compose kernels like these into more powerful kernels that result in practical MCMC algorithms.

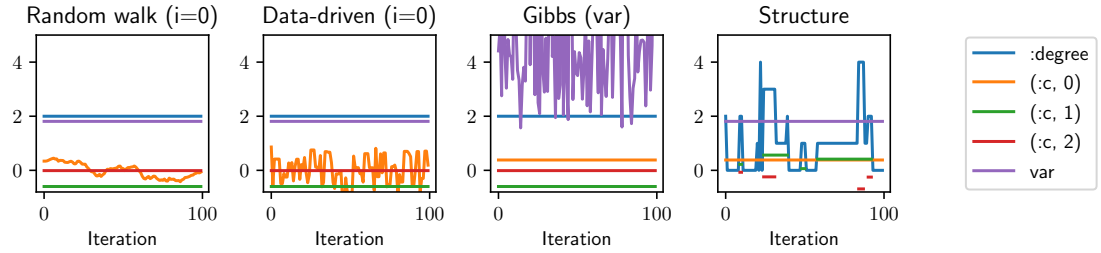


Figure 3-5: Metropolis-Hastings using generative functions as proposal distributions

3.4.3 Hamiltonian Monte Carlo with traces

The Metropolis-Hastings kernel construct of the previous section is highly flexible because the proposal distribution is defined in a probabilistic modeling language. This flexibility allows for specialization of the kernel to the model and in principle improved efficiency, but writing effective specialized proposals requires work and domain knowledge. Hamiltonian Monte Carlo (HMC [92]) is a class of MCMC kernels that can be applied to generic models without as much customization. Gen includes a class of HMC kernels (HMC-KERNEL in Algorithm 6) that can be applied to any model with continuous random variables for which the log density function is differentiable with respect to the values of these variables. The kernel internally uses the following generative function $\mathcal{Q}_{\text{HMC}} :=$ `q_hmc`:

```
@gen function q_hmc(addresses)
    for address in addresses
        {address} ~ normal(0, 1)
    end
end
```

The kernel is parametrized by a set of addresses A. Each $a \in A$ must be a continuous random choice, and the log joint density function $\boldsymbol{\tau} \mapsto \log p(\boldsymbol{\tau})$ must be differentiable with respect to $\boldsymbol{\tau}[a]$. Like MH-KERNEL, HMC-KERNEL uses the UPDATE operation of the trace data type. It also uses the GRADIENTS operation, which computes the necessary partial derivatives of the log density with respect to the value at each address a. This kernel is one example of *selection-based* inference operators that act on traces. Selection-based operators only require the user to specify a *set of addresses* on which to act, and strike a balance between ease-of-use and customizability that favors ease-of-use more than the Metropolis-Hastings construct presented in Section 3.4.2. Other examples of selection-based operators supported by Gen include MAP optimization and elliptical slice sampling [88].

Algorithm 6 Hamiltonian Monte Carlo with address selections and traces

procedure HMC-KERNEL$_{A,k,\epsilon,\boldsymbol{\rho}}(\mathbf{t})$
  assert $|A_{\boldsymbol{\rho}} \cap A| = 0$
  $\boldsymbol{\tau} \leftarrow \mathbf{t}$.CHOICES()
  $(\_, \boldsymbol{\gamma}) \leftarrow \mathbf{t}$.GRADIENT$(A, \mathbf{0})$
  $\mathbf{s} \leftarrow \mathcal{Q}_{\text{HMC}}$.SIMULATE$(A)$ ▷ Sample momenta
  $\boldsymbol{\sigma} \leftarrow \mathbf{s}$.CHOICES()
  $\mathbf{t}_0 \leftarrow \mathbf{t}$
  for $i \leftarrow 1 \ldots k$ **do**
    for $a \in A$ **do**
      $\boldsymbol{\sigma}[a] \leftarrow \boldsymbol{\sigma}[a] + (\epsilon/2)\boldsymbol{\gamma}[a]$ ▷ Half-step on momenta
      $\boldsymbol{\tau}[a] \leftarrow \boldsymbol{\tau}[a] + \epsilon\boldsymbol{\sigma}[a]$ ▷ Full step on positions
    end for
    $(\mathbf{t}_i, \_, \_, \_) \leftarrow \mathbf{t}_{i-1}$.UPDATE$(_, _, \boldsymbol{\tau})$
    $(\_, \boldsymbol{\gamma}) \leftarrow \mathbf{t}_i$.GRADIENT$(A, \mathbf{0})$
    for $a \in A$ **do**
      $\boldsymbol{\sigma}[a] \leftarrow \boldsymbol{\sigma}[a] + (\epsilon/2)\boldsymbol{\gamma}[a]$ ▷ Half-step on momenta
    end for
  end for
  $\boldsymbol{\sigma}' \leftarrow \{a \mapsto -\boldsymbol{\sigma}[a]\}_{a \in A}$ ▷ Negate momenta
  $\mathbf{s}' \leftarrow \mathcal{Q}_{\text{HMC}}$.GENERATE$(A, \boldsymbol{\sigma}')$
  $\alpha \leftarrow \min\{1, \exp(\mathbf{t}_k$.LOGPDF$() - \mathbf{t}_0$.LOGPDF$() - \mathbf{s}$.LOGPDF$() + \mathbf{s}'$.LOGPDF$())\}$
  $r \sim \text{Uniform}(0, 1)$
  if $r \leq \alpha$ **then return** $\mathbf{t}'$ **else return** $\mathbf{t}$
end procedure

3.4.4 A language for composing MCMC kernels

This section defines constructs for composing more sophisticated kernels from simpler primitive kernels like the Metropolis-Hastings kernels with custom proposals and Hamiltonian Monte Carlo kernels defined in the previous two sections. In particular, we describe sufficient conditions for three types of composite kernel to be stationary with respect to the target distribution: (i) conditional application of a kernel, (ii) applying a collection of kernels in sequence, and (iii) applying a kernel randomly chosen from a set of kernels. We prove the validity of these compositions for the setting of a discrete random choices. Although each of these three types of composite kernels can be easily implemented in Julia, we also introduce a *composite kernel DSL* that makes it easier to write stationary kernels by using a restricted syntax and built-in dynamic checks that detect common programming errors that invalidate the sufficient conditions for stationarity for each composition. We show example code in an implementation of the DSL that is embedded in Julia.

Constructing stationary composite kernels from stationary primitive kernels

We now describe a set of compositions of MCMC kernels that are stationary with respect to some target distribution, and give sufficient conditions for the resulting composite kernel to be stationary with respect to the same target distribution.

Sequencing kernels Given two kernels KERN_1 and KERN_2 that are stationary for some target distribution, the kernel k constructed by sequencing the two kernels is itself stationary. This is a standard result in the MCMC literature [5]. The procedure SEQ-KERNEL below shows how to construct a kernel that sequences two kernels using traces.

procedure $\text{SEQ-KERNEL}_{\text{KERN}_1,\text{KERN}_2}(\mathbf{t}_0)$
  $\mathbf{t}_1 \leftarrow \text{KERN}_1(\mathbf{t}_0)$
  $\mathbf{t}_2 \leftarrow \text{KERN}_2(\mathbf{t}_1)$
  return $\mathbf{t}_2$
end procedure

Dependent mixture of kernels Given an indexed collection of kernels $\{\text{KERN}_i\}_{i \in I}$, and a probability distribution on the kernels ($\theta_i \geq 0$ for $\sum_{i \in I} \theta_i = 1$), we can construct a *mixture kernel* that randomly samples an index i according to θ and then applies kernel KERN_i. If the component kernels KERN_i each are stationary with respect to the target distribution, then it is a standard result that the mixture kernel is also stationary with respect to the target distribution [5]. But what if the mixture probabilities change as a function of the current trace? That is $\theta_i(\mathbf{t}) \in [0, 1]$ and $\sum_i \theta_i(\mathbf{t}) = 1$ for each input trace $\mathbf{t}$ of the model such that $\mathbf{t}$.CHOICES() agrees with the observations $\boldsymbol{\rho}$. A composite kernel (DEPENDENT-MIXTURE-KERNEL) that implements this construction is shown below:

procedure $\text{DEPENDENT-MIXTURE-KERNEL}_{\{\text{KERN}_i\},\theta}(\mathbf{t})$
  $i \sim \theta(\mathbf{t})$ ▷ Sample $i \in I$ from discrete distribution $\theta(\mathbf{t})$
  $\mathbf{t}' \leftarrow \text{KERN}_i(\mathbf{t})$
  return $\mathbf{t}'$
end procedure

Although this composite kernel is not in general stationary with respect to the target distribution for stationary KERN_i, the following gives a sufficient condition relating $\{\text{KERN}_i\}_{i \in I}$ and θ that does guarantee stationarity of the composite kernel.

Proposition 3.4.1 (Sufficient condition for stationary dependent mixture kernel)**.** *Let $k_i(\boldsymbol{\upsilon}'; \boldsymbol{\upsilon})$ denote the probability that kernel KERN_i returns a trace with latent state $\boldsymbol{\upsilon}'$ given latent state $\boldsymbol{\upsilon}$. If KERN_i for each $i \in I$ is stationary with respect to a target distribution $p(\cdot|\boldsymbol{\rho})$ and if $\theta_i(\boldsymbol{t}) = \theta_i(\boldsymbol{t}')$ for each $(\boldsymbol{t}, i, \boldsymbol{t}')$ such that $\boldsymbol{t}.\text{CHOICES}() = \boldsymbol{\upsilon} \oplus \boldsymbol{\rho}$ and $\boldsymbol{t}'.\text{CHOICES}() = \boldsymbol{\upsilon}' \oplus \boldsymbol{\rho}$ and $k_i(\boldsymbol{\upsilon}'; \boldsymbol{\upsilon}) > 0$, then $\text{DEPENDENT-MIXTURE-KERNEL}_{\{\text{KERN}_i\}, \theta}$ is stationary with respect to $p(\cdot|\boldsymbol{\rho})$.*

Proof. The dependence of θ_i on $\mathbf{t}$ is simplified to a function of the latent choices only ($\theta_i(\boldsymbol{\upsilon})$) because the other elements of the trace $\mathbf{t}$ are constant. $\text{DEPENDENT-MIXTURE-KERNEL}_{\text{KERN}_i, \theta}$ has distribution:

$$k(\boldsymbol{\upsilon}'; \boldsymbol{\upsilon}) = \sum_{i \in I} \theta_i(\boldsymbol{\upsilon}) k_i(\boldsymbol{\upsilon}'; \boldsymbol{\upsilon})$$

It suffices to show that

$$\sum_{\boldsymbol{\upsilon}: p(\boldsymbol{\upsilon} \oplus \boldsymbol{\rho}) > 0} \sum_{i \in I} \theta_i(\boldsymbol{\upsilon}) k_i(\boldsymbol{\upsilon}'; \boldsymbol{\upsilon}) p(\boldsymbol{\upsilon}|\boldsymbol{\rho}) = p(\boldsymbol{\upsilon}'|\boldsymbol{\rho}) \quad \text{for all } \boldsymbol{\upsilon}' \text{ s.t. } p(\boldsymbol{\upsilon}' \oplus \boldsymbol{\rho}) > 0$$

Switching the order of summation and using the requirement on k and θ to substitute $\theta_i(\boldsymbol{\upsilon}') k_i(\boldsymbol{\upsilon}'; \boldsymbol{\upsilon})$ for $\theta_i(\boldsymbol{\upsilon}) k_i(\boldsymbol{\upsilon}'; \boldsymbol{\upsilon})$ gives:

$$\left(\sum_{i \in I} \theta_i(\boldsymbol{\upsilon}') \right) p(\boldsymbol{\upsilon}'|\boldsymbol{\rho}) = p(\boldsymbol{\upsilon}'|\boldsymbol{\rho}) \quad \text{for all } \boldsymbol{\upsilon}' \text{ s.t. } p(\boldsymbol{\upsilon}'|\boldsymbol{\rho}) > 0$$

□

Note that the requirement that $\theta_i(\boldsymbol{\upsilon}) = \theta_i(\boldsymbol{\upsilon}')$ when $k_i(\boldsymbol{\upsilon}'; \boldsymbol{\upsilon}) > 0$ can be interpreted as an *invariant* on each of the kernels k_i: For each kernel k_i for $i \in I$, and for any input $\boldsymbol{\upsilon}$ such that $p(\boldsymbol{\upsilon} \oplus \boldsymbol{\rho}) > 0$ and any execution of k_i resulting in output trace with latent choices $\boldsymbol{\upsilon}'$, the mixture probability must be unchanged ($\theta_i(\boldsymbol{\upsilon}) = \theta_i(\boldsymbol{\upsilon}')$).

Conditionally applying a kernel The procedure below applies a stationary MCMC kernel KERN only if some predicate $g(\mathbf{t}) \in \{0, 1\}$ of the current model trace holds, and otherwise deterministically returns the previous trace.

```
procedure COND-KERNEL_{KERN,g}(t)
    if g(t) = 1 then
        t' ← KERN(t)
    else
        t' ← t
    end if
    return t'
end procedure
```

This is a special case of a dependent mixture, as follows. For kernel KERN and predicate g, let $\text{KERN}_1 := \text{KERN}$ and let $\text{KERN}_2(\mathbf{t}) := \mathbf{t}$. Let $\theta_1(\mathbf{t}) := g(\mathbf{t})$ and $\theta_2(\mathbf{t}) := 1 - g(\mathbf{t})$. Consider the sufficient condition for stationarity. It holds trivially for KERN_2 because $\mathbf{t}' = \mathbf{t}$. Therefore, the condition reduces to $g(\mathbf{t}) = g(\mathbf{t}')$ for all $\mathbf{t}, \mathbf{t}'$ such that such that there is some nonzero probability that $\mathbf{t}'$ is produced by KERN on input $\mathbf{t}$. That is, the value of the predicate g must be invariant under all executions of the kernel KERN.

Dependent looping over kernels Consider a collection of kernels $\{\text{KERN}_i\}_{i \in I}$ that are stationary with respect to $p(\cdot|\boldsymbol{\rho})$ and a function $r(\mathbf{t}) \in \cup_{n=0}^{\infty} \times_{i=1}^{n} I$. that selects a finite list of kernels $\text{KERN}_{j_1}, \ldots, \text{KERN}_{j_n}$ (where $j_\ell \in I$) given input trace $\mathbf{t}$ of the model (r stands for 'range'). Consider the class of composite kernels that compute the kernel sequence, and then applies each kernel in turn:

procedure DEPENDENT-LOOP-KERNEL$_{\text{KERN},r}(\mathbf{t})$
  $(j_1, \ldots j_n) \leftarrow r(\mathbf{t})$ ▷ Compute kernel sequence
  $\mathbf{t}_0 \leftarrow \mathbf{t}$
  for $i \in 1 \ldots n$ **do**
    $\mathbf{t}_i \leftarrow \text{KERN}_{j_i}(\mathbf{t}_{i-1})$
  end for
  return $\mathbf{t}_n$
end procedure

Under what conditions will DEPENDENT-LOOP-KERNEL$_{\text{KERN},r}$ be stationary with respect to $p_{|s}$? It s equivalent to the composition of DEPENDENT-MIXTURE-KERNEL with SEQ-KERNEL as follows. For each value of $R := (j_1, \ldots, j_n)$, denote the kernel composed by sequencing $\text{KERN}_{j_1}, \ldots, \text{KERN}_{j_n}$ by KERN_R. Then, KERN_R is stationary with respect to $p(\cdot|\boldsymbol{\rho})$ because sequencing preserves stationarity. Then, consider a set of (degenerate) probability distributions on elements R, parametrized by $\mathbf{t}$, given by $\theta_R(\mathbf{t}) := [r(\mathbf{t}) = R]$. Then, DEPENDENT-LOOP-KERNEL$_{\text{KERN},r}$ is equivalent to DEPENDENT-MIXTURE-KERNEL$_{\{\text{KERN}_R\},\theta}$ (where the indexed family of kernels is indexed by R). Therefore, a sufficient condition for stationarity is that $r(\mathbf{t}) = r(\mathbf{t}')$ for all $\mathbf{t}, \mathbf{t}'$ such that $\mathbf{t}'$ is produced from KERN_R with nonzero probability, for all R. This is equivalent to the invariant that $r(\mathbf{t})$ cannot change under any execution of the kernel sequence $\text{KERN}_{j_1}, \ldots, \text{KERN}_{j_n}$, beginning from any input trace of the model $\mathbf{t}$.

A domain specific inference language for composing stationary MCMC kernels

Recall that users of Gen can construct primitive Metropolis-Hastings MCMC kernels using `Gen.mh`, which implements the Metropolis-Hastings construct of Section 3.4.2. Since user inference code in Gen is regular Julia code, users can easily compose these primitive kernels into more complex kernels by implementing the types of compositions introduced above in Julia code. For example, SEQ-KERNEL can be implemented simply by sequencing Julia statements, DEPENDENT-MIXTURE can be implemented by making random choices and applying a different kernel depending on these choices, and COND-KERNEL and DEPENDENT-LOOP-KERNEL can be implemented using Julia's control branching and looping constructs.

However it can be difficult to reason about the stationarity of general kernels composed this way in Julia, because Julia's flexibility makes it easy to accidentally write kernels that are not stationary with respect to the desired target distribution.

To address this issue, Gen includes a domain specific language (DSL) for composing stationary MCMC kernels called the *Composite Kernel DSL*. The DSL, which is embedded in Julia, allows for safer composition of MCMC kernels, by using a restricted syntax that syntactically eliminates the possibility for some types of errors that can lead to unsoundness. As described previously, sufficient conditions for the soundness of various MCMC kernel compositions reduce to *invariants*. Although the DSL does not statically verify that these invariants hold, the DSL compiler generates code for dynamic checks for these invariants. The dynamic checks can be enabled and disabled as needed (e.g. enabled during testing and prototyping and disabled during deployment for higher performance).

The DSL uses syntax similar to Julia function definition syntax, but with a `@kern` macro in front of the function definition expression, and with a distinct set of language constructs that can be used in the function body. The first argument to the function represents the trace of the model to which the constituent kernels will be applied. The set of supported language constructs are:

- **Applying a stationary kernel**. To apply a kernel, the syntax `trace ~ k(trace, args..)` is used. `k` must be a kernel constructed using the composite kernel DSL, a Gen primitive kernel (e.g. `Gen.mh`), or a Julia function that has been declared as stationary by the user.

- **For loops**. For loops are based on regular Julia for loops. The range of the loop (which in general may be a function of the trace) must be invariant under all possible executions of the body of the for loop. A dynamic check for this invariant is automatically inserted by the DSL compiler.

- **If-end expressions**. If-end expressions are based on Julia if-end expressions. The branching condition may be a deterministic function of the trace, but it also must also remain true under all possible executions of the body of the true branch. A dynamic check for this invariant is automatically inserted by the DSL compiler.

- **Deterministic pure let expressions**. It is possible to bind a value to a variable using `let` but the expression on the right-hand-side must be deterministic function of its free variables, its value must be invariant under all possible executions of the body. A dynamic check for this invariant is automatically inserted by the DSL compiler.

- **Stochastic pure let expressions**. It is possible to make random choices, using the syntax `let x ~ dist(args..) .. end`. The expression on the right-hand-side of the '~' must be the application of a Gen probability distribution to arguments, and the choice of distribution and all of its arguments must be invariant under all possible executions of the body of the let expression. A dynamic check for this invariant is automatically inserted by the DSL compiler.

Julia language features that do not have analogues include while loops, reassignment, and mutation. Arbitrary deterministic pure Julia expressions may appear in (i) the arguments

to stationary kernel application, (ii) the range expression for for loops, (iii) the branching condition for if-end expressions, (iv) the right-hand-side of deterministic let expressions, (v) the arguments to distributions on the right-hand-side of stochastic let expressions.

Example: Composing MCMC kernels for Bayesian polynomial regression Recall the model $\mathcal{P}$:= `poly_model`, and the Metropolis-Hastings proposals constructed for use with this model in Section 3.4.2. No one of these kernels was individually sufficient for accurate inference. The code below, written in Gen's Composite Kernel DSL, constructs an MCMC kernel called '`composite_kernel`' from these individual Metropolis-hastings kernels.

```
@kern function composite_kernel(trace, xs, ys)

  for i in 0:trace[:degree]
    trace ~ Gen.mh(trace, q_mh_random_walk, (i,))
  end

  let do_structure ~ bernoulli(0.1)
    if do_structure
        trace ~ Gen.mh(trace, q_mh_structure, ())
    end
  end

  for i in 0:trace[:degree]
      trace ~ Gen.mh(trace, q_mh_data_driven, (xs, ys, i))
  end

  trace ~ Gen.mh(trace, q_var_gibbs, (xs, ys))
end
```

The composite kernel can be decomposed into a sequence of four kernels: (i) a loop kernel that loops over the coefficients, and applies a random walk MH kernel to each one in turn; (ii) a mixture kernel that passes through the previous trace with probability 0.9 and otherwise applies a structure-changing MH kernel; (iii) a loop kernel that loops over the coefficients, and applies a data-driven MH kernel to each one in turn; (iv) a primitive MH kernel that is equivalent to a Gibbs-sampling kernel on the variance. Figure 3-6 shows a history of the values of random choices in the model trace, over the course of iterations of this kernel. The values at four addresses are plotted for the first 400 iterations. The effect of each of the constituent MH kernels is visible in this plot: The changes to degree are due to the structure kernel (which is constructed from `q_structure`). The sudden changes to the coefficients are due to the data-driven kernel (which is constructed from `q_data_driven`). The small adjustments to the coefficients are due to the random walk kernels (which are constructed from `q_random_walk`).

3.5 Resample-move particle filtering with traces

Particle filtering [35] is a class of Monte Carlo techniques for inference in probabilistic models where observed data accumulates over time. Each new piece of observed data leads to

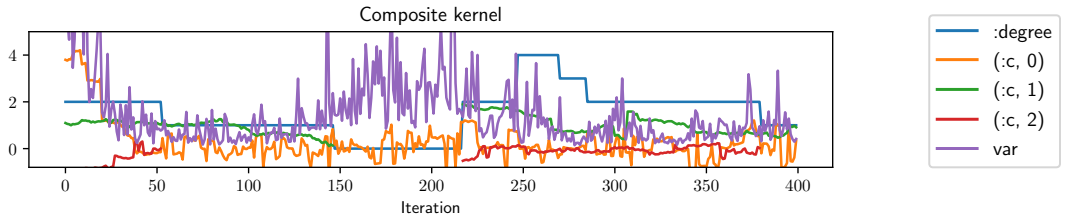


Figure 3-6: Iterates produced by a composite MCMC kernel in a polynomial curve model

a new target distribution. Particle filters maintain a weighted particle approximation to the current target distribution by evolving the collection of particles over time to approximate successive target distributions. Particle filtering has been widely applied in tracking and online parameter estimation in computer vision, robotics, finance, and other fields. A popular variant of particle filtering, called *resample-move* [50], uses a resampling step where some particles are replicated and others are culled in proportion to their weight, and applies 'rejuvenation' MCMC kernels that are stationary with respect to the current target distribution to each particle at each time step. This section shows how to implement resample-move particle filtering using the generative function and trace abstract data types. Each particle is represented by a trace of a generative function that represents the model. Traces are extended from one time step to the next using the UPDATE operation. The MCMC kernels constructed in Section 3.4.2 can be used directly as rejuvenation MCMC kernels, because they are also based on same the trace abstract data type.

The section also shows how generative functions and traces can be used to implement annealed importance sampling (AIS) [91], a gold-standard technique for marginal likelihood estimation in probabilistic models. AIS evolves a single particle over a fine-grained sequence of target distributions that interpolates from a simple distribution for which sampling is trivial to the distribution whose marginal likelihood is required, by gradually changing a parameter like temperature or stochasticity. Like the resample-move particle filter, MCMC moves are applied in between each distribution transition. The marginal likelihood estimate is computed by accumulating incremental importance weights over all time steps. We implement AIS with traces using a construction similar to that used for resample-move particle filtering. UPDATE is used to implement the incremental transition from one target distribution to the next, and the weight returned by UPDATE is precisely the incremental importance weight used in AIS to compute the marginal likelihood estimate.

3.5.1 Trace-based particle filtering with rejuvenation kernels

This section introduces an inference programming construct for implementing resample-move particle filters [50] using the generative function and trace abstract data types. This construct can be used with generative models represented as a generative function $\mathcal{P}$ that takes arguments (j, θ) where j is an integer argument indicating the time step (ranging from 0 to m), and θ represents any other arguments. The other arguments θ are constant

for the purposes of this algorithm, and therefore we omit them from the density notation in this section, using $p(\cdot; j)$ in place of $p(\cdot; (j, \theta))$. Each time step is associated with its own set of latent addresses $S_j \subset A$ and its own set of observed addresses $R_j \subset A$ such that S_{j_1} and S_{j_2} are disjoint and R_{j_1} and R_{j_2} are disjoint for all $j_1 \neq j_2$, and R_{j_1} and S_{j_2} are disjoint for all j_1, j_2. We denote latent choice dictionaries and observed choice dictionaries for time step j by $\boldsymbol{\sigma}_j$ and $\boldsymbol{\rho}_j$ respectively (where $A_{\boldsymbol{\sigma}_j} \subseteq S_j$ and $A_{\boldsymbol{\rho}_j} \subseteq R_j$). The sequence of observed data is an input to the algorithm, and takes the form $\boldsymbol{\rho}_0, \ldots, \boldsymbol{\rho}_m$.

The model generative function must satisfy additional properties that relate the distributions on choice dictionaries $p(\cdot; j)$ for different j: First, every choice dictionary with nonzero density must decompose into latent and observed choice dictionaries. That is:

$$\boldsymbol{\tau} = (\boldsymbol{\sigma}_0 \oplus \cdots \oplus \boldsymbol{\sigma}_j) \oplus (\boldsymbol{\rho}_0 \oplus \cdots \oplus \boldsymbol{\rho}_j) \text{ for all } \boldsymbol{\tau} \text{ s.t. } p(\boldsymbol{\tau}; j) > 0 \text{ for all } j = 0, \ldots, m$$

where $A_{\boldsymbol{\sigma}_k} \subseteq S_k$ and $A_{\boldsymbol{\rho}_k} \subseteq R_k$ for all $k = 0, \ldots, j$. Second, the distributions for each $j - 1$ and j must agree on the marginal density of choice dictionaries.[1] That is, for all $j = 1, \ldots, m$ and all $\boldsymbol{\tau}$ such that $p(\boldsymbol{\tau}; k) > 0$:

$$\bar{p}(\boldsymbol{\sigma}_0 \oplus \cdots \oplus \boldsymbol{\sigma}_{j-1} \oplus \boldsymbol{\rho}_0 \oplus \cdots \oplus \boldsymbol{\rho}_{j-1}; j) = p(\boldsymbol{\sigma}_0 \oplus \cdots \oplus \boldsymbol{\sigma}_{j-1} \oplus \boldsymbol{\rho}_0 \oplus \cdots \oplus \boldsymbol{\rho}_{j-1}; j-1)$$

In addition to $\mathcal{P}$ and the observed data, the construction uses a sequence of generative functions $\mathcal{Q}_0, \ldots, \mathcal{Q}_m$ that encode proposal distributions for each time step. $\mathcal{Q}_0$ is equivalent to a self-normalized importance sampling proposal (Section 3.2). For $j > 0$, $\mathcal{Q}_j$ takes as input a trace $\mathbf{t}_{j-1}$ of $\mathcal{P}$ on arguments $(j - 1, \theta)$. The model and the proposals must satisfy the following requirement:

$$p(\boldsymbol{\sigma}_0 \oplus \cdots \oplus \boldsymbol{\sigma}_j \oplus \boldsymbol{\rho}_0 \oplus \cdots \oplus \boldsymbol{\rho}_j; j) > 0 \implies q_k(\boldsymbol{\sigma}_k; \boldsymbol{\sigma}_0 \oplus \cdots \oplus \boldsymbol{\sigma}_{k-1}) > 0 \; \forall \; k = 1, \ldots, j$$

where $\boldsymbol{\sigma}_0, \ldots, \boldsymbol{\sigma}_{k-1}$ denote the latent choices in the trace $\mathbf{t}_{k-1}$ that is input to $\mathcal{Q}_k$ for each k (the proposal distributions may depend on the observations in the trace as well, but this is omitted from the notation since the observations are treated as a constant).

Given such $\mathcal{P}$, $\boldsymbol{\rho}_0, \ldots, \boldsymbol{\rho}_m$, and $\mathcal{Q}_0, \ldots, \mathcal{Q}_m$, the particle filtering procedure (Algorithm 7) sequentially produces a weighted collection of particles at each time step $j = 0, \ldots, m$, where each particle is represented by a trace $\mathbf{t}_j^{(i)}$ of $\mathcal{P}$ for $i = 1, \ldots, n$. At each time step, each trace is *extended* by first sampling new latent choices $\boldsymbol{\sigma}_j^{(i)} \sim q_j(\cdot; \mathbf{t}_{j-1}^{(i)})$ by invoking SIMULATE on $\mathcal{Q}_j$ and then invoking UPDATE on the previous trace $\mathbf{t}_{j-1}^{(i)}$. The call to UPDATE passes new arguments (j, θ), where $(j - 1, \theta)$ were the arguments for trace $\mathbf{t}_{j-1}^{(i)}$, as well as a *change hint* for these arguments, $\delta_X = (\bot, \top)$, which indicates that the first argument j may have changed, and the second argument θ is not changed. Knowledge that θ has not changed allows an implementation of UPDATE to more efficiently compute its weight, which due to the second requirement on $\mathcal{P}$ above, simplifies and can typically

[1] Technically this requirement can be removed, but it is necessary to recover the asymptotic scaling properties typically associated with particle filtering (linearity in the total number of time steps m).

be computed in a number of operations that is constant, instead of linear, in j:

$$\frac{p(\boldsymbol{\sigma}_0 \oplus \cdots \oplus \boldsymbol{\sigma}_j \oplus \boldsymbol{\rho}_0 \oplus \cdots \oplus \boldsymbol{\rho}_j; j)}{p(\boldsymbol{\sigma}_0 \oplus \cdots \oplus \boldsymbol{\sigma}_{j-1} \oplus \boldsymbol{\rho}_0 \oplus \cdots \oplus \boldsymbol{\rho}_{j-1}; j-1)} = p(\boldsymbol{\sigma}_j, \boldsymbol{\rho}_j | \boldsymbol{\sigma}_0 \oplus \cdots \oplus \boldsymbol{\sigma}_j \oplus \boldsymbol{\rho}_0 \oplus \cdots \oplus \boldsymbol{\rho}_{j-1}; j)$$

The call to UPDATE also passes the new observations $\boldsymbol{\rho}_j$ and the newly sampled latent choices $\boldsymbol{\sigma}_j^{(i)}$, which, together with the accumulated observations $\boldsymbol{\rho}_0 \oplus \cdots \oplus \boldsymbol{\rho}_{j-1}$ and the previous latent choices $\boldsymbol{\upsilon}_{j-1}^{(i)}$, uniquely determine the new trace $\mathbf{t}_j^{(i)}$ via the semantics of UPDATE. Before extending the trace, the particle filter runs a *rejuvenation* MCMC kernel KERN_{j-1} that must be stationary with respect to the target distribution $p(\cdot|\boldsymbol{\rho}_0 \oplus \cdots \oplus \boldsymbol{\rho}_{j-1}; (j-1, \theta))$. This kernel is constructed using the techniques of Section 3.4. Note that the kernel can modify the value of any latent choice, including latent choices that were originally introduced at any time step.

Algorithm 7 Resample-move particle filtering with traces

procedure RESAMPLE($w^{(1)}, \ldots, w^{(n)}$)
  for $i \leftarrow 1, \ldots, n$ **do**
    $b^{(i)} \sim \text{Categorical}(w^{(1)}, \ldots, w^{(n)})$
  end for
  return $[b^{(1)}, \ldots, b^{(n)}]$
end procedure
procedure RESAMPLE-MOVE-PARTICLE-FILTER($\mathcal{P}$, θ, n, $\{(\boldsymbol{\rho}_j, \mathcal{Q}_j)\}_{j=0}^{m}$, $\{\text{KERN}_j\}_{j=0}^{m-1}$)
  for $i \leftarrow 1 \ldots n$ **do**
    $\mathbf{s}_0^{(i)} \leftarrow \mathcal{Q}_0.\text{SIMULATE}(\_)$
    $\mathbf{t}_0^{(i)} \leftarrow \mathcal{P}.\text{GENERATE}((0, \theta), \mathbf{s}_0^{(i)}.\text{CHOICES}() \oplus \boldsymbol{\rho}_0)$
    $\log \tilde{w}^{(i)} \leftarrow \mathbf{t}_0^{(i)}.\text{LOGPDF}() - \mathbf{s}_0^{(i)}.\text{LOGPDF}()$
  end for
  $((w^{(1)}, \ldots, w^{(n)}), \bar{\ell}) \leftarrow \text{NORMALIZE}(\log \tilde{w}^{(1)}, \ldots, \log \tilde{w}^{(n)})$
  $\widehat{z}_0 \leftarrow \bar{\ell}$
  for $j \leftarrow 1 \ldots m$ **do**
    $[b^{(1)}, \ldots, b^{(n)}] \leftarrow \text{RESAMPLE}(w^{(1)}, \ldots, w^{(n)})$
    for $i \leftarrow 1 \ldots n$ **do**
      $\mathbf{t}_{j-1}^{(i)} \leftarrow \text{KERN}_{j-1}(\mathbf{t}_{j-1}^{(b_i)})$
      $\mathbf{s}_j^{(i)} \leftarrow \mathcal{Q}_j.\text{SIMULATE}(\mathbf{t}_{j-1}^{(i)})$
      $(\mathbf{t}_j^{(i)}, \log \tilde{w}^{(i)}, \boldsymbol{\upsilon}, \_) \leftarrow \mathbf{t}_{j-1}^{(i)}.\text{UPDATE}((j, \theta), (\bot, \top), \mathbf{s}_j^{(i)}.\text{CHOICES}() \oplus \boldsymbol{\rho}_j)$
      assert $|A_{\boldsymbol{\upsilon}}| = 0$
    end for
    $((w^{(1)}, \ldots, w^{(n)}), \bar{\ell}) \leftarrow \text{NORMALIZE}(\log \tilde{w}^{(1)}, \ldots, \log \tilde{w}^{(n)})$
    $\widehat{z}_j \leftarrow \widehat{z}_{j-1} + \bar{\ell}$
  end for
  return $\left(\{(\mathbf{t}_m^{(1)}, w^{(1)}), \ldots, (\mathbf{t}_m^{(n)}, w^{(n)})\}, \widehat{z}_m\right)$
end procedure

Example: A resample-move particle particle filter for a state space model Figure 3-7 shows a generative function that describes a state space model $\mathcal{P} = \texttt{bearing\_model}$ adapted from Gilks and Berzuini [50]. $\mathcal{P}$ models the motion of an object in a two-dimensional coordinate system over a sequence of time steps and noisy measurements ($\texttt{z}$) of the bearing of the object, relative to the origin. In this model, the latent addresses and observed addresses at each time step are $S_0 = \{\texttt{x0}, \texttt{y0}, \texttt{vx0}, \texttt{vy0}\}$, $R_0 = \{\texttt{z0}\}$, and $S_j = \{(\texttt{vx}, j), (\texttt{vy}, t)\}$ and $R_j = \{(\texttt{z}, j)\}$ for each $j > 0$. Given a sequence of noisy bearing measurements $\boldsymbol{\rho}_j = \{(\texttt{z}, j) \mapsto z_j\}$ for each time step j, we use Algorithm 7 to infer the trajectory of the object over time. A Julia implementation using the version of Gen at the time of this writing is shown in Figure 3-9a. The code uses functions from Gen's built-in inference library for some of the key operations in Algorithm 7. In particular, `particle_filter_step!` performs the calls to $\mathcal{Q}_j.\textsc{simulate}(\mathbf{t}_{j-1}^{(i)})$ and $\mathbf{t}_{j-1}^{(i)}.\textsc{update}$ for each particle. The proposals $\mathcal{Q}_0 = \texttt{q\_init\_bearing}$ and $\mathcal{Q}_j := \texttt{q\_bearing}$ are defined in Figure 3-8. In addition to the proposals, the implementation also makes use of a family of composite rejuvenation MCMC kernels (`bearing_kernel`, not shown), which apply a sequence of Metropolis-Hastings moves to the latent variables for the five most recent time steps. The value `(Gen.UnknownChange(),)` represents the change hint $(\bot,)$, which indicates that no information is given about the change to the index. There are no other arguments to this model. If the model did take additional arguments, then the tuple would be extended with `Gen.NoChange()`, which represents the change hint $\top$.

```
@gen function bearing_model(num_steps::Int)

    # prior on initial conditions
    x = ({:x0} ~ normal(0.01, 0.01))
    y = ({:y0} ~ normal(0.95, 0.01))
    vx = ({:vx0} ~ normal(0.002, 0.01))
    vy = ({:vy0} ~ normal(-0.013, 0.01))

    # initial bearing measurement
    z0 ~ normal(atan(y, x), 0.005)

    # generate successive states and measurements
    for t in 1:num_steps

        # update the state of the point
        vx = ({(:vx, t)} ~ normal(vx, sqrt(1.0/1e6)))
        vy = ({(:vy, t)} ~ normal(vy, sqrt(1.0/1e6)))
        x += vx; y += vy

        # bearing measurement
        {(:z, t)} ~ normal(atan(y, x), 0.005)
    end
end
```

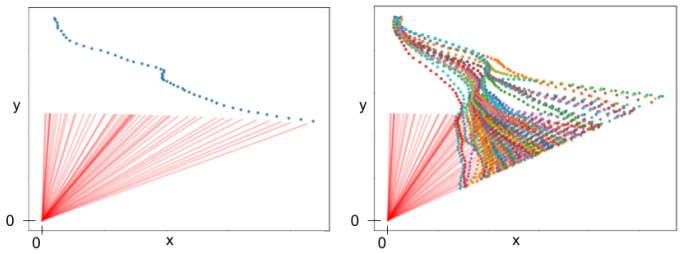
Figure 3-7: A state space model used to illustrate particle filtering with traces

```
@gen function q_init_bearing()
    x0 ~ normal(0.01, 0.01)
    y0 ~ normal(0.95, 0.01)
    vx0 ~ normal(0.002, 0.01)
    vy0 ~ normal(-0.013, 0.01)
end

@gen function q_bearing(trace, t::Int)
    {(:vx, t)} ~ normal(trace[(:vx, t-1)], sqrt(1.0/1e6))
    {(:vy, t)} ~ normal(trace[(:vy, t-1)], sqrt(1.0/1e6))
end
```

Figure 3-8: Generative functions for proposal distributions in particle filtering

```
init_obs = Gen.choicemap((:z0, zs[1]))
state = Gen.initialize_particle_filter(
            model, (0,), init_obs, q_init_bearing, (), n)
for t=1:length(zs)-1

    # apply rejuvenation MCMC moves to each particle in parallel
    Threads.@threads for i in 1:n
        state.traces[i] = bearing_kernel(state.traces[i], max(1, t-5), t-1)
    end

    # resample if the effective sample size is low
    Gen.maybe_resample!(state, ess_threshold=n/2)

    # extend particles to the next time step
    obs = Gen.choicemap(((:z, t), zs[t+1]))
    arg_change = (UnknownChange(),)
    Gen.particle_filter_step!(state, (t,), arg_change, obs, q_bearing, (t,))
end
```

(a) An implementation of a resample-move particle filter in Julia using Gen.



(b) Left: Ground truth trajectory (blue) and observed bearing measurements relative to the origin (red). Right: Trajectories inferred using a resample-move particle filter implemented in Gen.

Figure 3-9: Resample-move particle filtering using generative functions and traces

3.5.2 Annealed importance sampling with traces

A small modification to the particle filtering procedure above recovers a procedure for annealed importance sampling (AIS) [91] using generative functions and traces (Algorithm 8). This AIS procedure applies to models $\mathcal{P}$ that take an argument x for which gradually adjusting x from an initial value x_0 to a final value x_m causes the conditional distribution $p(\cdot|\boldsymbol{\rho}; x_j)$ to evolve gradually from a distribution that is easy to sample from (e.g. unimodal) to the more complex target distribution $p(\cdot|\boldsymbol{\rho}; x_m)$, for some observed data $\boldsymbol{\rho}$. The procedure computes an estimate $\widehat{z}_m$ of the log marginal likelihood $\log \bar{p}(\boldsymbol{\rho}; x_m)$ under the final setting of $x = x_m$. To apply Algorithm 8, $\mathcal{P}$ must be such that incrementing x_j does not change the set of random choices that are sampled. More precisely, we require that:

$$p(\boldsymbol{\tau}; x_{j-1}) > 0 \iff p(\boldsymbol{\tau}; x_j) > 0 \quad \text{for all } j = 1, \ldots, m$$

The procedure repeatedly interleaves an application of an MCMC kernel KERN_{j-1} that must be stationary with respect to $p(\cdot|\boldsymbol{\rho}; x_{j-1})$ with a call to UPDATE, which performs each change of the argument from x_{j-1} to x_j. The procedure passes a change hint δ_X to UPDATE that provides information about the change from x_{j-1} to x_j. If the argument x is complex, but only a single scalar parameter is changed from x_{j-1} to x_j, the information in the change hint may allow UPDATE to use asymptotically fewer operations, depending on how the trace abstract data type for $\mathcal{P}$ is implemented.

Algorithm 8 Annealed importance sampling with traces

procedure ANNEALED-IS($\mathcal{P}$, $\boldsymbol{\rho}$, $\mathcal{Q}$, δ_X, $\{x_j\}_{j=0}^{m}$, $\{\text{KERN}_j\}_{j=0}^{m-1}$)
  $\mathbf{s} \leftarrow \mathcal{Q}.\text{SIMULATE}(\_)$
  $\boldsymbol{\sigma} \leftarrow \mathbf{s}.\text{CHOICES}()$
  $\mathbf{t}_0 \leftarrow \mathcal{P}.\text{GENERATE}(x_0, \boldsymbol{\sigma} \oplus \boldsymbol{\rho})$
  $\log \tilde{w} \leftarrow \mathbf{t}_0.\text{LOGPDF}() - \mathbf{s}_0.\text{LOGPDF}()$
  $\widehat{z}_0 \leftarrow \log \tilde{w}$
  for $j \leftarrow 1 \ldots m$ **do**
    $\mathbf{t}_j \leftarrow \text{KERN}_{j-1}(\mathbf{t}_{j-1})$
    $(\mathbf{t}_j, \log \tilde{w}, \boldsymbol{\upsilon}, \_) \leftarrow \mathbf{t}_j.\text{UPDATE}(x_j, \delta_X, \{\})$
    assert $|A_{\boldsymbol{\upsilon}}| = 0$
    $\widehat{z}_j \leftarrow \widehat{z}_{j-1} + \log \tilde{w}$
  end for
  return $(\mathbf{t}_m, \widehat{z}_m)$
end procedure

3.6 Bridging between models with trace translators

This section defines a new probabilistic programming inference construct called *trace translators* that allows a trace of one generative function to be used as a proposal for another generative function, even when there is no one-to-one correspondence between the sets of traces of the two generative functions, and no correspondence between the addresses in the two generative functions. Trace translators significantly expand the expressiveness of Monte Carlo inference with probabilistic programs. Trace translators are inspired by the general sequential Monte Carlo sampler of Del Moral et al. [33], but are adapted to the probabilistic programming setting, can employ deterministic differentiable transformations, and can be used in settings other than sequential Monte Carlo (a special case is used for MCMC in the next section). This section also shows how a combination of probabilistic programming and differentiable programming techniques can be used to automate the low-level implementation of trace translators.

3.6.1 Trace translators

Consider two generative functions $\mathcal{P}_1$ and $\mathcal{P}_2$ representing two generative models of a domain. Without loss of generality we assume the generative functions take no arguments to simplify notation. Suppose that $\mathcal{P}_1$ and $\mathcal{P}_2$ both use an address universe containing only discrete (counting measure) addresses. Consider observation dictionaries $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ for these two generative models (note that if $\mathcal{P}_1$ and $\mathcal{P}_2$ use the same representation for observed data, then $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ may be the same, but this is not necessary). Let $\mathcal{Z}_1 := \{\boldsymbol{v}_1 : p_1(\boldsymbol{v}_1 \oplus \boldsymbol{\rho}_1) > 0\}$ and $\mathcal{Z}_2 := \{\boldsymbol{v}_2 : p_2(\boldsymbol{v}_2 \oplus \boldsymbol{\rho}_2) > 0\}$. Suppose that there is a bijection h between $\mathcal{Z}_1$ and $\mathcal{Z}_2$. Consider the process where we obtain a trace $\mathbf{t}_1$ of $\mathcal{P}_1$ with $\mathbf{t}_1.\textsc{choices}() = \boldsymbol{v}_1 \oplus \boldsymbol{\rho}_1$ via exact conditional sampling from $\mathcal{P}_1$ conditioned on $\boldsymbol{\rho}_1$ (i.e. $\boldsymbol{v}_1 \sim p_1(\cdot|\boldsymbol{\rho}_1)$), and then compute $\boldsymbol{v}_2 = h(\boldsymbol{v}_1)$, and return $\mathbf{t}_2 = \mathcal{P}_2.\textsc{generate}(\_, \boldsymbol{v}_2 \oplus \boldsymbol{\rho}_2)$. The probability that this process returns some $\mathbf{t}_2$ where $\mathbf{t}_2.\textsc{choices}() = \boldsymbol{v}_2 \oplus \boldsymbol{\rho}_2$ is simply $p_1(h^{-1}(\boldsymbol{v}_2)|\boldsymbol{\rho}_1)$. Therefore, an importance weight if we are targeting the conditional distribution $p_2(\cdot|\boldsymbol{\rho}_2)$, is:

$$\frac{p_2(\boldsymbol{v}_2 \oplus \boldsymbol{\rho}_2)}{p_1(h^{-1}(\boldsymbol{v}_2) \oplus \boldsymbol{\rho}_1)} = \frac{p_2(h(\boldsymbol{v}_1) \oplus \boldsymbol{\rho}_2)}{p_1(\boldsymbol{v}_1 \oplus \boldsymbol{\rho}_1)} \tag{3.10}$$

If $\boldsymbol{\rho}_1 := \{\}$ then by repeatedly sampling $\boldsymbol{v}_1 \sim p_1(\cdot|\boldsymbol{\rho}) = p_1$ and weighing samples according to the above weight, we can construct a self-normalized importance sampling algorithm where $\mathcal{P}_1$ plays the role of the proposal. This algorithm is more flexible than Algorithm 3 because this procedure $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ need not share the same addresses, whereas Algorithm 3 assumes that the model and proposal sample at the same latent addresses (and therefore, the same latent representation). Trace translators generalize this idea along two additional dimensions—permitting continuous addresses to be used, and permitting extension of one or both state spaces with auxiliary random variables (which is necessary when there is no one-to-one correspondence between the two sets $\mathcal{Z}_1$ and $\mathcal{Z}_2$).

Consider an address universe with discrete addresses D (μ_a is the counting measure for $a \in D$) and continuous addresses C (μ_a is the Lebesgue measure on $\mathbb{R}^{n_a}$ for $a \in C$), with $A = D \cup C$. To extend to handle continuous addresses in $\mathcal{P}_1$ and $\mathcal{P}_2$, we require

that there are countable partitions of $\mathcal{Z}_1$ and $\mathcal{Z}_2$ into $\{\mathcal{Z}_{1,e} : e \in E_1\}$ and $\{\mathcal{Z}_{2,e} : e \in E_2\}$ respectively such that if $\boldsymbol{v}, \boldsymbol{v}' \in \mathcal{Z}_{1,e}$ for some e then $\boldsymbol{v}[a] = \boldsymbol{v}'[a]$ for all discrete addresses a, and similarly for each $\mathcal{Z}_{2,e}$. Then, each set $\mathcal{Z}_{1,e}$ and $\mathcal{Z}_{2,e}$ is isomorphic to a subset of a Euclidean space of assignments to continuous addresses. Let $e_1(\boldsymbol{v}) \in E_1$ denote the element of the partition that a dictionary $\boldsymbol{v} \in \mathcal{Z}_1$ belongs to, and similarly for e_2. One example of a valid partition is given by equivalence classes of the following equivalence relation:

$$\boldsymbol{v} \sim \boldsymbol{v}' \iff (A_{\boldsymbol{v}} = A_{\boldsymbol{v}'}) \wedge (\boldsymbol{v}[a] = \boldsymbol{v}'[a] \forall a \in A_{\boldsymbol{v}} \cap D) \tag{3.11}$$

Suppose there exists a bijection $g : E_1 \to E_2$, and a family of continuously differentiable bijections $h_e : \mathcal{Z}_{1,e} \to \mathcal{Z}_{2,g(e)}$. Then, $h : \mathcal{Z}_1 \to \mathcal{Z}_2$ given by $h(\boldsymbol{v}_1) = h_{e_1(\boldsymbol{v}_1)}(\boldsymbol{v}_1)$ is a bijection, with inverse $h^{-1} : \mathcal{Z}_2 \to \mathcal{Z}_1$ given by $h^{-1}(\boldsymbol{v}_2) = h^{-1}_{g^{-1}(e_2(\boldsymbol{v}_2))}(\boldsymbol{v}_2)$. Let $|\boldsymbol{J}_h|(\boldsymbol{v}_1)$ denote the absolute value of the determinant of the Jacobian of $h_{e_1(\boldsymbol{v}_1)}$, evaluated at $\boldsymbol{v}_1$. Then, an importance weight and sampling process for target distribution $p_2(\cdot|\boldsymbol{\rho}_2)$ is:

$$\frac{p_2(h(\boldsymbol{v}_1) \oplus \boldsymbol{\rho}_2)}{p_1(\boldsymbol{v}_1 \oplus \boldsymbol{\rho}_1)} |\boldsymbol{J}_h|(\boldsymbol{v}_1) \quad \text{for} \quad \boldsymbol{v}_1 \sim p_1(\cdot|\boldsymbol{\rho}_1) \tag{3.12}$$

If there is no bijection between $\mathcal{Z}_1$ and $\mathcal{Z}_2$, then we extend the two state spaces in such a way that there is a bijection. This is done using a pair of auxiliary generative functions $\mathcal{Q}_1$ and $\mathcal{Q}_2$, which we use to extend the generative functions $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively. Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be generative functions taking arguments $\boldsymbol{\tau}_1 \in \text{supp}(p_1)$ and $\boldsymbol{\tau}_2 \in \text{supp}(p_2)$ respectively. Without loss of generality, assume that $\mathcal{P}_1$ and $\mathcal{Q}_1$ use disjoints sets of addresses, and similarly for $\mathcal{P}_2$ and $\mathcal{Q}_2$.[2] let $\mathcal{Z}_1 := \{\boldsymbol{v} \oplus \boldsymbol{\sigma} : p_1(\boldsymbol{v} \oplus \boldsymbol{\rho}_1) q_1(\boldsymbol{\sigma}; \boldsymbol{v} \oplus \boldsymbol{\rho}_1) > 0\}$ and $\mathcal{Z}_2 := \{\boldsymbol{v} \oplus \boldsymbol{\sigma} : p_2(\boldsymbol{v} \oplus \boldsymbol{\rho}_2) q_2(\boldsymbol{\sigma}; \boldsymbol{v} \oplus \boldsymbol{\rho}_2) > 0\}$. Suppose that $\mathcal{Z}_1$ and $\mathcal{Z}_2$ can be partitioned as above, with a bijection g and a family of continuously differentiable bijections h_{e_1}. Then, we can construct a proposal process and an importance weight for an unnormalized target density $\tilde{\pi}(\boldsymbol{v}_2 \oplus \boldsymbol{\sigma}_2) := p_2(\boldsymbol{v}_2 \oplus \boldsymbol{\rho}_2) q_2(\boldsymbol{\sigma}_2; \boldsymbol{v}_2 \oplus \boldsymbol{\rho}_2)$ on the extended state space $\mathcal{Z}_2$ of dictionaries of the form $\boldsymbol{v}_2 \oplus \boldsymbol{\sigma}_2$. The importance weight is equivalent to Equation (3.12), but with the unnormalized target density $\tilde{\pi}$ in place of the unnormalized target density $p_2(\boldsymbol{v} \oplus \boldsymbol{\rho})$:

$$\frac{p_2(\boldsymbol{v}_2 \oplus \boldsymbol{\rho}_2) q_2(\boldsymbol{\sigma}_2; \boldsymbol{v}_2 \oplus \boldsymbol{\rho}_2)}{p_1(\boldsymbol{v}_1 \oplus \boldsymbol{\rho}_1) q_1(\boldsymbol{\sigma}_1; \boldsymbol{v}_1 \oplus \boldsymbol{\rho}_1)} |\boldsymbol{J}_h|(\boldsymbol{v}_1 \oplus \boldsymbol{\sigma}_1) \quad \text{for} \quad \begin{array}{l} \boldsymbol{v}_1 \sim p_1(\cdot|\boldsymbol{\rho}_1) \\ \boldsymbol{\sigma}_1|\boldsymbol{v}_1 \sim q_1(\cdot; \boldsymbol{v}_1 \oplus \boldsymbol{\rho}_1) \\ (\boldsymbol{v}_2 \oplus \boldsymbol{\sigma}_2) = h(\boldsymbol{v}_1 \oplus \boldsymbol{\sigma}_1) \end{array} \tag{3.13}$$

Definition 3.6.1 (Trace translator and trace transform)**.** *A* trace translator *is a tuple* $(\mathcal{P}_1, \boldsymbol{\rho}_1, \mathcal{P}_2, \boldsymbol{\rho}_2, \mathcal{Q}_1, \mathcal{Q}_2, h)$ *of elements that satisfy the conditions above. A* trace transform *is a bijection* h *between spaces of choice dictionaries.*

The importance weight in Equation 3.13 is a valid importance weight in the context when $\boldsymbol{v}_1 \sim p_1(\cdot|\boldsymbol{\rho}_1)$, but the same weight computation is useful in other settings when $\boldsymbol{v}_1$ is not necessarily sampled from $p_1(\cdot|\boldsymbol{\rho}_1)$. In particular, Section 3.6.4 gives an algorithm that uses this weight function as an incremental importance weight within a sequential Monte

[2] If the sets of addresses sampled by the two generative functions are not disjoint, they can be made so by adding a different prefix to the addresses of each set.

Carlo scheme, and Section 3.7 gives an MCMC algorithm that uses a special case of this weight function as the basis of an acceptance probability. To support these and other uses, we associate a procedure (TRANSLATE in Algorithm 9) with every trace translator that takes a trace of $\mathcal{P}_1$ as input and returns a trace of $\mathcal{P}_2$ as output (along with a weight). The procedure (i) takes as input a trace $\mathbf{t}_1$ of $\mathcal{P}_1$ with $\mathbf{t}_1.\text{CHOICES}() = \boldsymbol{v}_1 \oplus \boldsymbol{\rho}_1$ for some $\boldsymbol{v}_1$, (ii) samples $\boldsymbol{\sigma}_1 | \boldsymbol{v}_1 \sim q_1(\cdot; \boldsymbol{v} \oplus \boldsymbol{\rho}_1)$, (iii) computes $(\boldsymbol{v}_2 \oplus \boldsymbol{\sigma}_2) = h(\boldsymbol{v}_1 \oplus \boldsymbol{\sigma}_1)$ and (iv) returns a trace $\mathbf{t}_2$ of $\mathcal{P}_2$ with $\mathbf{t}_2.\text{CHOICES}() = \boldsymbol{v}_2 \oplus \boldsymbol{\rho}_2$, along with the importance weight of Equation 3.13.

Algorithm 9 Trace translator procedure

procedure TRANSFORM($\mathbf{t}$, $\boldsymbol{\rho}$, $\mathcal{P}'$, $\boldsymbol{\rho}'$, h, $\boldsymbol{\sigma}$)
  $\boldsymbol{\tau} \leftarrow \mathbf{t}.\text{CHOICES}()$
  $B \leftarrow A^{\mathsf{C}}_{\boldsymbol{\rho}}$
  $\boldsymbol{v} \leftarrow \boldsymbol{\tau}|_B$
  $(\boldsymbol{v}' \oplus \boldsymbol{\sigma}') \leftarrow h(\boldsymbol{\sigma} \oplus \boldsymbol{v})$
  $\mathbf{t}' \leftarrow \mathcal{P}'.\text{GENERATE}(\_, \boldsymbol{v}' \oplus \boldsymbol{\rho}')$
  $I_{\boldsymbol{\sigma}} \leftarrow \{A_{\boldsymbol{\sigma}} \cap C\} \setminus \{a \in A_{\boldsymbol{\sigma}} : (\exists b \in A_{\boldsymbol{\sigma}'}\ \boldsymbol{\sigma}'[b] = \boldsymbol{\sigma}[a]) \vee (\exists b \in A_{\boldsymbol{v}'}\ \boldsymbol{v}'[b] = \boldsymbol{\sigma}[a])\}$
  $I_{\boldsymbol{v}} \leftarrow \{A_{\boldsymbol{v}} \cap C\} \setminus \{a \in A_{\boldsymbol{v}} : (\exists b \in A_{\boldsymbol{\sigma}'}\ \boldsymbol{\sigma}'[b] = \boldsymbol{v}[a]) \vee (\exists b \in A_{\boldsymbol{v}'}\ \boldsymbol{v}'[b] = \boldsymbol{v}[a])\}$
  $O_{\boldsymbol{\sigma}'} = \{A_{\boldsymbol{\sigma}'} \cap C\} \setminus \{b \in A_{\boldsymbol{\sigma}'} : (\exists a \in A_{\boldsymbol{\sigma}}\ \boldsymbol{\sigma}[a] = \boldsymbol{\sigma}'[b]) \vee (\exists a \in A_{\boldsymbol{v}}\ \boldsymbol{v}[a] = \boldsymbol{\sigma}'[b])\}$
  $O_{\boldsymbol{v}'} = \{A_{\boldsymbol{v}'} \cap C\} \setminus \{b \in A_{\boldsymbol{v}'} : (\exists a \in A_{\boldsymbol{\sigma}}\ \boldsymbol{\sigma}[a] = \boldsymbol{v}'[b]) \vee (\exists a \in A_{\boldsymbol{v}}\ \boldsymbol{v}[a] = \boldsymbol{v}'[b])\}$
  $\boldsymbol{J}_{11} \leftarrow \left[\frac{\partial \boldsymbol{\sigma}'[b]}{\partial \boldsymbol{\sigma}[a]}\right]^{b \in O_{\boldsymbol{\sigma}'}}_{a \in I_{\boldsymbol{\sigma}}}$; $\boldsymbol{J}_{12} \leftarrow \left[\frac{\partial \boldsymbol{v}'[b]}{\partial \boldsymbol{\sigma}[a]}\right]^{b \in O_{\boldsymbol{v}'}}_{a \in I_{\boldsymbol{\sigma}}}$; $\boldsymbol{J}_{21} \leftarrow \left[\frac{\partial \boldsymbol{\sigma}'[b]}{\partial \boldsymbol{v}[a]}\right]^{b \in O_{\boldsymbol{\sigma}'}}_{a \in I_{\boldsymbol{v}}}$; $\boldsymbol{J}_{22} \leftarrow \left[\frac{\partial \boldsymbol{v}'[b]}{\partial \boldsymbol{v}[a]}\right]^{b \in O_{\boldsymbol{v}'}}_{a \in I_{\boldsymbol{v}}}$
  $\alpha \leftarrow \mathbf{t}'.\text{LOGPDF}() - \mathbf{t}.\text{LOGPDF}() + \log \left| \begin{bmatrix} \boldsymbol{J}_{11} & \boldsymbol{J}_{12} \\ \boldsymbol{J}_{21} & \boldsymbol{J}_{22} \end{bmatrix} \right|$
  return $(\mathbf{t}', \boldsymbol{\sigma}', \alpha)$
end procedure
procedure TRANSLATE($\mathbf{t}_1$, $\boldsymbol{\rho}_1$, $\mathcal{P}_2$, $\boldsymbol{\rho}_2$, $\mathcal{Q}_1$, $\mathcal{Q}_2$, h)
  $\mathbf{s}_1 \leftarrow \mathcal{Q}_1.\text{SIMULATE}(\mathbf{t}_1)$
  $\boldsymbol{\sigma}_1 \leftarrow \mathbf{s}_1.\text{CHOICES}()$
  $(\mathbf{t}_2, \boldsymbol{\sigma}_2, \alpha) \leftarrow \text{TRANSFORM}(\mathbf{t}_1, \boldsymbol{\rho}_1, \mathcal{P}_2, \boldsymbol{\rho}_2, h, \boldsymbol{\sigma}_1)$
  $\mathbf{s}_2 \leftarrow \mathcal{Q}_2.\text{GENERATE}(\mathbf{t}_2, \boldsymbol{\sigma}_2)$
  $\alpha \leftarrow \alpha + \mathbf{s}_2.\text{LOGPDF}() - \mathbf{s}_1.\text{LOGPDF}()$
  return $(\mathbf{t}_2, \alpha)$
end procedure

The procedure TRANSLATE uses the trace abstract data type operations to sample $\boldsymbol{\sigma}$ and compute the various probability densities used in Equation 3.13. The bijection h and its Jacobian determinant are evaluated by the deterministic subroutine TRANSFORM. We now describe a differentiable programming language for writing the bijections h and an interpreter for that language that uses a general-purpose implementation of TRANSFORM.

3.6.2 Sparsity-aware Jacobian computation

Recall the Jacobian determinant $|\boldsymbol{J}_h|$ in Equation (3.13). While it is possible to compute the entire m-by-m Jacobian matrix first where m is the total number of continuous addresses

in $\boldsymbol{v}_1 \oplus \boldsymbol{\sigma}_1$ ($\boldsymbol{v}_2 \oplus \boldsymbol{\sigma}_2$ also has m continuous addresses), and then compute the absolute value of its determinant, the Jacobian matrix may have sparse structure that we can exploit to reduce unnecessary computation. In many applications of trace translators, the values at some continuous addresses in the input traces are directly *copied* to addresses in the output traces. Copied addresses result in columns in the Jacobian matrix that have a single 1 entry and remaining entries equal to 0. For example, for the function $(u, v, x, y) \mapsto (u, 2u - v, y, x) = (u', v', x', y')$, where the first, third, and fourth elements are copied, the Jacobian is (with columns corresponding to u', v', x', and y'):

$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{matrix} u \\ v \\ x \\ y \end{matrix}$$

Using the cofactor expansion of the determinant, we observe that for any 'copy' column in an m-by-m Jacobian matrix (a column with a single 1 and all other entries 0), the absolute value of the determinant is equivalent to that of the $(m-1)$-by-$(m-1)$ sub-matrix with the corresponding column *and* row omitted (even if that would remove other nonzero entries from the matrix). By applying this rule recursively, we can instead compute the determinant of a much smaller matrix; for the example above with $m = 4$, the absolute value of the determinant simplifies to the absolute value of a single entry ($|-1|$). Indeed, if some input address is copied to some output address, then we can entirely avoid computing its row (and corresponding column) of the Jacobian. Therefore, assuming an LU decomposition is used for the determinant, the number of operations (which is dominated by the determinant), reduces from m^3 to $(m-c)^3$ where c is the number of input addresses that were copied to some output address. The procedure TRANSFORM implements this logic by checking for copied columns and only computing the determinant of the sub-matrix of the Jacobian matrix that excludes these columns (and their corresponding rows). For an efficient implementation we would like that the copied addresses are immediately available without requiring a quadratic scan of all pairs of addresses. The differentiable programming language of Section 3.6.3 uses a language construct to make the set of copied addresses immediately available to the interpreter, and uses automatic differentiation to fully automate the computation of the sub-matrix of the Jacobian that excludes the copied addresses.

3.6.3 A differentiable programming language for trace transforms

Algorithm 9 shows how the densities used in the weight of Equation 3.13 can be computed via the operations of the generative function and trace abstract data types, but it does not describe how to implement the transformation h or how to compute derivatives required for $|\boldsymbol{J}_h|$. This section describes a differentiable programming language in which users specify the transformation h, and an interpreter for this language that evaluates the transformation $((\boldsymbol{v}' \oplus \boldsymbol{\sigma}') = h(\boldsymbol{v} \oplus \boldsymbol{\sigma}))$ and also automatically computes the absolute value of the Jacobian determinant $(|\boldsymbol{J}_h|(\boldsymbol{v} \oplus \boldsymbol{\sigma}))$ using automatic differentiation.

The language extends the Julia language [12] with special syntax for reading from an address in the input $\boldsymbol{v} \oplus \boldsymbol{\sigma} \in \mathcal{Z}_1$, and writing to an address in the output $\boldsymbol{v}' \oplus \boldsymbol{\sigma}' \in \mathcal{Z}_2$. The

logic of the function h is implemented using regular Julia code. The language also includes a construct for copying a value directly from some address in the input to some address in the output. Users read from addresses in the input with the `@read` keyword:

```
value = @read(<address>, <type>)
```

The first argument is the address a and the second argument is a type tag that is either `:disc` or `:cont`, and informs the interpreter whether the random choice at that address is drawn from a discrete or continuous distribution (this information will be used to support automatic differentiation). Recall that $\boldsymbol{v}$ is the choice dictionary of the model generative function $\mathcal{P}_1$, and $\boldsymbol{\sigma}$ is the choice dictionary of the auxiliary generative function $\mathcal{Q}_1$. Each address a therefore needs to specify which of these two choice dictionaries to read from, and the address within that dictionary. The dictionaries are given names in the function signature, has the following syntax:

```
@transform h (model_in, aux_in) to (model_out, aux_out) begin
..
end
```

Here, $\boldsymbol{v}$, $\boldsymbol{\sigma}$, $\boldsymbol{v}'$ and $\boldsymbol{\sigma}'$ have been assigned user-defined names `model_in`, `aux_in`, `model_out`, and `aux_out`, respectively. The inputs also support reading the value of derived quantities including the return values of $\mathcal{P}_1$ and $\mathcal{Q}_1$ (if any), and additional context information like the observations $\boldsymbol{\rho}$, and the arguments to $\mathcal{P}_1$ and $\mathcal{Q}_1$ (if any). Because derived quantities and context information are not part of the input choice dictionaries, but are part of the traces $\mathbf{t}$ and $\mathbf{s}$, we refer to the inputs and outputs as *traces* in the remainder of this section. The syntax for address `a` within the choice dictionary of trace `trace` is `trace[a]`. For example, to read the value of a continuous address `a` from the input model trace $\mathbf{t}$ (that is, either a value $\boldsymbol{v}[\mathtt{a}]$ or $\boldsymbol{\rho}[\mathtt{a}]$):

```
val = @read(model_in[a], :cont)
```

and similarly for reading from the input auxiliary trace $\mathbf{s}$ (that is, a value $\boldsymbol{\sigma}[\mathtt{a}]$):

```
val = @read(aux_in[a], :cont)
```

The syntax for writing to an address in $\boldsymbol{v}' \oplus \boldsymbol{\sigma}' \in \mathcal{Z}_2$ is similar. For example, the expressions for writing a discrete value `val` to $\boldsymbol{v}'[\mathtt{a}]$ and $\boldsymbol{\sigma}'[\mathtt{a}]$ respectively are:

```
@write(model_out[a], val, :disc)
@write(aux_out[a], val, :disc)
```

The inputs $\boldsymbol{v}$ and $\boldsymbol{\sigma}$ can only be read from, and the outputs $\boldsymbol{v}'$ and $\boldsymbol{\sigma}'$ can only be written to. For example, it is not possible to write a value to $\boldsymbol{v}'$ with `@write` and then read the value back with `@read` later.

Often, a user intends to simply copy the value from some address in the input traces to some address in the output traces. While this is possible via a `@read` followed by a `@write`, the language provides a special syntax:

```
@copy(<source-address>, <destination-address>)
```

For example, to copy the value from address `a` in $\boldsymbol{v}$ to address `b` in $\boldsymbol{v}'$, we use:

```
@copy(model_in[a], model_out[b])
```

Of course, it is also possible to copy from $\boldsymbol{v}$ to $\boldsymbol{\sigma}'$, from $\boldsymbol{\sigma}$ to $\boldsymbol{v}'$ and from $\boldsymbol{\sigma}$ to $\boldsymbol{\sigma}'$. It is preferable to use `@copy` when possible instead of reading and then writing, as this informs the interpreter of the special sparse structure in the Jacobian that the copy introduces, as discussed in Section 3.6.2.

The interpreter for the differentiable programming language uses automatic differentiation (AD) to compute the necessary submatrix of the Jacobian as defined in TRANSFORM in Algorithm 9. Forward-mode, reverse-mode, or other approaches to AD can be used, but the current Gen implementation of this language uses forward-mode AD. The interpreter runs the body of the transform function several times. The first run computes the output $\boldsymbol{v}' \oplus \boldsymbol{\sigma}'$ and records any addresses that were copied. There is one run for each input address that was read but not copied, and this run computes the row of the submatrix of the Jacobian corresponding to its input address. Note that only the continuous addresses are involved in the Jacobian calculation, but that the language offers a unified syntax for specifying the action of the bijection h for both the discrete and continuous addresses.

Example: A deterministic change-of-variables trace translator Consider the generative functions given by $\mathcal{P}_1 :=$ `cartesian` and $\mathcal{P}_2 :=$ `polar` for the following code:

```
@gen function cartesian()
    x ~ normal(0, 1)
    y ~ normal(0, 1)
    obsx ~ normal(x, 0.1)
    obsy ~ normal(y, 0.1)
end
```

```
@gen function polar()
    r ~ gamma(1, 1)
    theta ~ uniform(-pi, pi)
    obsr ~ normal(r, 1.)
end
```

Assume that `obsx` and `obsy` are observed for the first model ($A_{\boldsymbol{\rho}_1} = \{\texttt{obsx}, \texttt{obsy}\}$), and that `obsr` is observed for the second model ($A_{\boldsymbol{\rho}_2} = \{\texttt{obsr}\}$). Then the latent spaces for both (the sets of $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ with positive density, respectively) are isomorphic to $\mathbb{R}^2$, but they parametrize $\mathbb{R}^2$ differently (cartesian versus polar coordinates). We can translate between (the latent parts of) traces of these two models using the pair of trace transform programs expressed in the differentiable programming language of this section:

```
@transform h_cart_to_polar (t1) to (t2)
  x = @read(t1[:x], :cont)
  y = @read(t1[:y], :cont)
  r = sqrt(x*x + y*y)
  theta = atan(y, x)
  @write(t2[:r], r, :cont)
  @write(t2[:theta], theta, :cont)
end
```

```
@transform h_polar_to_cart (t2) to (t1)
  r = @read(t2[:r], :cont)
  theta = @read(t2[:theta], :cont)
  x = r * cos(theta)
  y = r * sin(theta)
  @write(t1[:x], x, :cont)
  @write(t1[:y], y, :cont)
end
```

There is no need to extend the spaces with auxiliary choice dictionaries $\boldsymbol{\sigma}$ and $\boldsymbol{\sigma}'$, so $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are both empty generative functions that sample no random choices, and the transforms h and h^{-1} simply read and write to traces of $\mathcal{P}_1$ or $\mathcal{P}_2$.

Example: A coarse-to-fine and discrete-to-continuous trace translator This example describes a trace translator that does employ auxiliary generative functions. Consider the following pair of generative models ($\mathcal{P}_1$:= `disc_model` and $\mathcal{P}_2$:= `cont_model`):

```
@gen function disc_model(start_cell, num_time_steps)
   destination_cell ~ uniform_discrete(1, num_cells)
   observed_cells ~ disc_likelihood_model(start_cell, destination_cell, n)
end

@gen function cont_model(start_loc, n)
   x ~ uniform(0, 1)
   y ~ uniform(0, 1)
   destination_loc = (x, y)
   observed_locs ~ cont_likelihood_model(start_loc, destination_loc, n)
end
```

Both model the observed movement of a person throughout a space over a set of `n` time steps using a combination of their starting location (which is known a-priori) and their destination location (which is a latent variable). $\mathcal{P}_1$ uses a parametrization of the space in terms of a discretization of the space into a set of 20 'cells', identified by integers and $\mathcal{P}_2$ uses a continuous parametrization of the space in terms of (x, y) coordinates (see Figure 3-10). In addition to the different latent representations (address `destination_cell` for $\mathcal{P}_1$ and addresses `x` and `y` for $\mathcal{P}_2$) the two generative functions also use different observed representations: We assume that the likelihood models `disc_likelihood_model` and `cont_likelihood_model` are generative functions that only sample at addresses `1` through `n` for each observed data point (i.e. the likelihood models do not include any latent variables). Each address of the form '`observed_cells => i`' is integer-valued and each address of the form '`observed_locs => i`' takes values in $[0, 1]^2$. (The details of the likelihood models are not shown.) The semantic relationship between these two models is clear (Figure 3-10). To translate a trace of the coarse and discrete model to a trace of the continuous model, recall that we need to translate between the latent parts of these traces

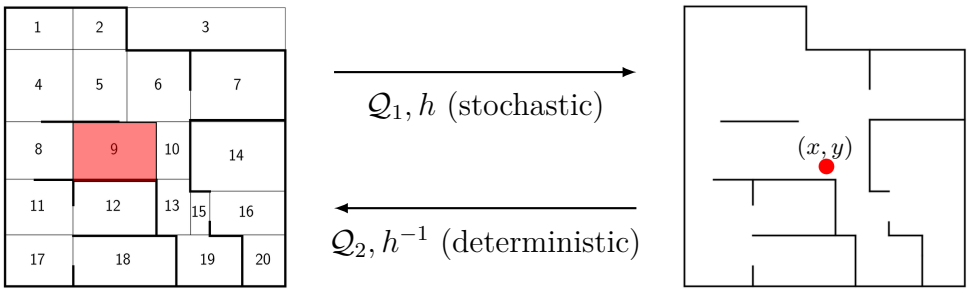


Figure 3-10: Trace translators allow translation between arbitrary latent representations

$\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ (containing addresses {`destination_cell`} and {`x`, `y`} respectively) and not the observed parts $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ (containing addresses {`observed_cells =>` $i : i = 1 \ldots$ `n`} and {`observed_locs =>` $i : i = 1 \ldots$ `n`} respectively). However, the space of all $\boldsymbol{v}_2$ is uncountably infinite (isomorphic to $[0, 1]^2$) and the space of all $\boldsymbol{v}_1$ is finite (containing just 20 elements). Therefore, no bijection is possible, and we generate additional randomness as part of $\mathcal{Q}_1 :=$ `q_disc_to_cont` shown below.

```
@gen function q_disc_to_cont(trace)
    (xmin, xmax, ymin, ymax) = get_bounds(trace[:destination_cell])
    x ~ uniform(xmin, xmax)
    y ~ uniform(ymin, xmax)
end
```

The generative function takes as an argument a trace (`trace`) of the discrete model $\mathcal{P}_1$, reads the value of the destination cell from this trace, and looks up the bounds of the rectangular region that this cell represents. Then, it samples the x and y coordinates of a random point within the region at addresses `x` and `y`. It is now straightforward to write a transform h; it simply copies the continuous destination location from the trace of $\mathcal{Q}_1$ to the output model trace:

```
@transform h_disc_to_cont (model_in, aux_in) to (model_out)
    @copy(aux_in[:x], model_out[:x])
    @copy(aux_in[:y], model_out[:y])
end
```

The backward auxiliary distribution ($\mathcal{Q}_2 :=$ `q_cont_to_disc`) samples no random choices:

```
@gen function q_cont_to_disc(trace)
end
```

Given observed data $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ for the two models, we can then compose a trace translator $(\mathcal{P}_1, \boldsymbol{\rho}_1, \mathcal{P}_2, \boldsymbol{\rho}_2, \mathcal{Q}_1, \mathcal{Q}_2, h)$ from these elements, which is able to translate from traces of the discrete model to traces of the continuous model.

To translate from a trace of the continuous model into a trace of the coarse and discrete model, we can use the same $\mathcal{Q}_1$ and $\mathcal{Q}_2$, but with their roles reversed. For the transform we can use h^{-1}, which computes the destination cell (using `get_cell`) and populates the discrete model trace with this value:

```
@transform function h_cont_to_disc (model_in) to (model_out, aux_out)
    x = @read(model_in[:x], :cont)
    y = @read(model_in[:y], :cont)
    @copy(model_in[:x], aux_out[:x])
    @copy(model_in[:y], aux_out[:y])
    @write(model_out[:destination_cell], get_cell((x, y)), :disc)
end
```

Note that h^{-1} also copies the value of the destination location into the trace of $\mathcal{Q}_1$. Intuitively, the invertibility requirement on h means that neither direction of the bijection can

discard information— information must be preserved either in the new model trace or the new proposal trace. The resulting trace translator is $(\mathcal{P}_2, \boldsymbol{\rho}_2, \mathcal{P}_1, \boldsymbol{\rho}_1, \mathcal{Q}_2, \mathcal{Q}_1, h^{-1})$.

Note that while our pair of trace translators from the discrete to continuous model and vice versa are constructed from the same elements (swapping the role of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ and using h and h^{-1} for the transforms), this is not required—there is an infinite set of valid trace translators from any inference problem $(\mathcal{P}_1, \boldsymbol{\rho}_1)$ to any other problem $(\mathcal{P}_2, \boldsymbol{\rho}_2)$.

3.6.4 Sequential Monte Carlo with trace translators

Sequential Monte Carlo (SMC) samplers [33] are a flexible template for constructing Monte Carlo inference algorithms that consecutively approximate the target distribution for a sequence of inference problems. SMC is most commonly applied using a sequence of inference problems that incrementally extend the state space of one model, but the framework is significantly more general, and can be used with arbitrary sequences of inference problems. For each step in the sequence, the approximation for the previous inference problem is used as a starting point for the next inference problem. This is done by sampling a particle for the new inference problem conditioned on each particle in the approximation for the previous inference problem, and incrementing an importance weight so that the new particle and its weight constitute a 'properly weighted' sample [76]. Trace translators, implemented using Gen's abstract data types, are a general-purpose construct for sampling the new particle given the previous particle, and can be used as the basis of a general SMC procedure, shown in Algorithm 10. The previous particle is represented by the previous trace $(\mathbf{t}_{j-1}^{(i)})$, the new particle is represented by the new trace $(\mathbf{t}_j^{(i)})$, and the log weight returned by TRANSLATE is the incremental importance weight required by SMC. Note that the SMC sampler framework also allows these transitions from one inference problem to the next to be interleaved with application of MCMC kernels (KERN_j) that are stationary for each target distribution $(p_{j-1}(\cdot|\boldsymbol{\rho}_{j-1}))$ in the sequence. Note that SMC construction generalizes particle filtering (Algorithm 7) and annealed importance sampling (Algorithm 8).

While trace translators are very flexible, and the asymptotic properties of SMC hold for any valid trace translator, the efficiency of the resulting SMC algorithm depends on the details of the trace translator. Cusumano-Towner et al. [27] includes a more detailed discussion of the properties of a trace translator that make SMC efficient. Briefly, the distribution on $(\boldsymbol{v}_2, \boldsymbol{\sigma}_2)$ associated with sampling $\boldsymbol{v}_1 \sim p_1(\cdot; \boldsymbol{\rho}_1)$ and $\boldsymbol{\sigma}_2|\boldsymbol{v}_2 \sim q_2(\cdot; \boldsymbol{v}_1)$ and applying $(\boldsymbol{v}_2, \boldsymbol{\sigma}_2) = h(\boldsymbol{v}_1, \boldsymbol{\sigma}_1)$ should be close in Kullback-Leibler (KL) divergence to the distribution associated with sampling $\boldsymbol{v}_2 \sim p_2(\cdot|\boldsymbol{\rho}_2)$ and then $\boldsymbol{\sigma}_2|\boldsymbol{v}_2 \sim q_2(\cdot; \boldsymbol{v}_2)$. Specifically, the following KL divergence, from the distribution induced by p_2 and q_2 to the distribution induced by p_1, q_1, and h, should be small, where $(\boldsymbol{v}_1, \boldsymbol{\sigma}_1) := h^{-1}(\boldsymbol{v}_2 \oplus \boldsymbol{\sigma}_2)$:

$$\int p_2(\boldsymbol{v}_2|\boldsymbol{\rho}_2) q_2(\boldsymbol{\sigma}_2; \boldsymbol{v}_2) \log \left(\frac{p_2(\boldsymbol{v}_2|\boldsymbol{\rho}_2) q_2(\boldsymbol{\sigma}_2; \boldsymbol{v}_2)}{p_1(\boldsymbol{v}_1|\boldsymbol{\rho}_1) q_1(\boldsymbol{\sigma}_1; \boldsymbol{v}_1)} |\boldsymbol{J}_h|(\boldsymbol{v}_1 \oplus \boldsymbol{\sigma}_1) \right) \mu_A^\star(d(\boldsymbol{v}_2 \oplus \boldsymbol{\sigma}_2)) \qquad (3.14)$$

SMC with trace translators is likely to be an effective inference strategy when the initial target distribution (induced by conditioning $\mathcal{P}_0$ on data $\boldsymbol{\rho}$) is trivial or straightforward to sample from efficiently, and the sequence of inference problems and trace translators is such that each instance of Equation 3.14 is small.

Algorithm 10 Sequential Monte Carlo with Trace Translators

procedure TRACE-TRANSLATOR-SMC($\mathcal{Q}_0$, $\{(\mathcal{P}_j, \boldsymbol{\rho}_j)\}_{j=0}^m$, $\{(\mathcal{Q}_{j,1}, \mathcal{Q}_{j,2}, h_j, \text{KERN}_{j-1})\}_{j=1}^m$)
  for $i \leftarrow 1 \ldots n$ **do**
    $\mathbf{s}_0^{(i)} \leftarrow \mathcal{Q}_0.\text{SIMULATE}(\_)$
    $\boldsymbol{\tau} \leftarrow \mathbf{s}_0^{(i)}.\text{CHOICES}() \oplus \boldsymbol{\rho}_0$
    $\mathbf{t}_0^{(i)} \leftarrow \mathcal{P}_0.\text{GENERATE}(\_, \boldsymbol{\tau})$
    $\log \tilde{w}^{(i)} \leftarrow \mathbf{t}_i^{(0)}.\text{LOGPDF}() - \mathbf{s}_i^{(0)}.\text{LOGPDF}()$
  end for
  $((w^{(1)}, \ldots, w^{(n)}), \bar{\ell}) \leftarrow \text{NORMALIZE}(\log \tilde{w}^{(1)}, \ldots, \log \tilde{w}^{(n)})$
  $\beta \leftarrow \bar{\ell}$
  for $j \leftarrow 1 \ldots m$ **do**
    $[b^{(1)}, \ldots, b^{(n)}] \leftarrow \text{RESAMPLE}(w^{(1)}, \ldots, w^{(n)})$
    for $i \leftarrow 1 \ldots n$ **do**
      $\mathbf{t}_{j-1}^{(i)} \leftarrow \text{KERN}_{j-1}(\mathbf{t}_{j-1}^{(b^{(i)})})$
      $\left(\mathbf{t}_j^{(i)}, \log \tilde{w}^{(i)}\right) \leftarrow \text{TRANSLATE}(\mathbf{t}_{j-1}^{(i)}, \boldsymbol{\rho}_{j-1}, \mathcal{P}_j, \boldsymbol{\rho}_j, \mathcal{Q}_{j,1}, \mathcal{Q}_{j,2}, h_j)$
    end for
    $((w^{(1)}, \ldots, w^{(n)}), \bar{\ell}) \leftarrow \text{NORMALIZE}(\log \tilde{w}^{(1)}, \ldots, \log \tilde{w}^{(n)})$
    $\beta \leftarrow \beta + \bar{\ell}$
  end for
  return $\left(\{(\mathbf{t}_m^{(1)}, w^{(1)}), \ldots, (\mathbf{t}_m^{(n)}, w^{(n)})\}, \beta\right)$
end procedure

Example: Coarse-to-fine and discrete-to-continuous SMC Recall the trace translator described above for translating between $\mathcal{P}_1 :=$ `disc_model` and $\mathcal{P}_2 :=$ `cont_model`. Because `disc_model` is a discrete model with a modest-size latent state space, we can perform exact inference in this model conditioned on the observed cells by enumerating over the discrete states `destination_cell`, computing the log joint density for each state, and normalizing. Julia code for this, using the current version of Gen, is given below:

```
function compute_disc_posterior(start_cell::Int, observed_cells::Vector{Int})
    n = length(observed_cells)
    constraints = Gen.choicemap()
    for (i, v) in enumerate(observed_cells)
        constraints[:observed_cells => i] = v
    end
    log_probs = [NaN for destination_cell in 1:20]
    for destination_cell in 1:20
        constraints[:destination_cell] = destination_cell
        (trace, _) = Gen.generate(disc_model, (start_cell, n), constraints)
        log_probs[destination_cell] = Gen.get_score(trace)
    end
    destination_cell_probs = exp.(log_probs .- Gen.logsumexp(log_probs))
    return destination_cell_probs
end
```

We compute this distribution for the observed data (`observed_cells`) and other contextual information (`start_cell`), whose definitions are not shown. Then, we define a generative function `disc_model_proposal` that samples from this distribution:

```
destination_cell_probs = compute_disc_posterior(start_cell, observed_cells)

@gen function disc_model_proposal()
    destination_cell ~ categorical(destination_cell_probs)
end
```

We use trace translator SMC (Algorithm 10) to do inference in the continuous model (`cont_model`) conditioned on data $\{(\texttt{observed\_locs => } i) \mapsto (x_i, y_i)\}_{i=1}^{10}$. We set:

$$
\begin{aligned}
m &:= 1 \\
\mathcal{P}_0 &:= \texttt{disc\_model} \\
\mathcal{Q}_0 &:= \texttt{disc\_model\_proposal} \\
\mathcal{P}_1 &:= \texttt{cont\_model} \\
\boldsymbol{\rho}_1 &:= \{(\texttt{observed\_locs => } i) \mapsto (x_i, y_i)\}_{i=1}^{10} \\
\boldsymbol{\rho}_0 &:= \{(\texttt{observed\_cells => } i) \mapsto \texttt{get\_cell}((x_i, y_i))\}_{i=1}^{10} \\
\mathcal{Q}_{1,1} &:= \texttt{q\_disc\_to\_cont} \\
\mathcal{Q}_{1,2} &:= \texttt{q\_cont\_to\_disc} \\
h_1 &:= \text{ as defined by } \texttt{h\_disc\_to\_cont} \\
\text{KERN}_0 &:= \mathbf{t} \mapsto \mathbf{t}
\end{aligned}
$$

Because $\mathcal{Q}_0 :=$ `disc_model_proposal` samples exactly from the conditional distribution $p_1(\cdot|\boldsymbol{\rho}_1)$, we have satisfied one of our desiderata for an efficient SMC algorithm—the first inference problem in our sequence can be solved efficiently. Since $\mathcal{Q}_0$ samples from the conditional distribution, we do not need to employ an MCMC kernel, so we set $\text{KERN}_1 := \mathbf{t} \to \mathbf{t}$ (i.e. it returns its input). The efficiency of the resulting algorithm depends on the discrete model being an accurate approximation to the continuous model, in the sense of having small Equation (3.14). For this experiment, the likelihood function of the discrete model was trained using maximum likelihood on data simulated from the continuous model. This technique for training a surrogate model for use as an intermediate target distribution SMC is closely related to the technique of training a proposal from Section 3.3.

Figure 3-11 shows estimates of the accuracy of the resulting 'coarse-to-fine' SMC algorithm, as the number of particles is increased, and compares these estimates to those obtained with importance sampling using the prior as the proposal distribution in the continuous model. For each setting of the number of particles (n), accuracy was measured by averaging the log marginal-likelihood estimate over 200 independent runs of the two algorithms with n particles. (The average log marginal likelihood estimate is a common metric of accuracy for importance sampling and sequential Monte Carlo algorithms that is closely related to KL divergence [26]). Also for each n, the median running time of the algorithms over all replicates was measured. These running time and accuracy estimates were then computed for 29 different values of n from $n = 1$ to $n = 2000$, and plotted. Note

that for the SMC algorithm, the distribution `destination_cell_probs` was precomputed instead of computing it within each run of the proposal $\mathcal{Q}_0$. The trace translator SMC algorithm is much more efficient than importance sampling in the continuous model. Cru-

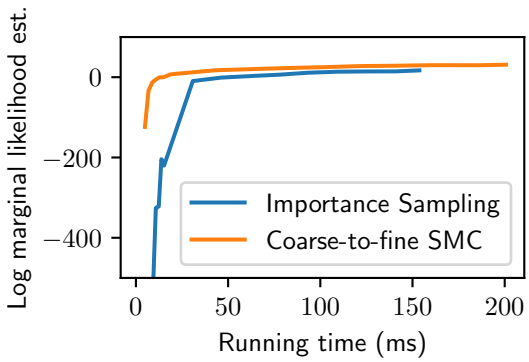


Figure 3-11: Efficiency of coarse-to-fine sequential Monte Carlo using trace translators

cially, the greater efficiency is obtained without requiring us to tailor an MCMC algorithm for the continuous model, which possesses a complex likelihood model that is prone to local minima, making efficient MCMC inference in the continuous model difficult.

3.7 Involutive MCMC

Markov chain Monte Carlo (MCMC) algorithms are powerful tools for approximate sampling from probability distributions, but designing and deriving efficient MCMC algorithms is mathematically involved, and implementing MCMC kernels is tedious and notoriously error-prone. These challenges are especially pronounced when sampling from probability distributions on complex state spaces that combine symbolic, numeric, and structural uncertainty. This section introduces a very general class of MCMC kernels called *involutive MCMC* kernels that can be used to construct efficient structure-changing moves, and a programming construct for implementing involutive MCMC kernels using a combination of generative functions and transforms implemented in the differentiable programming language of Section 3.6.3. This construct is implemented in Gen. The section ends with examples of involutive MCMC kernels implemented using Gen for (i) a split-merge reversible jump [53] move in an infinite mixture model, and (ii) a state-dependent mixture of Metropolis-Hastings proposals on an infinite combinatorial space of covariance functions for a Gaussian process.

3.7.1 Symmetric trace translators

Recall that trace translators are a general construct for generating a trace of one model given a trace of another model while computing an appropriate importance weight that can be used in importance sampling or sequential Monte Carlo. This section discusses a special case of trace translators, called *symmetric trace translators*, that will form the basis for

involutive MCMC kernels. Symmetric trace translators are a type of trace translator from a conditional distribution $p_1(\cdot|\boldsymbol{\rho}_1)$ to *itself*:

Definition 3.7.1 (Symmetric trace translator)**.** *A* symmetric trace translator *is trace translator* $(\mathcal{P}_1, \boldsymbol{\rho}_1, \mathcal{P}_2, \boldsymbol{\rho}_2, \mathcal{Q}_1, \mathcal{Q}_2, h)$ *where* $\mathcal{P}_1 = \mathcal{P}_2$*,* $\boldsymbol{\rho}_1 = \boldsymbol{\rho}_2$*,* $\mathcal{Q}_1 = \mathcal{Q}_2$ *and* h *is an involution (that is,* $h = h^{-1}$*). A symmetric trace translator is denoted by the tuple* $(\mathcal{P}, \boldsymbol{\rho}, \mathcal{Q}, h)$ *with* $\mathcal{P} = \mathcal{P}_1 = \mathcal{P}_2$*,* $\boldsymbol{\rho} = \boldsymbol{\rho}_1 = \boldsymbol{\rho}_2$ *and* $\mathcal{Q} = \mathcal{Q}_1 = \mathcal{Q}_2$*.*

The importance weight (Equation 3.13) for a symmetric trace translator simplifies to:

$$\frac{p(\boldsymbol{v}' \oplus \boldsymbol{\rho})q(\boldsymbol{\sigma}'; \boldsymbol{v}')}{p(\boldsymbol{v} \oplus \boldsymbol{\rho})q(\boldsymbol{\sigma}; \boldsymbol{v})}|\boldsymbol{J}_h|(\boldsymbol{v} \oplus \boldsymbol{\sigma}) \quad \text{for } \boldsymbol{\sigma} \sim q(\cdot; \boldsymbol{v}) \quad \text{and} \quad h(\boldsymbol{v} \oplus \boldsymbol{\sigma}) = (\boldsymbol{v}' \oplus \boldsymbol{\sigma}') \tag{3.15}$$

Below are some simple pedagogical examples of symmetric trace translators. More realistic symmetric trace translators will be defined in the next section.

Example: Identity symmetric trace translator The simplest type of symmetric trace translator has $\mathcal{Q}$ sample no random choices (so that $\boldsymbol{\sigma}_1 = \boldsymbol{\sigma}_2 = \{\}$), and h is simply the identity function ($h(\boldsymbol{v}, \boldsymbol{\sigma}) = (\boldsymbol{v}, \boldsymbol{\sigma})$). The importance weight for this translator is always 1.

Example: A deterministic symmetric trace translator Consider a generative function $\mathcal{P}_1$ that makes a single latent random choice at address `a` taking values in $(0, \infty)$, sampled from a Gamma(1, 1) distribution, and where $\mathcal{Q}$ makes no random choices. Consider the function $h(\boldsymbol{v}, \{\}) = (\{\texttt{a} \mapsto \boldsymbol{v}[\texttt{a}]^{-1}\}, \{\})$. The importance weight for this translator and for $\boldsymbol{v} = \{\texttt{a} \mapsto x\}$ is:

$$\frac{p_{\text{gamma}(1,1)}(x^{-1})}{p_{\text{gamma}(1,1)}(x)} \cdot \frac{1}{x^2}$$

Example: Conditional sample and swap symmetric trace translator Consider the class of symmetric trace translators where $q := p(\cdot|\boldsymbol{\rho})$ and $h(\boldsymbol{v}, \boldsymbol{\sigma}) = (\boldsymbol{\sigma}, \boldsymbol{v})$ (that is, h swaps $\boldsymbol{v}$ and $\boldsymbol{\sigma}$). The importance weight for this translator is always 1. This symmetric trace translator returns a trace that is sampled from the conditional distribution, but such a trace translator is not typically feasible to implement because it requires writing a generative function $\mathcal{Q}$ that samples exactly from the conditional distribution $p(\cdot|\boldsymbol{\rho})$.

Example: Propose and swap symmetric trace translator Consider a trace translator where $h(\boldsymbol{v}, \boldsymbol{\sigma}) = (\boldsymbol{\sigma}, \boldsymbol{v})$, but where $q \neq p(\cdot|\boldsymbol{\rho})$. The importance weight is:

$$\frac{p(\boldsymbol{v}' \oplus \boldsymbol{\rho})q(\boldsymbol{v})}{p(\boldsymbol{v} \oplus \boldsymbol{\rho})q(\boldsymbol{v}')} \quad \text{for } \boldsymbol{v}' \sim q(\cdot) \tag{3.16}$$

This expression is closely related to the acceptance probability in the Metropolis-Hastings algorithm, where $q(\cdot)$ is the proposal distribution and $p(\cdot|\boldsymbol{\rho})$ is the target distribution. As we will see in the next section, this is not a coincidence—symmetric trace translators can be used to construct MCMC kernels that include Metropolis-Hastings kernels and their generalizations including reversible jump MCMC kernels.

3.7.2 Incremental computation for symmetric trace translators

Consider running TRANSLATE (Algorithm 9) for a symmetric trace translator on an input trace $\mathbf{t}$ of $\mathcal{P}$ with $\mathbf{t}.\text{CHOICES}() = \boldsymbol{v} \oplus \boldsymbol{\rho}$, sampling auxiliary trace $\mathbf{s}$ of $\mathcal{Q}$ with $\mathbf{s}.\text{CHOICES}() = \boldsymbol{\sigma}$, and applying the involution ($h(\boldsymbol{v} \oplus \boldsymbol{\sigma}) = (\boldsymbol{v}' \oplus \boldsymbol{\sigma}')$). Let $n_1 := |A_{\boldsymbol{v}}|$, $n_2 := |A_{\boldsymbol{\sigma}}|$, $n_1' := |A_{\boldsymbol{v}'}|$ and $n_2' := |A_{\boldsymbol{\sigma}'}|$, and $n := n_1 + n_2$ and $n' := n_1' + n_2'$. Suppose that $n_1 + n_1' \gg n_2 + n_2'$. Assuming the involution h is specified using the differentiable programming language of Section 3.6.3, TRANSLATE runs the program for h and writes the value for each element of $\boldsymbol{v}' \oplus \boldsymbol{\sigma}'$, requiring $n' = n_1' + n_2' \approx n_1'$ `@write` or `@copy` operations. It also uses approximately n_1' operations to evaluate the log-densities required for the importance weight, because running GENERATE on $\boldsymbol{v}'$ must visit each random choice in $\boldsymbol{v}'$ (we assume the log-density $\mathbf{t}.\text{LOGPDF}()$ was already computed and cached inside $\mathbf{t}$).

If the involution h has sparse structure and the model density p has conditional independence structure, it is possible to improve the asymptotic efficiency of trace translation for a symmetric trace translator to become sub-linear or even constant in $n_1 + n_1'$. The key idea is to modify the program specifying h so that it only specifies the subset of the addresses a in the output $\boldsymbol{v}'$ for which $\boldsymbol{v}'[a] \neq \boldsymbol{v}[a]$ or for which $a \notin A_{\boldsymbol{v}}$, and use $\mathbf{t}.\text{UPDATE}$ (Section 2.3) to efficiently generate the new trace $\mathbf{t}'$ and compute the density ratio $p(\boldsymbol{v}' \oplus \boldsymbol{\rho})/p(\boldsymbol{v} \oplus \boldsymbol{\rho})$ that is required for the importance weight. Let $\tilde{\boldsymbol{v}}'$ denote the restriction of $\boldsymbol{v}'$ to the set of addresses that are either not in $\boldsymbol{v}$ or are in $\boldsymbol{v}$ but have different values, and let $\tilde{h}$ denote the function that is equivalent to h but maps $\boldsymbol{v} \oplus \boldsymbol{\sigma}$ to $\tilde{\boldsymbol{v}}' \oplus \boldsymbol{\sigma}$ instead of $\boldsymbol{v}' \oplus \boldsymbol{\sigma}$. Then, assuming p is a structured density, $\mathbf{t}.\text{UPDATE}(\_, \_, \tilde{\boldsymbol{v}}')$ is well-defined, and results in the same trace $\mathbf{t}'$ that would be returned by the call to $\mathcal{P}.\text{GENERATE}$ in TRANSLATE. But unlike the call to $\mathcal{P}.\text{GENERATE}$, $\mathbf{t}.\text{UPDATE}$ is able to exploit cancellation in the density ratio $p(\boldsymbol{v}' \oplus \boldsymbol{\rho})/p(\boldsymbol{v} \oplus \boldsymbol{\rho})$ due to conditional independence in p, and can use efficient functional data structures to produce $\mathbf{t}'$ from $\mathbf{t}$ in logarithmic or effectively constant time. Algorithm 11 shows a specialized procedure for symmetric trace translators that makes use of UPDATE.

3.7.3 Involutive MCMC

Consider a model generative function $\mathcal{P}$ with density p and observed data $\boldsymbol{\rho}$ for which $\bar{p}(\boldsymbol{\rho}) > 0$. Every symmetric trace translator $(\mathcal{P}, \boldsymbol{\rho}, \mathcal{Q}, h)$ for some $\mathcal{Q}$ and h defines an *involutive MCMC kernel* shown in Algorithm 12 that is stationary with respect to the target distribution $p(\cdot|\boldsymbol{\rho})$. The kernel takes a trace $\mathbf{t}$ of $\mathcal{P}$ as input where $\mathbf{t}.\text{CHOICES}() = \boldsymbol{v} \oplus \boldsymbol{\rho}$ for some $\boldsymbol{v}$, and returns a new trace $\mathbf{t}'$ of $\mathcal{P}$ where $\mathbf{t}'.\text{CHOICES}() = \boldsymbol{v}' \oplus \boldsymbol{\rho}$ for some $\boldsymbol{v}'$. The kernel is parametrized by $\boldsymbol{\rho}$, $\mathcal{Q}$, and $\tilde{h}$, and depends on the model generative function $\mathcal{P}$ through the input trace $\mathbf{t}$. The kernel is parametrized by $\tilde{h}$ instead of h so that we can use the SYMMETRIC-TRANSLATE procedure to exploit incremental computation, as described above.

Cusumano-Towner et al. [28] give a proof that involutive MCMC kernels are stationary with respect to $p(\cdot|\boldsymbol{\rho})$, which is summarized here: Briefly, Tierney [122] defines a construction for a class of deterministic involutive MCMC kernels based on accepting or rejecting the state proposed via an involution. The construction is shown to satisfy detailed balance (and therefore stationarity) with respect to a target distribution. It is possible to instantiate this construction for the involution h on the space $\{(\boldsymbol{v} \oplus \boldsymbol{\sigma}) \in \mathcal{T}_A^\star : p(\boldsymbol{v}|\boldsymbol{\rho})q(\boldsymbol{\sigma}; \boldsymbol{v}) > 0\}$

Algorithm 11 Symmetric trace translator procedure

procedure SYMMETRIC-TRANSFORM($\mathbf{t}$, $\boldsymbol{\rho}$, $\tilde{h}$, $\boldsymbol{\sigma}$)
  $\boldsymbol{\tau} \leftarrow \mathbf{t}$.CHOICES()
  assert $\boldsymbol{\tau}[a] = \boldsymbol{\rho}[a]$ for all $a \in A_{\boldsymbol{\rho}}$
  $\boldsymbol{\upsilon} \leftarrow \boldsymbol{\tau}|_{A_{\boldsymbol{\rho}}^{\mathsf{c}}}$
  $(\tilde{\boldsymbol{\upsilon}}', \boldsymbol{\sigma}') \leftarrow \tilde{h}(\boldsymbol{\sigma}, \boldsymbol{\upsilon})$
  $(\mathbf{t}', \log w, \tilde{\boldsymbol{\upsilon}}, \_) \leftarrow \mathbf{t}$.UPDATE($\_, \_, \tilde{\boldsymbol{\upsilon}}'$)
  $\boldsymbol{\tau}' \leftarrow \mathbf{t}'$.CHOICES()
  assert $\boldsymbol{\tau}'[a] = \boldsymbol{\rho}[a]$ for all $a \in A_{\boldsymbol{\rho}}$
  $I_{\boldsymbol{\sigma}} \leftarrow \{A_{\boldsymbol{\sigma}} \cap C\} \setminus \{a \in A_{\boldsymbol{\sigma}} : (\exists b \in A_{\boldsymbol{\sigma}'}\ \boldsymbol{\sigma}'[b] = \boldsymbol{\sigma}[a]) \vee (\exists b \in A_{\tilde{\boldsymbol{\upsilon}}'}\ \tilde{\boldsymbol{\upsilon}}'[b] = \boldsymbol{\sigma}[a])\}$
  $I_{\boldsymbol{\upsilon}} \leftarrow \{A_{\tilde{\boldsymbol{\upsilon}}} \cap C\} \setminus \{a \in A_{\boldsymbol{\upsilon}} : (\exists b \in A_{\boldsymbol{\sigma}'}\ \boldsymbol{\sigma}'[b] = \boldsymbol{\upsilon}[a]) \vee (\exists b \in A_{\tilde{\boldsymbol{\upsilon}}'}\ \tilde{\boldsymbol{\upsilon}}'[b] = \boldsymbol{\upsilon}[a])\}$
  $O_{\boldsymbol{\sigma}'} = \{A_{\boldsymbol{\sigma}'} \cap C\} \setminus \{b \in A_{\boldsymbol{\sigma}'} : (\exists a \in A_{\boldsymbol{\sigma}}\ \boldsymbol{\sigma}[a] = \boldsymbol{\sigma}'[b]) \vee (\exists a \in A_{\boldsymbol{\upsilon}}\ \boldsymbol{\upsilon}[a] = \boldsymbol{\sigma}'[b])\}$
  $O_{\tilde{\boldsymbol{\upsilon}}'} = \{A_{\tilde{\boldsymbol{\upsilon}}'} \cap C\} \setminus \{b \in A_{\tilde{\boldsymbol{\upsilon}}'} : (\exists a \in A_{\boldsymbol{\sigma}}\ \boldsymbol{\sigma}[a] = \tilde{\boldsymbol{\upsilon}}'[b]) \vee (\exists a \in A_{\boldsymbol{\upsilon}}\ \boldsymbol{\upsilon}[a] = \tilde{\boldsymbol{\upsilon}}'[b])\}$
  $\boldsymbol{J}_{11} \leftarrow \left[\frac{\partial \boldsymbol{\sigma}'[b]}{\partial \boldsymbol{\sigma}[a]}\right]_{a \in I}^{b \in O}$; $\boldsymbol{J}_{12} \leftarrow \left[\frac{\partial \tilde{\boldsymbol{\upsilon}}'[b]}{\partial \boldsymbol{\sigma}[a]}\right]_{a \in I_{\boldsymbol{\sigma}}}^{b \in O_{\tilde{\boldsymbol{\upsilon}}'}}$; $\boldsymbol{J}_{21} \leftarrow \left[\frac{\partial \boldsymbol{\sigma}'[b]}{\partial \boldsymbol{\upsilon}[a]}\right]_{a \in I_{\boldsymbol{\upsilon}}}^{b \in O_{\boldsymbol{\sigma}'}}$; $\boldsymbol{J}_{22} \leftarrow \left[\frac{\partial \tilde{\boldsymbol{\upsilon}}'[b]}{\partial \boldsymbol{\upsilon}[a]}\right]_{a \in I_{\boldsymbol{\upsilon}}}^{b \in O_{\tilde{\boldsymbol{\upsilon}}'}}$
  return $\left(\mathbf{t}', \boldsymbol{\sigma}', \log w + \log \left|\begin{bmatrix} \boldsymbol{J}_{11} & \boldsymbol{J}_{12} \\ \boldsymbol{J}_{21} & \boldsymbol{J}_{22} \end{bmatrix}\right|\right)$
end procedure
procedure SYMMETRIC-TRANSLATE($\mathbf{t}$, $\boldsymbol{\rho}$, $\mathcal{Q}$, $\tilde{h}$)
  $\mathbf{s} \leftarrow \mathcal{Q}$.SIMULATE($\mathbf{t}$)
  $\boldsymbol{\sigma} \leftarrow \mathbf{s}$.CHOICES()
  $(\mathbf{t}', \boldsymbol{\sigma}', \alpha) \leftarrow$ SYMMETRIC-TRANSFORM($\mathbf{t}, \boldsymbol{\rho}, \tilde{h}, \boldsymbol{\sigma}$)
  $\mathbf{s}' \leftarrow \mathcal{Q}$.GENERATE($\mathbf{t}', \boldsymbol{\sigma}'$)
  $\alpha \leftarrow \alpha + \mathbf{s}'$.LOGPDF() $- \mathbf{s}$.LOGPDF()
  return $(\mathbf{t}', \alpha)$
end procedure

with the target distribution $\pi(\boldsymbol{\upsilon} \oplus \boldsymbol{\sigma}) := p(\boldsymbol{\upsilon}|\boldsymbol{\rho})q(\boldsymbol{\sigma}|\boldsymbol{\upsilon})$ and with acceptance probability where $(\boldsymbol{\upsilon}' \oplus \boldsymbol{\sigma}') = h(\boldsymbol{\upsilon} \oplus \boldsymbol{\sigma})$:

$$\min\left\{1, \frac{p(\boldsymbol{\upsilon}' \oplus \boldsymbol{\rho})q(\boldsymbol{\sigma}'; \boldsymbol{\upsilon}')}{p(\boldsymbol{\upsilon} \oplus \boldsymbol{\rho})q(\boldsymbol{\sigma}; \boldsymbol{\upsilon})}|\boldsymbol{J}_h|(\boldsymbol{\upsilon} \oplus \boldsymbol{\sigma})\right\} \tag{3.17}$$

Stationarity of the kernel INVOLUTIVE-MCMC-KERNEL on the space $\{\boldsymbol{\upsilon} \in \mathcal{T}_A^\star : p(\boldsymbol{\upsilon}|\boldsymbol{\rho}) > 0\}$ then follows from the fact that if $\boldsymbol{\upsilon} \sim p(\cdot|\boldsymbol{\rho})$ then the joint distribution of $\boldsymbol{\upsilon}$ and $\boldsymbol{\sigma}$ resulting from sampling $\boldsymbol{\sigma}|\boldsymbol{\upsilon} \sim q(\cdot|\boldsymbol{\upsilon})$ in SYMMETRIC-TRANSLATE is precisely the stationary distribution π of the deterministic involutive kernel on the joint space. By stationarity of the deterministic involutive kernel, $(\boldsymbol{\upsilon}' \oplus \boldsymbol{\sigma}') \sim \pi$. Marginalizing out $\boldsymbol{\sigma}'$ then gives $\boldsymbol{\upsilon}' \sim p(\cdot|\boldsymbol{\rho})$.

Involutive MCMC is a very flexible family of kernels, due to the flexibility of the symmetric trace translator construction. The next section describe two classes of kernels that are special cases, and show examples of these implemented using probabilistic and differentiable programs in Gen.

Algorithm 12 Involutive MCMC

procedure INVOLUTIVE-MCMC-KERNEL$_{\boldsymbol{\rho},\mathcal{Q},\tilde{h}}(\mathbf{t})$
  $(\mathbf{t}', \alpha) \leftarrow$ SYMMETRIC-TRANSLATE$(\mathbf{t}, \boldsymbol{\rho}, \mathcal{Q}, \tilde{h})$
  $r \sim \text{Uniform}(0, 1)$
  if $r \leq \alpha$ **then return** $\mathbf{t}'$ **else return** $\mathbf{t}$
end procedure

3.7.4 Implementing reversible jump MCMC using involutive MCMC

Reversible jump MCMC [53, 56] is a special case of involutive MCMC, and the implementation of reversible jump MCMC kernels can be simplified using Algorithm 12, which uses probabilistic modeling languages and a differentiable programming language to automate the acceptance probability calculation. We first review reversible jump MCMC, then show how it can be implemented with generative functions and differentiable programs using the involutive MCMC inference construct.

The reversible jump MCMC framework uses a set of 'models' $k \in \mathcal{K}$, and a prior distribution on models $p(k)$. For each model, there is a latent parameter vector $\theta_k \in \mathbb{R}^{n(k)}$ where $n(k)$ is the dimension of model h, and a likelihood function $\ell_k(\theta_k)$ for each k. The latent state x is a pair (k, θ_k) of model and continuous parameter. There is a set of *move types* $\mathcal{M}$. Each move type $m \in \mathcal{M}$ is associated with an unordered pair of models (k_1, k_2) and a dimensionality $d(m)$ such that $d(m) \geq n(k_1)$ and $d(m) \geq n(k_2)$ (zero, one, or more than one move types may be associated with a given pair of models). For each latent state $x = (k, \theta_k)$, there is a probability distribution $q_x(m)$ on move types such that $q_x(m) > 0$ implies that h is one of the models for move type m. For each move type $m \in \mathcal{M}$ between k_1 and k_2 there is a pair of continuously differentiable bijections $g_{m,k_1\to k_2} : \mathbb{R}^{d(m)} \to \mathbb{R}^{d(m)}$ and $g_{m,k_2\to k_1} := g^{-1}_{m,k_1\to k_2}$, and a pair of proposal densities $q_{m,k_1\to k_2}(u_{k_1\to k_2})$ and $q_{m,k_2\to k_1}(u_{k_2\to k_1})$ where $u_{k_1\to k_2} \in \mathbb{R}^{d(m)-n(k_1)}$ and $u_{k_2\to k_1} \in \mathbb{R}^{d(m)-n(k_2)}$. A proposal is made from state $x = (k, \theta_k)$ by (i) sampling a move type $m \sim q_x(\cdot)$, and (ii) sampling continuous variable $u \sim q_{m,k\to k'}(\cdot)$ for (k, k') associated with m, and (iii) computing $(\theta'_h, u') := g_{m,k\to k'}(\theta_k, u)$, and proposing new state $x' = (k', \theta'_k)$. The move is accepted with probability:

$$\min\left\{1, \frac{p(k')}{p(h)}\frac{p_{k'}(\theta_{k'})}{p_k(\theta_k)}\frac{\ell_{k'}(\theta'_k)}{\ell_k(\theta_k)}\frac{q_{x'}(m)}{q_x(m)}\frac{q_{m,k'\to h}(u')}{q_{m,k\to k'}(u)}|\boldsymbol{J}g_{m,h\to k'}|(\theta_k, u)\right\}$$

To encode reversible jump MCMC in our framework, we use a generative function $\mathcal{P}$ that encodes the space of models $\mathcal{K}$, the prior distribution on models, $p(k)$, the per-model priors $p_k(\theta_k)$ and the per-model likelihoods $\ell_k(\theta_k)$. Each choice dictionary $\boldsymbol{\tau}$ with $p(\boldsymbol{\tau}) > 0$ decomposes into $\boldsymbol{\tau} = \boldsymbol{v}_\text{d} \oplus \boldsymbol{v}_\text{c} \oplus \boldsymbol{\rho}$ where $\boldsymbol{v}_\text{d}$ are discrete latent choices, $\boldsymbol{v}_\text{c}$ are continuous latent choices, and $\boldsymbol{\rho}$ are observations. The set $\mathcal{K}$ of all models is encoded by the set of all pairs $\{(A_{\boldsymbol{v}}, \boldsymbol{v}_\text{d}) : \boldsymbol{v}_\text{d} \in \mathcal{T}^\star_D, \exists \boldsymbol{v}_\text{c} \in \mathcal{T}^\star_C\ p(\boldsymbol{v}_\text{d} \oplus \boldsymbol{v}_\text{c} \oplus \boldsymbol{\rho}) > 0\}\}$. That is, the the set of possible trace structures (i.e. control-flow paths through the program encoding $\mathcal{P}$) and latent discrete choices. The per-model continuous parameters θ are encoded as $\boldsymbol{v}_\text{c}$. The

likelihood is encoded as $p(\boldsymbol{\rho}|\boldsymbol{v}_{\mathrm{c}} \oplus \boldsymbol{v}_{\mathrm{d}})$. The auxiliary generative function $\mathcal{Q}$ encodes both the probability distribution on moves types using discrete random choices and possibly stochastic control flow, and the per-move-type probability densities on u using continuous random choices. The involution h factors into an (i) involution g on the set of dictionaries of discrete choices $i = (\boldsymbol{v}_{\mathrm{d}} \oplus \boldsymbol{\sigma}_{\mathrm{d}})$ made by $\mathcal{P}$ or $\mathcal{Q}$ that defines the association between move types $\mathcal{M}$ and the model pairs (k_1, k_2); and (ii) a family of bijections h_i on the space of pairs $\boldsymbol{v}_{\mathrm{c}} \oplus \boldsymbol{\sigma}_{\mathrm{c}}$ of continuous random choices for made by $\mathcal{P}$ or $\mathcal{Q}$, indexed by fixed values of the discrete random choices.

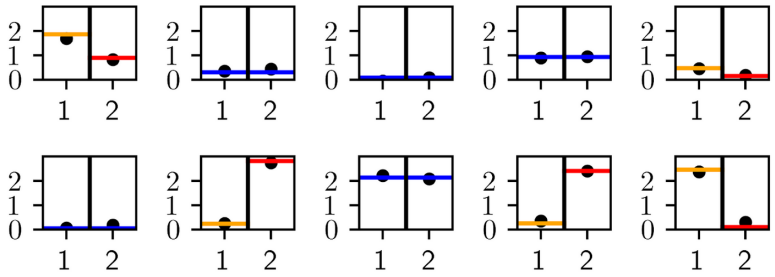
Example: Improving model-switching acceptance rates We now give a simple example of a reversible jump MCMC move encoded using involutive MCMC with a symmetric trace translator $(\mathcal{P}, \boldsymbol{\rho}, \mathcal{Q}, h)$, where $\mathcal{P}$ and $\mathcal{Q}$ are specified using Gen's dynamic modeling language, and where h is specified using Gen's trace transform language. The example motivates the added expressiveness of involutive MCMC over the Metropolis-Hastings construct of Section 3.4.2. The generative model is $\mathcal{P} =$ `branching_model`, and the observed data are $\boldsymbol{\rho} = \{\texttt{y1} \mapsto 1.0, \texttt{y2} \mapsto 1.3\}$:

```
@gen function branching_model()
    if ({:z} ~ bernoulli(0.5))
        m1 ~ gamma(1, 1))
        m2 ~ gamma(1, 1))
    else
        m ~ gamma(1, 1))
        (m1, m2) = (m, m)
    end
    y1 ~ normal(m1, 0.1)
    y2 ~ normal(m2, 0.1)
end
```

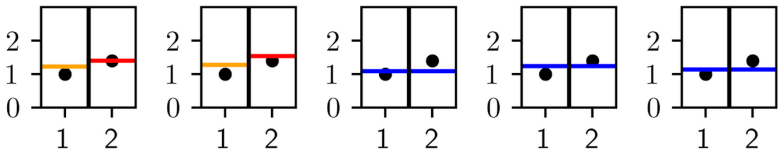
Because this model has stochastic control flow, it represents two distinct structural hypotheses about how observed data are generated: If `z` is true then we enter the first branch, and we hypothesize that the two data points were generated from separate means, sampled at addresses `m1` and `m2`. If `z` is false then we enter the second branch, and we hypothesize that there is a single mean that explains both data points, sampled at address `m`.

Figure 3-12a shows 10 traces of this model sampled using $\mathcal{P}$.SIMULATE, rendered using black dots to represent the two y-values (`y1` on the left and `y2` on the right) and using a blue line to represent a latent sample for which `z` = `F` and red and orange lines to represent a sample fro which `z` = `T`. Figure 3-12b shows approximate posterior samples for given observed data that exhibit uncertainty about the value of `z`.

We want to construct an MCMC kernel that is able to transition between these two distinct structural hypotheses. We could construct a Metropolis-Hastings kernel (Section 3.4.2) $\text{MH-KERNEL}_{\mathcal{Q}_{\text{switch}},\boldsymbol{\rho}}$ with $\mathcal{Q}_{\text{switch}} :=$ `q_mh_switch` that switches between the two branches and proposes new values for each branch; and we could interleave this kernel with another kernel $\text{MH-KERNEL}_{\mathcal{Q}_{\text{walk}},\boldsymbol{\rho}}$ where $\mathcal{Q}_{\text{walk}} :=$ `q_mh_walk` performs a random walk on the values within each branch:



(a) Prior samples from `branching_model`.



(b) Approximate conditional samples from `branching_model` given data $\{\texttt{y1} \mapsto 1.0, \texttt{y2} \mapsto 1.3\}$.

Figure 3-12: Visualization of samples from a model with stochastic control flow

```
@gen function q_mh_switch(trace)
   z ~ bernoulli(trace[:z] ? 0.0 : 1.0)
   if z
      m1 ~ gamma(1, 1)
      m2 ~ gamma(1, 1)
   else
      m ~ gamma(1, 1)
   end
end
```

```
@gen function q_mh_walk(trace)
   if trace[:z]
      m1 ~ normal(trace[:m1], 0.1)
      m2 ~ normal(trace[:m2], 0.1)
   else
      m ~ normal(trace[:m], 0.1)
   end
end
```

Sequencing these two kernels together gives a composite kernel that is stationary with respect to the target distribution $p(\cdot|\boldsymbol{\rho})$ (and ergodic). However, this kernel will not be very efficient, because the branch-switching proposals from $\mathcal{Q}_{\text{switch}}$ are unlikely to be accepted. When switching from the branch with a single mean to the branch with two means, the values of the new addresses `m1` and `m2` are proposed from the prior distribution. This is inefficient, since if we have inferred an accurate value for `m`, we expect the values for `m1` and `m2` to be near this value. The same is true when proposing a structure change in the opposite direction. That means it will take many iterations of the composite kernel to get an accurate estimate of the posterior probability distribution on the two structures.

We would like to use inferred values for `m1` and `m2` to inform our proposal for the value of `m`. For example, we could take the geometric mean (`m = sqrt(m1 * m2)`). However, there are many combinations of `m1` and `m2` that have the same geometric mean. In other words,

the geometric mean is not invertible. However, if we return the additional degree of freedom alongside the geometric mean (`dof`), then we do have an invertible function (`merge_means` with inverse `split_means`):

```
function merge_means(m1, m2)
    m = sqrt(m1 * m2)
    dof = m1 / (m1 + m2)
    (m, dof)
end
```

```
function split_mean(m, dof)
    m1 = m * sqrt((dof / (1 - dof)))
    m2 = m * sqrt(((1 - dof) / dof))
    (m1, m2)
end
```

We use these two functions to construct an involution h, and we use this involution to construct an involutive MCMC kernel that we call a 'split/merge' kernel, because it either splits a parameter value, or merges two parameter values. The auxiliary distribution $\mathcal{Q} :=$ `q_branch_inv` of the symmetric trace translator is responsible for generating the extra degree of freedom when splitting:

```
@gen function q_branch_inv(trace)
    if trace[:z]
        # currently two segments, switch to one
    else
        # currently one segment, switch to two
        {:dof} ~ uniform_continuous(0, 1)
    end
end
```

The transform h, written in Gen's trace transform language, invokes either `merge_means` or `split_means`, depending on the current branch in the input model trace $\mathbf{t}$ (`model_in`):

```
@transform h (model_in, aux_in) to (model_out, aux_out) begin
    if @read(model_in[:z], :disc)
        # currently two segments, switch to one
        @write(model_out[:z], false)
        m1 = @read(model_in[:m1], :cont)
        m2 = @read(model_in[:m2], :cont)
        (m, dof) = merge_means(m1, m2)
        @write(model_out[:m], m, :cont)
        @write(aux_out[:dof], dof, :cont)
    else
        # currently one segment, switch to two
        @write(model_out[:z], true, :disc)
        m = @read(model_in[:m], :cont)
        dof = @read(aux_in[:dof], :cont)
        (m1, m2) = split_mean(m, dof)
        @write(model_out[:m1], m1, :cont)
        @write(model_out[:m2], m2, :cont)
    end
end
```

The Julia code below shows how to construct the involutive MCMC kernel using $\mathcal{Q}$ and h and apply it in a cycle with the random-walk Metropolis-Hastings kernel, using the version of Gen at time of this writing:

```
branch_inv_kernel(trace) = Gen.involutive_mcmc(trace, q_branch_inv, (), h)[1]
random_walk_kernel(trace) = Gen.metropolis_hastings(trace, q_mh_walk, ())[1]
(y1, y2) = (1.0, 1.3)
constraints = Gen.choicemap((:y1, y1), (:y2, y2), (:z, false), (:m, 1.))
trace, = Gen.generate(model, (), constraints)
for iter in 1:100
    trace = branch_inv_kernel(trace)
    trace = random_walk_kernel(trace)
end
```

Figure 3-13 compares the results of this algorithm, which uses the involutive MCMC kernel to switch branches, with results of the algorithm that uses the Metropolis-Hastings kernel to switch branches. The added expressiveness afforded by involutive MCMC allows users to construct efficient reversible jump MCMC kernels that transform the state of random choices in the previous control-flow path into values of random choices in the new control-flow path, leading to higher acceptance rates. Proposals from the Metropolis-Hastings structure-changing kernel are rarely accepted.

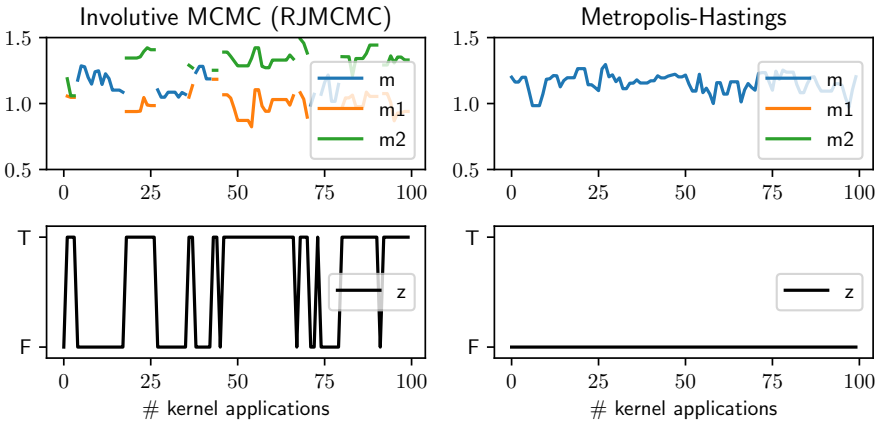


Figure 3-13: Involutive MCMC can express efficient structure-changing moves

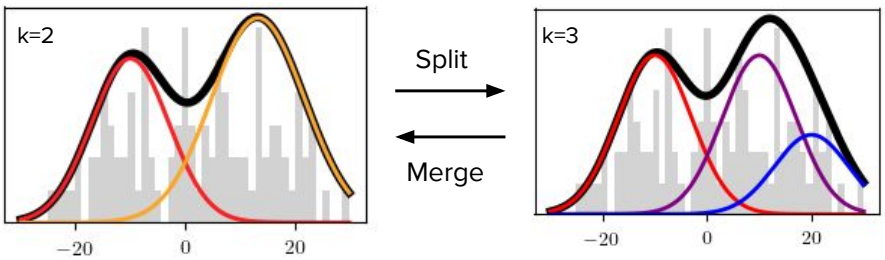


Figure 3-14: Split-merge reversible jump MCMC in an infinite Gaussian mixture model

```
@gen function infinite_mixture(n::Int)
   k ~ poisson_plus_one(1)
   means = [({(:mu, j)} ~ normal(0, 10)) for j in 1:k]
   vars = [({(:var, j)} ~ inv_gamma(1, 10)) for j in 1:k]
   weights ~ dirichlet([2.0 for j in 1:k])
   for i in 1:n
      {(:x, i)} ~ mixture_of_normals(weights, means, vars)
   end
end
```

Figure 3-15: Generative function for an infinite Gaussian mixture model

```
@gen function q_split_merge(trace)
   k = trace[:k] # current number of clusters
   split = (k == 1) ? true : ({:split} ~ bernoulli(0.5))
   if split
      cluster_to_split ~ uniform_discrete(1, k)
      u1 ~ beta(2, 2); u2 ~ beta(2, 2); u3 ~ beta(1, 1)
   else
      cluster_to_merge ~ uniform_discrete(1, k-1)
   end
end
```

Figure 3-16: Auxiliary generative function for a split-merge reversible jump MCMC move

Example: Split-merge reversible jump MCMC in an infinite mixture model One common application of reversible jump MCMC is the construction of 'split-merge' kernels in infinite mixture models [102] (Figure 3-14). Figure 3-15 shows a generative function ($\mathcal{P}$:= `infinite_mixture`) that expresses an infinite univariate Gaussian mixture model, defined using Gen's dynamic modeling language. We construct an involutive MCMC kernel that implements the a split-merge move from an auxiliary generative function ($\mathcal{Q}$:= `q_split_merge`) defined using Gen's dynamic modeling language (Figure 3-16) and an involutive trace transform (h, Figure 3-17) defined using Gen's trace transform language. The model generative function $\mathcal{P}$ takes the number of data points as input, then samples the number of clusters from a Poisson distribution, then samples cluster parameters and mixture proportions, and finally samples the data points from the resulting finite mixture. The auxiliary generative function $\mathcal{Q}$ takes a trace of the model program as input, and randomly decides whether to split a cluster and increase the number of clusters by one or merge two clusters and decrease the number of clusters by one. Then, the program randomly picks which cluster to split, or which clusters to merge.[3] If a split is chosen, then the program also samples the three degrees of freedom necessary to generate the new parameters for the clusters in an invertible manner. Figure 3-18 shows graphically how the involutive trace transform defined in Figure 3-17 acts on pairs of model traces and auxiliary

[3]This kernel always merges the *last* cluster with a random other cluster; for ergodicity the kernel can be composed with a kernel (that is always accepted) that swaps a random cluster with the last cluster.

traces. The yellow section (1) defines an involution g on the discrete random choices that specifies that (i) the `split` choice should be flipped (so that split moves are always mapped to merge moves and vice versa) and that (ii) the number of clusters should be increased by one for a split move and decreased by one for a merge move, and (iii) which merged cluster corresponds to which split clusters. The green section (2) specifies the continuous bijections that govern the transformation of continuous random choices during split moves and the purple section (3) specifies the inverses of these bijections, which govern the transformation of continuous choices during merge moves.

```
@transform h_split_merge (model_in, aux_in) to (model_out, aux_out) begin
   k = @read(model_in[:k], :discrete)
   split = (k == 1) ? true : @read(aux_in[:split], :discrete)
   if split
      cluster_to_split = @read(aux_in[:cluster_to_split], :discrete)
      @write(model_out[:k], k+1, :discrete)
      @copy(aux_in[:cluster_to_split], aux_out[:cluster_to_merge])
      @write(aux_out[:split], false, :discrete)
   else
      cluster_to_merge = @read(aux_in[:cluster_to_merge], :discrete)
      @write(p_out, :k, k-1, :discrete)
      @copy(aux_in[:cluster_to_merge], aux_out[:cluster_to_split])
      if (k > 2) @write(aux_in[:split], true, :discrete) end
   end
   if split
      u1 = @read(aux_in[:u1], :cont)
      u2 = @read(aux_in[:u2], :cont)
      u3 = @read(aux_in[:u3], :cont)
      weights = @read(model_in[:weights], :cont)
      mu = @read(model_in[(:mu, cluster_to_split)], :cont)
      var = @read(model_in[(:var, cluster_to_split)], :cont)
      new_weights = split_weights(weights, cluster_to_split, u1, k)
      w1 = new_weights[cluster_to_split]; w2 = new_weights[k+1]
      (mu1, mu2, var1, var2) = split_params(mu, var, u2, u3, w1, w2)
      @write(model_out[:weights, new_weights, :cont)
      @write(model_out[(:mu, cluster_to_split), mu1, :cont)
      @write(model_out[(:mu, k+1), mu2, :cont)
      @write(model_out[(:var, cluster_to_split), var1, :cont)
      @write(model_out[(:var, k+1), var2, :cont)
   else
      mu1 = @read(model_in[(:mu, cluster_to_merge)], :cont)
      mu2 = @read(model_in[(:mu, k)], :cont)
      var1 = @read(model_in[(:var, cluster_to_merge)], :cont)
      var2 = @read(model_in[(:var, k)], :cont)
      weights = @read(model_in[:weights], :cont)
      w1 = weights[cluster_to_merge]; w2 = weights[k]
      (new_weights, u1) = merge_weights(weights, cluster_to_merge, k)
      w = new_weights[cluster_to_merge]
      (mu, var, u2, u3) = merge_params(mu1, mu2, var1, var2, w1, w2, w)
      @write(model_out[:weights], new_weights, :cont)
      @write(model_out[(:mu, cluster_to_merge)], mu, :cont)
      @write(model_out[(:var, cluster_to_merge)], var, :cont)
      @write(model_out[(:u1, u1)], :cont)
      @write(model_out[(:u2, u2)], :cont)
      @write(model_out[(:u3, u3)], :cont)
   end
end
```

1

2

3

Figure 3-17: Involution trace transform for a split-merge reversible MCMC move

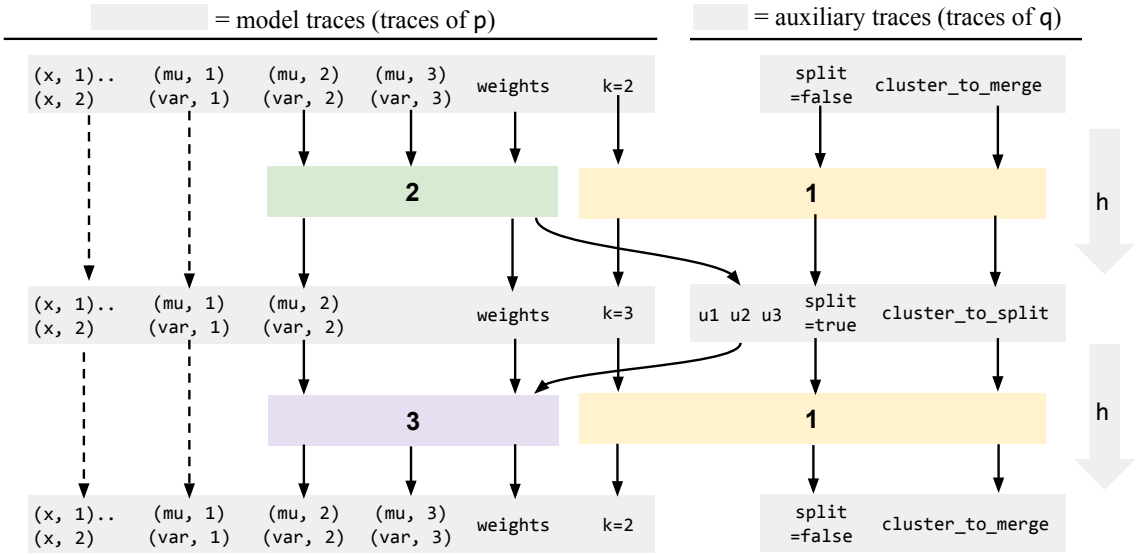


Figure 3-18: Schematic of the involution trace transform for a split-merge MCMC move

3.7.5 State-dependent mixture kernels and involutive MCMC

A common tactic for composing MCMC algorithms is to randomly select an MCMC kernel to apply [121]. As shown in Section 3.4.4, the mixture probabilities cancel out in the Metropolis-Hastings acceptance probability when they are independent of the state (as in e.g. a 0.4 probability of applying one kernel and a 0.6 probability of applying another kernel). But in general, the mixture probabilities may depend on the state. In this case, the mixture probabilities must be accounted for in the acceptance probability. Involutive MCMC allows users to express state-dependent mixture kernels using generative functions and trace transforms, and automates the computation of the acceptance probability for the entire composite mixture kernel, including the mixture probabilities. The pattern for this construction is:

1. The auxiliary generative function $\mathcal{Q}$ is partitioned into two segments with choices $\boldsymbol{\sigma} = \boldsymbol{\sigma}_1 \oplus \boldsymbol{\sigma}_2$. The random choices $\boldsymbol{\sigma}_1$ made in the first segment are all discrete, and represent the mixture distribution. Since $\mathcal{Q}$ takes the model trace $\mathbf{t}$ as input, this distribution depends on the current state of the model. For every possible $\boldsymbol{\sigma}_1$, there is a set of random choices in the model for which new values will be proposed. The random choices $\boldsymbol{\sigma}_2$ in the second segment, which may be discrete and continuous, are the proposed values to these choices in the model.

2. The involution h swaps the previous values of the proposed-to random choices with their new values $\boldsymbol{\sigma}_2$, by swapping data between the model trace and the auxiliary trace. The involution program also copies the random choices $\boldsymbol{\sigma}_1$ that determined what subset of random choices to propose to from the input auxiliary trace to the output auxiliary trace.

Example: Mixture kernel for Bayesian inference of Gaussian process structure Figure 3-19 shows a generative function $\mathcal{P} :=$ `gp_model` written in Gen's dynamic modeling language that defines a generative model of univariate time series data based on a Gaussian process that uses a prior on covariance functions of the Gaussian process that is based on a probabilistic context-free grammar. The prior posits a compositional and combinatorial space of covariance functions that is based on the set of parse trees of a context-free grammar. The generative function $\mathcal{P}$ samples from the prior by invoking the generative function `cov_function_prior`, which is itself recursive. Therefore, traces of $\mathcal{P}$ have a hierarchical address structure that mirrors the structure of the parse trees. One example trace is depicted in Figure 3-19b. We will construct an MCMC kernel that (i) randomly picks a subtree of the parse tree that should be modified, and (ii) randomly samples a new subtree to replace the previous subtree by sampling from `cov_function_prior`. The kernel is a mixture distribution over the set of all nodes in the parse tree, which depends on the current state. Related inference algorithms were previously studied in [113, 109] based on a model of Grosse et al. [54]. We express this kernel using an involutive MCMC construction that includes an auxiliary generative function $\mathcal{Q} :=$ `q_mixture` (Figure 3-20) and the involution h defined by `h_mixture` in Figure 3-21.

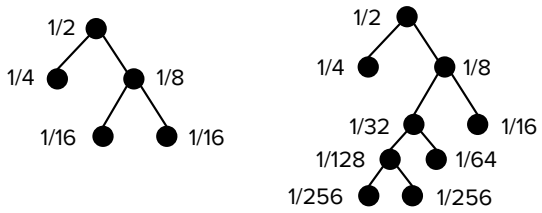
When the resulting involutive MCMC kernel is applied, the auxiliary generative function $\mathcal{Q}$ first picks a random node in the parse tree of the covariance function, by doing a stochastic walk of the existing parse tree that terminates at the chosen node:

```
prev_cov_function = trace[:cov_function]
path ~ walk_tree(prev_cov_function, ..)
```

The code walks the tree using the following recursion, which results in a probability distribution that assigns exponentially lower probability to nodes that are deeper in the tree:

```
if ({:done} ~ bernoulli(0.5))
  return path
elseif ({:recurse_left} ~ bernoulli(0.5))
  path = (path..., :left_node)
  return ({:left} ~ walk_tree(node.left, path))
else
  path = (path..., :right_node)
  return ({:right} ~ walk_tree(node.right, path))
end
```

The resulting distributions on selected nodes for two possible input trees are shown below:



Each node in the tree represents a different proposal that will be applied. The mixture distribution over possible nodes in the tree is state-dependent (and even the *support* of this mixture distribution is state-dependent, because the set of nodes in the tree can change from one state to the next). The rest of the auxiliary generative function $\mathcal{Q}$ proposes a new subtree by sampling from the same process used to recursively define the prior distribution:

```
new_subtree ~ cov_function_prior()
```

The first part of the involution (Figure 3-21, Lines 4-5) swaps the newly proposed random choices for the subtree with the existing choices for that subtree in the model trace:

```
@copy(model_in[subtree_address], aux_out[:new_subtree])
@copy(aux_in[:new_subtree], model_out[subtree_address)
```

The second part of the involution (Line 6) copies the random choices made during this walk from the input auxiliary trace to the output auxiliary trace:

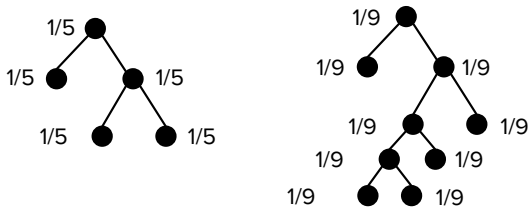
```
@copy(aux_in[:path], aux_out[:path])
```

Note that in both of these instances, `@copy` is being used to copy the entire *set* of random choices from the namespace `:path` in `aux_in` to the namespace `:path` in `aux_out`.

Because the mixture distribution is specified using a probabilistic program, it is straightforward to modify the program `p` to define a different mixture distribution. The code below specifies a mixture distribution that is uniform over all nodes in the tree.

```
n1 = size(node.left); n2 = size(node.right)
if ({:done} ~ bernoulli(1 / (1 + n1 + n2)))
  return path
elseif ({:recurse_left} ~ bernoulli(n1 / (n1+n2))
  path = (path..., :left_node)
  return ({:left} ~ walk_tree(node.left, path))
else
  path = (path..., :right_node)
  return ({:right} ~ walk_tree(node.right, path))
end
```

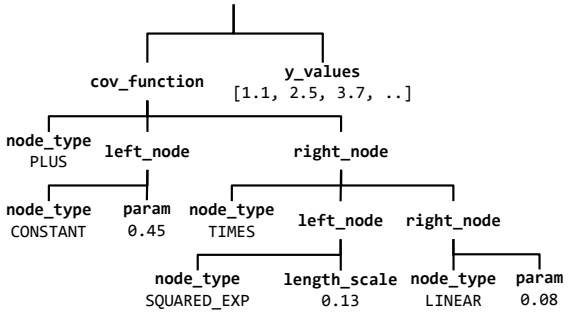
The resulting distributions, for two possible input trees, are:



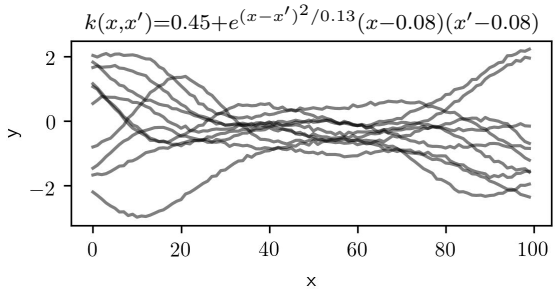
Note that the probability of choosing a given subtree to propose to is itself changed when the subtree changes. Therefore, the mixture probabilities do not in general cancel in the the acceptance probability calculation, and must be accounted for. For the original mixture distribution, the ratio of mixture probabilities is either 1, 0.5, or 2 depending on whether the previous and new subtrees are leaf or internal nodes. For this alternative mixture distribution, the ratio of mixture probabilities is the ratio of sizes of the two trees (e.g. 9/5 or 5/9 for the trees above). In both cases, involutive MCMC (Algorithm 12) automatically computes the acceptance probability.

```
@gen function gp_model(x_values::Vector)
   cov_function ~ cov_function_prior()
   cov_matrix = compute_cov_matrix(cov_function, x_values)
   n = length(xs)
   y_values ~ mvnormal(zeros(n), cov_matrix .+ 0.01*I(n))
end

@gen function cov_function_prior()
   node_type ~ categorical(production_rule_probs))
   if node_type == CONSTANT
      param ~ uniform(0, 1)
      return ConstantNode(param)
   elseif node_type == LINEAR
      param ~ uniform(0, 1)
      return LinearNode(param)
   elseif node_type == SQUARED_EXP
      length_scale ~ uniform(0, 1)
      return SquaredExponentialNode(length_scale)
   elseif node_type == PERIODIC
      ..
   elseif node_type == PLUS
      left_node ~ cov_function_prior()
      right_node ~ cov_function_prior()
      return PlusNode(left_node, right_node)
   elseif node_type == TIMES
      left_node ~ cov_function_prior()
      right_node ~ cov_function_prior()
      return TimesNode(left_node, right_node)
   end
end
```

(a) Generative function encoding the generative model.



(b) A trace of the generative function in (a).



(c) A sampled covariance function and data.

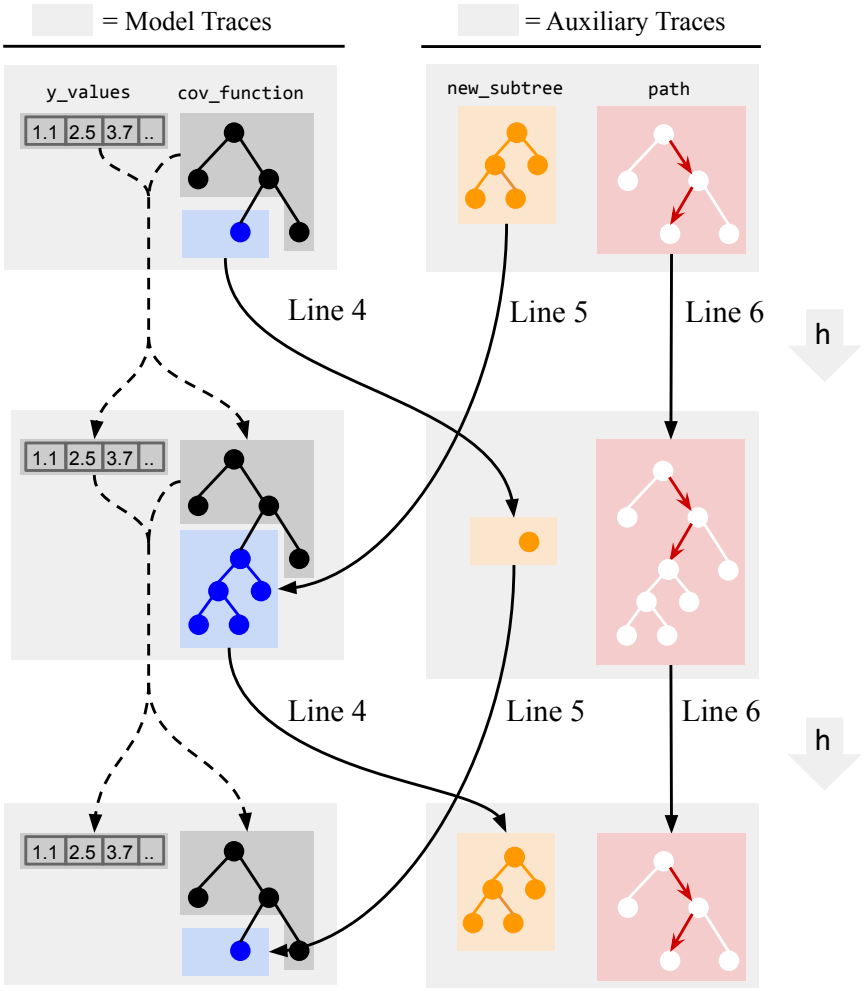
Figure 3-19: A Gaussian process model with a nonparametric prior on covariance functions

```
@gen function q_mixture(trace)
   prev_cov_function = trace[:cov_function]
   path ~ walk_tree(prev_cov_function, (:cov_function,))
   new_subtree ~ cov_function_prior()
   return path
end

@gen function walk_tree(node::Node, path)
   if isa(node, LeafNode)
      done ~ bernoulli(1)
      return path
   elseif ({:done} ~ bernoulli(0.5))
      return path
   elseif ({:recurse_left} ~ bernoulli(0.5))
      path = (path..., :left_node)
      return ({:left} ~ walk_tree(node.left, path))
   else
      path = (path..., :right_node)
      return ({:right} ~ walk_tree(node.right, path))
   end
end
```

Figure 3-20: Auxiliary generative function for a state-dependent mixture MCMC kernel

```
1  @transform h_mixture (model_in, aux_in) to (model_out, aux_out) begin
2    path = @read(aux_in[], :discrete)
3    subtree_address = foldr(=>, path))
4    @copy(model_in[ subtree_address ], aux_out[ :new_subtree ])
5    @copy(aux_in[ :new_subtree ], model_out[ subtree_address ])
6    @copy(aux_in[ :path ], aux_out[ :path ])
7  end
```

(a) The trace transform encoded in Gen's trace transform language.



(b) Schematic of the trace transform.

Figure 3-21: Trace transform for a state-dependent mixture MCMC kernel

3.8 Related work

Composing Monte Carlo inference algorithms using probabilistic programs Some other probabilistic programming systems also support importance sampling using proposals expressed in probabilistic programming languages [13] and Hamiltonian Monte Carlo over selected sets of random choices [79]. However, to our knowledge no other probabilistic programming system supports a Metropolis-Hastings construct analogous to that of Section 3.4.2, particle filtering and annealed importance sampling constructs analogous to those of Section 3.5, involutive MCMC construct analogous to that of Section 3.7, or general sequential Monte Carlo that bridges arbitrary pairs of models (Section 3.6).

Involutive MCMC The involutive MCMC construction was implemented [24] as part of the inference library of the Gen probabilistic programming system [32]. The construction was motivated in part by a desire for a simple interface that automated the implementation of reversible jump MCMC samplers [53], state-dependent mixtures of proposals on complex state spaces, and data-driven neural proposals. Gen's involutive MCMC construction has since been used by a number of researchers to design and implement MCMC algorithms in diverse domains, including computational biology [81] and artificial intelligence [133]. Neklyudov et al. [93] independently identified the involutive MCMC construction as a unifying framework for MCMC algorithms, and showed how more than a dozen classic and recent MCMC algorithms can be cast within this framework. Neklyudov et al. [93] also identified design principles for developing new MCMC algorithms using the involutive MCMC construction, and showed that the framework aids in the derivation of novel efficient MCMC algorithms. The involutive MCMC construction encompasses many existing classes of MCMC kernels, some of which explicitly make use of bijective or involutive deterministic maps. In particular, the reversible jump framework [53, 56] employs a family of continuously differentiable bijections between the parameter spaces of different models. Tierney [122] described a family of deterministic proposals based on a deterministic involution that is equivalent to involutive MCMC but without the auxiliary probability distribution. More recently, Spanbauer et al. [116] defined a class of deep generative models based on differentiable involutions and trained these models to serve as efficient proposal distributions on continuous state spaces; the resulting algorithm is an instance of involutive MCMC. Other probabilistic programming system besides Gen support some custom reversible jump samplers: Roberts et al. [105] present a system embedded in Haskell that generates the implementation of some reversible jump MCMC kernels from a high-level specification. Narayanan and Shan [90] give a technique that automatically computes Metropolis-Hastings acceptance probabilities in some settings; however, these approaches do not handle many kernels that can be handled by our involutive MCMC construct.

Sequential Monte Carlo in probabilistic programs While many probabilistic programming systems implement generic sequential Monte Carlo (SMC) schemes based on incremental extension of the model with additional state [130, 79, 40, 89], to our knowledge Gen is the only system that supports more general SMC schemes that involve doing inference in an arbitrary sequence of models represented as probabilistic models that may use

different representations. Gen's more flexible variant of SMC is based on a general template for SMC algorithms [33] that includes annealed importance sampling [91] as a special case. An more restricted version of Gen's support for inference across multiple models, that does not use an arbitrary bijection to map between states, was described by Cusumano-Towner et al. [27]. Stuhlmüller et al. [119] devised a sequential Monte Carlo scheme involving coarsening the domain of random choices incrementally, and implemented in a probabilistic programming language via a program transformation that generated a larger model that included all other models as sub models, which was then processed by the generic sequential Monte Carlo inference engine. However, it is much more restrictive because only handles a limited class of models, does not inter-operate well with rejuvenation moves, and can only use sequences of the same models with coarsened domains for random choices.

Differentiable programming languages for transforming random choices Other systems support computing probability density functions of random variables that result from deterministic transformations of other random variables written in differentiable programming languages [34]. However, these systems do not support distributions on objects with stochastic structure like choice dictionaries. Incorporating performance optimizations from these systems into Gen's trace transform DSL is an interesting area for future work.

Checking validity of inference procedures Gen performs dynamic checks that can detect some violations of the assumptions and is able to find many bugs in practice. Gen does not statically verify the asymptotic soundness of inference algorithms. Although Gen encourages use of asymptotically sound inference algorithm templates, Gen does not enforce this—users are free to use Gen's primitives however they please. However, it is possible to statically verify the validity of proposals for the types of constructs used by Gen. A natural approach for reasoning about the validity of proposals is to introduce a type system that describes the support of the probability distributions on choice dictionaries. For example, Lew et al. [73] builds on the inference programming constructs introduced by Gen and describes a restricted modeling language and type system for models and proposals, and shows that automatic type checking and type inference can detect invalid proposals. Lee et al. [72] demonstrated verification of stochastic variational inference in probabilistic programs via static analysis. Adding type systems that express the 'shape' (i.e. set of domains of choice dictionaries with nonzero density) and support of random choices (i.e. the address universes they are valid in) for generative functions to Gen, and providing users with optional automatic verification of soundness is a promising area of future work. Atkinson et al. [7] introduce a language for hand-coded inference implementations and a compiler that checks the validity of these implementations and generates an optimized implementation. While the inference language used is significantly different from that of Gen (which is based on explicit manipulation of traces), it is possible that ideas from the MCMC verification approach could be applied to analyze loops of Gen inference MCMC kernels.

Chapter 4

Encapsulating Inference Logic in Generative Functions and Traces

A system with a modular design is more easily changed than one with highly coupled components, because changes can be made to one module without requiring knowledge of how the other modules function. Modules with clearly defined interfaces are also more likely to be reusable across multiple applications. More modular systems are also more easily debugged and explained than less modular systems. How can we support modular probabilistic inference implementations? The abstract data types defined in Chapter 2 provide one type of modularity—they separate the low-level implementation details of inference algorithms from the high-level algorithm strategies described in Chapter 3. The designer of an inference algorithm does not need to understand how a trace data type is implemented in order to use traces, and the trace implementation for a model can be improved, resulting in speedups of any inference code that uses it, without requiring the inference code to change.

Many other probabilistic programming systems have focused on a more ambitious type of modularity in which modeling is completely decoupled from the inference algorithm using an 'inference engine' based on generic, built-in, probabilistic inference algorithms. This is an appealing goal, but can be ineffective because many inference problems require specialization to the model that is not readily automated. This chapter explores an intermediate level of modularity in inference implementations that is based on extending generative functions and their traces with built-in inference capabilities called *internal proposals*. These built-in inference capabilities have well-defined interfaces, so that users can use them without knowing about how they are implemented, like the user of an inference engine. However, this design differs from the 'inference engine' architecture in three key respects: First, users are not restricted to using the built-in inference capabilities—they can freely combine their own custom inference logic with the built-in inference logic (e.g. by combining custom proposal distributions with the internal proposals). Second, because generative functions are composed from other generative functions, built-in inference capabilities can be used for only selected parts the model. Third, the built-in capabilities themselves have flexibility. For example, with *selection Metropolis-Hastings* (Section 4.3), users can apply MCMC moves based on the internal proposal to arbitrary subsets of random choices of their choosing.

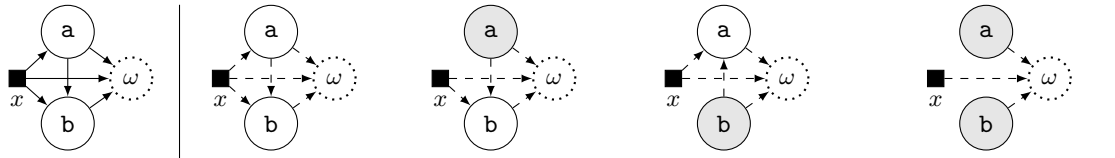


Figure 4-1: Schematic of internal proposal family for a simple generative function

4.1 Generative functions with internal proposals

This section extends the generative function and trace data types with a notion of *internal proposal*, which allows generative functions to encapsulate reusable inference logic. Generative functions with internal proposals can be used with the inference procedures of Chapter 3, but also serve as the basis for additional inference procedures described later in this chapter. Internal proposals enable reuse of inference logic—the internal proposal of a generative function can be used by any inference implementation for any model that invokes the generative function. The consumer of a generative function with an internal proposal does not need to know about what the internal proposal is in order to write asymptotically exact inference procedures that use the proposal.

A generative function $\mathcal{P}$ with an internal proposal is a tuple of five elements ($\mathcal{P} = (X, Y, p, f, q)$), where q is the *internal proposal family*. Briefly, q is a family of probability densities on choice dictionaries that, like p, is parametrized by the arguments x. But unlike p, q is also parametrized by a choice dictionary $\boldsymbol{\sigma}$ that represents a partial assignment to the random choices in a trace. The internal proposal family maps partial assignments $\boldsymbol{\sigma}$ to probability distributions that 'fill in' the rest of the random choices. Figure 4-1 diagrams these distributions for a generative function that makes two random choices at addresses `a` and `b` (ω is encapsulated randomness and will be introduced in Section 4.5).

Definition 4.1.1 (Internal proposal)**.** *A* generative function $\mathcal{P}$ *with an* internal proposal *is a tuple $\mathcal{P} = (X, Y, p, f, q)$, where X, Y, p, f are as in Definition 2.1.16, and where for each $(x, \boldsymbol{\sigma})$ such that $x \in X$ and $\bar{p}(\boldsymbol{\sigma}; x) > 0$, $q(\cdot; x, \boldsymbol{\sigma}) : \mathcal{T}^{\star}_{A \setminus A_{\boldsymbol{\sigma}}} \to [0, \infty)$ is a probability density on $\mathcal{T}^{\star}_{A \setminus A_{\boldsymbol{\sigma}}}$. In the discrete setting, this means that $\sum_{\boldsymbol{v} \in \mathcal{T}^{\star}_{A \setminus A_{\boldsymbol{\sigma}}}} q(\boldsymbol{v}; x, \boldsymbol{\sigma}) = 1$. In the general setting, this means that $q(\cdot; x, \boldsymbol{\sigma})$ is a $\mu^{\star}_{A \setminus A_{\boldsymbol{\sigma}}}$-measurable function such that $\int_{\mathcal{T}^{\star}_{A \setminus A_{\boldsymbol{\sigma}}}} q(\boldsymbol{v}; x, \boldsymbol{\sigma}) \mu^{\star}_{A \setminus A_{\boldsymbol{\sigma}}}(d\boldsymbol{v}) = 1$. Finally, q must satisfy $q(\boldsymbol{v}; x, \boldsymbol{\sigma}) > 0 \iff p(\boldsymbol{v} | \boldsymbol{\sigma}; x) > 0$ for all x and $\boldsymbol{\sigma}$ where $x \in X$ and $\bar{p}(\boldsymbol{\sigma}; x) > 0$.*

Although the internal proposal is only defined for x and $\boldsymbol{\sigma}$ such that $\bar{p}(\boldsymbol{\sigma}; x) > 0$, if $p(\cdot; x)$ is supportive (Definition 2.1.7) for all $x \in X$ then $q(\cdot; x, \boldsymbol{\sigma})$ is defined for all $(x, \boldsymbol{\sigma}) \in X \times \mathcal{T}^{\star}_{A}$.

4.1.1 Extending the generate operation using the internal proposal

Recall the GENERATE operation from Chapter 2.3, defined by $\mathcal{P}.\text{GENERATE}(x, \boldsymbol{\sigma}) := \mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$ for a generative function $\mathcal{P} = (X, Y, p, f)$ where $x \in X$ and $\boldsymbol{\tau} = \boldsymbol{\sigma}|_B \in \text{supp}(p(\cdot; x))$

for some B. A limitation of GENERATE as defined earlier is that it may be hard to construct a choice dictionary $\boldsymbol{\sigma}$ satisfying the requirements. We now describe a generalization of the operation that allows it to accept essentially any choice dictionary (denoted $\boldsymbol{\sigma}$) and *automatically* construct a valid choice dictionary $\boldsymbol{\tau}$ that agrees with $\boldsymbol{\sigma}$ using the internal proposal. Intuitively, the operation is able to 'fill in' the choices required to construct a choice that are not provided in the choice dictionary $\boldsymbol{\sigma}$. Furthermore, the operation returns a log importance weight that relates the internal proposal density to the model density p that can be used in various inference algorithms.

Extended generate operation The new operation input signature is $\mathcal{P}.\text{GENERATE}(x, \boldsymbol{\sigma})$ where $\mathcal{P} = (X, Y, p, f, q)$ and $x \in X$ and $\bar{p}(\boldsymbol{\sigma}; x) > 0$. It returns a pair $(\mathbf{t}, \log w)$ where $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$ and $\boldsymbol{\tau}$ and $\log w$ are defined as follows. First, $\boldsymbol{\upsilon} \sim q(\cdot; x, \boldsymbol{\sigma})$. Next, let $B \subseteq A_{\boldsymbol{\sigma}}$ be such that $\boldsymbol{\tau} := \boldsymbol{\upsilon} \oplus (\boldsymbol{\sigma}|_B) \in \text{supp}(p(\cdot; x))$. Such a B is guaranteed to exist because (i) $q(\boldsymbol{\upsilon}; x, \boldsymbol{\sigma}) > 0$ implies $p(\boldsymbol{\upsilon}|\boldsymbol{\sigma}; x) > 0$, and (ii) by the definition of $p(\boldsymbol{\upsilon}|\boldsymbol{\sigma}; x)$ in Equation (2.8). The resulting dictionary $\boldsymbol{\tau}$ is unique by Proposition 2.1.3 because $p(\cdot; x)$ is structured. Let $\log w \leftarrow p(\boldsymbol{\tau}; x)/q(\boldsymbol{\upsilon}; x, \boldsymbol{\sigma})$.

Example: Consider the generative function $\mathcal{P} = (X, Y, p, f, q)$ that takes no arguments and has return value `nothing`, but where p and $q(\cdot; x, \boldsymbol{\sigma})$ for $\boldsymbol{\sigma}_1 = \{\texttt{c} \mapsto \texttt{T}\}$ and $\boldsymbol{\sigma}_2 = \{\texttt{b} \mapsto \texttt{F}\}$ are defined below, along with the $\boldsymbol{\tau} = \boldsymbol{\upsilon} \oplus (\boldsymbol{\sigma}|_B)$ that results from each combination of $\boldsymbol{\sigma}$ and $\boldsymbol{\upsilon}$. Note that defining q for all $\boldsymbol{\sigma}$ would require many probability tables.

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau})$
$\{\texttt{a} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{F}\}$	0.45
$\{\texttt{a} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{T}\}$	0.05
$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{F}\}$	0.05
$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{T}\}$	0.2
$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{T}, \texttt{c} \mapsto \texttt{F}\}$	0.125
$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{T}, \texttt{c} \mapsto \texttt{T}\}$	0.125

$\boldsymbol{\upsilon}$	$q(\boldsymbol{\upsilon}; x, \boldsymbol{\sigma}_1)$	$\boldsymbol{\tau}$
$\{\texttt{a} \mapsto \texttt{F}\}$	0.4	$\{\texttt{a} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{T}\}$
$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}\}$	0.3	$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{T}\}$
$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{T}\}$	0.3	$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{T}, \texttt{c} \mapsto \texttt{T}\}$

$\boldsymbol{\upsilon}$	$q(\boldsymbol{\upsilon}; x, \boldsymbol{\sigma}_2)$	$\boldsymbol{\tau}$
$\{\texttt{a} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{F}\}$	0.2	$\{\texttt{a} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{F}\}$
$\{\texttt{a} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{T}\}$	0.2	$\{\texttt{a} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{T}\}$
$\{\texttt{a} \mapsto \texttt{T}, \texttt{c} \mapsto \texttt{F}\}$	0.3	$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{F}\}$
$\{\texttt{a} \mapsto \texttt{T}, \texttt{c} \mapsto \texttt{T}\}$	0.3	$\{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{T}\}$

Suppose we called $\mathcal{P}.\text{GENERATE}(x, \boldsymbol{\sigma}_1)$ and sampled $\boldsymbol{\upsilon} = \{\texttt{a} \mapsto \texttt{F}\}$ from $q(\cdot; x, \boldsymbol{\sigma}_1)$. Then, $B = \{\texttt{c}\}$ and $\boldsymbol{\tau} = \{\texttt{a} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{T}\}$, and $\log w = \log p(\boldsymbol{\tau}) - \log q(\boldsymbol{\upsilon}; x, \boldsymbol{\sigma}_1) = \log 0.05 - \log 0.4$.

4.1.2 The regenerate trace operation

Recall the UPDATE operation of the trace ADT from Chapter 2.3, $\mathbf{t}.\text{UPDATE}(x', \delta_X, \boldsymbol{\sigma})$. This operation returned a new trace $\mathbf{t}'$ that with new choices $\boldsymbol{\tau}'$ that are constructed from a combination of the previous choices $\boldsymbol{\tau}$ and the newly provided choices $\boldsymbol{\sigma}$. This section describes an additional operation that can be used to obtain a modified trace $\mathbf{t}'$ from an input trace $\mathbf{t}$. But instead of taking the new values $\boldsymbol{\sigma}$ as an input, this new operation, which is called REGENERATE, samples new choices from the internal proposal distribution,

and the caller only needs to specify *which* addresses should have new values sampled, not what the values should be.

Regenerate operation This operation is only defined for traces $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$ and $\mathcal{P} = (X, Y, p, f, q)$ where $p(\cdot; x)$ is supportive (Definition 2.1.7) for all $x \in X$. The input signature of the operation is $\mathbf{t}.\text{REGENERATE}(x', \delta_X, S)$ where $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$. Like for the UPDATE operation, the first and second inputs x' and δ_X are new arguments $x' \in X$ and a change hint δ_X from x to x', respectively. The third argument $S \subseteq A$ is a set of addresses (A is the set of all addresses in the address universe) called the *selection*. The operation returns a tuple $(\mathbf{t}', \log w, \delta_Y)$ where $\mathbf{t}' = (\mathcal{P}, x', \boldsymbol{\tau}')$ with $\boldsymbol{\tau}'$ and $\log w$ defined as follows. Let $\boldsymbol{\sigma} := \boldsymbol{\tau}|_{S^\complement}$ (the dictionary resulting from removing all addresses in S from $\boldsymbol{\tau}$). The operation samples from the internal proposal as follows: $\boldsymbol{\upsilon} \sim q(\cdot; x', \boldsymbol{\sigma})$. Let $B \subseteq A_{\boldsymbol{\sigma}}$ be such that $\boldsymbol{\tau}' := \boldsymbol{\upsilon} \oplus (\boldsymbol{\sigma}|_B) \in \text{supp}(p(\cdot; x'))$. Such a B is guaranteed to exist and the resulting $\boldsymbol{\tau}'$ is guaranteed to be unique by the same argument used for the generate operation above. Let $\mathbf{t}' := (\mathcal{P}, x', \boldsymbol{\tau}')$. Let $\boldsymbol{\sigma}' = \boldsymbol{\tau}'|_{S^\complement}$ and let $\boldsymbol{\upsilon}' := \boldsymbol{\tau}|_{((A_{\boldsymbol{\tau}} \cap A_{\boldsymbol{\tau}'}) \setminus S)^\complement}$. Then, define $\log w$ to be:

$$\log w := \log \frac{p(\boldsymbol{\tau}'; x') q(\boldsymbol{\upsilon}'; x, \boldsymbol{\sigma}')}{p(\boldsymbol{\tau}; x) q(\boldsymbol{\upsilon}; x', \boldsymbol{\sigma})} \tag{4.1}$$

Finally set δ_Y to a change hint from $f(x, \boldsymbol{\tau})$ to $f(x', \boldsymbol{\tau}')$. The supportive requirement ensures that $\bar{p}(\boldsymbol{\sigma}; x) > 0$ for all x and all $\boldsymbol{\sigma}$. In particular, it ensures that $\bar{p}(\boldsymbol{\tau}|_{S^\complement}; x') > 0$ and $\bar{p}(\boldsymbol{\tau}'|_{S^\complement}; x) > 0$, for all S, which ensures that the internal proposal is defined as necessary for all S. While the REGENERATE operation could be defined for generative functions where $p(\cdot; x)$ is not supportive for all x, the set of valid selections S would be complicated to characterize.

Example: Consider the generative function $\mathcal{P} = (X, Y, p, f, q)$ defined in immediately preceding example. Consider an input trace $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$ where $\boldsymbol{\tau} = \{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{T}\}$. Consider applying REGENERATE with selection $S = \{\texttt{a}, \texttt{b}\}$ (the inputs x and δ_X do not matter here because $\mathcal{P}$ takes no arguments). This choice of selection indicates that new values should be proposed using the internal proposal for `a` and `b` and/or the addresses may be removed. Specifically, we have $\boldsymbol{\sigma} := \boldsymbol{\tau}|_{S^\complement} = \{\texttt{c} \mapsto \texttt{T}\}$. The set of possible $\boldsymbol{\upsilon}$ that could be sampled from $q(\cdot; x, \{\texttt{c} \mapsto \texttt{T}\})$ are $\{a \mapsto \texttt{F}\}$, $\{a \mapsto \texttt{T}, b \mapsto \texttt{F}\}$, and $\{a \mapsto \texttt{T}, b \mapsto \texttt{T}\}$ with probabilities 0.4, 0.3, and 0.3 respectively. Suppose we sample $\boldsymbol{\upsilon} = \{a \mapsto \texttt{F}\}$. Then, $\boldsymbol{\tau}' := \boldsymbol{\upsilon} \oplus (\boldsymbol{\sigma}|_B) = \{\texttt{a} \mapsto \texttt{F}, \texttt{c} \mapsto \texttt{T}\}$ where $B = \{\texttt{c}\}$. Then, $\boldsymbol{\upsilon}' := \boldsymbol{\tau}|_{\{\texttt{c}\}^\complement} = \{\texttt{a} \mapsto \texttt{T}, \texttt{b} \mapsto \texttt{F}\}$. The log weight is:

$$\log w := \log \frac{p(\boldsymbol{\tau}'; x') q(\boldsymbol{\upsilon}'; x, \boldsymbol{\sigma}')}{p(\boldsymbol{\tau}; x) q(\boldsymbol{\upsilon}; x', \boldsymbol{\sigma})} = \log \frac{0.05 \cdot 0.3}{0.2 \cdot 0.4}$$

4.1.3 Example internal proposal families

There is a great deal of flexibility in the internal proposal family of a generative function. We now describe two classes of internal proposal families and how the ADT operations GENERATE and REGENERATE behave for these classes.

Example: Forward sampling in Bayesian networks Suppose that X and p are such that $\boldsymbol{\tau} \in \text{supp}(p(\cdot; x))$ implies $A_{\boldsymbol{\tau}} = \{a_1, \ldots, a_n\}$ for all $x \in X$. Let $A_{1:i} := \{a_1, \ldots, a_i\}$ for each i. Suppose that for all $\boldsymbol{\tau} \in \text{supp}(p(\cdot; x))$:

$$p(\boldsymbol{\tau}; x) = \prod_{i=1}^{n} p(\{a_i \mapsto \boldsymbol{\tau}[a_i]\} | (\boldsymbol{\tau}|_{A_{1:i-1}}); x)$$

Consider the internal proposal family defined by:

$$q(\boldsymbol{\upsilon}; x, \boldsymbol{\sigma}) := \prod_{i=1}^{n} \left(p(\{a_i \mapsto \boldsymbol{\upsilon}[a_i]\} | (\boldsymbol{\sigma}|_{A_{1:i-1}} \oplus \boldsymbol{\upsilon}|_{A_{1:i-1}}); x)\right)^{[a_i \notin A_{\boldsymbol{\sigma}}]}$$

for each $\boldsymbol{\upsilon}$ such that $\boldsymbol{\upsilon} \oplus (\boldsymbol{\sigma}|_{\{a_1,\ldots,a_n\}}) \in \text{supp}(p(\cdot; x))$ and zero otherwise. The log weight $\log w$ returned by the extended GENERATE operation is:

$$\log w = \sum_{i=1}^{n} [a_i \in A_{\boldsymbol{\sigma}}] \log p(\{a_i \mapsto \boldsymbol{\upsilon}[a_i]\} | (\boldsymbol{\sigma}|_{A_{1:i-1}} \oplus \boldsymbol{\upsilon}|_{A_{1:i-1}}); x)$$

The log weight returned by REGENERATE is, due to cancellation of factors:

$$\log w = \sum_{i=1}^{n} [a_i \in A_{\boldsymbol{\sigma}}] \begin{pmatrix} \log p(\{a_i \mapsto \boldsymbol{\upsilon}[a_i]\} | (\boldsymbol{\sigma}|_{A_{1:i-1}} \oplus \boldsymbol{\upsilon}|_{A_{1:i-1}}); x) \\ -\log p(\{a_i \mapsto \boldsymbol{\upsilon}'[a_i]\} | (\boldsymbol{\sigma}|_{A_{1:i-1}} \oplus \boldsymbol{\upsilon}'|_{A_{1:i-1}}); x) \end{pmatrix}$$

Note that the behavior of GENERATE and REGENERATE is different depending on the ordering of the addresses $a_1, \ldots, a_n$ used. A similar construction is possible for more general p for which the set of address is not fixed. Chapter 5 discusses how proposal families based on forward sampling can be compiled for flexible probabilistic modeling languages that include stochastic control flow. Briefly, the we sample from the internal proposal by executing the source code of the generative function, but we intercept each random choice expression and look up its address a in $\boldsymbol{\sigma}$. If $a \notin A_{\boldsymbol{\sigma}}$ then the value of the random choice is randomly sampled as in a regular execution; if $a \in A_{\boldsymbol{\sigma}}$ then the value of the random choice is deterministically set to $\boldsymbol{\sigma}[a]$.

Example: The optimal internal proposal family For a given X and p the *optimal* internal proposal family q is given by $q(\boldsymbol{\upsilon}; x, \boldsymbol{\sigma}) := p(\boldsymbol{\upsilon}|\boldsymbol{\sigma}; x)$ for all $\boldsymbol{\sigma}$ such that $\bar{p}(\boldsymbol{\sigma}; x) > 0$ and all $x \in X$ and all $\boldsymbol{\upsilon}$. For a generative function with such q, the log weight $\log w$ returned by the extended GENERATE operation is:

$$\log w = \log \frac{p(\boldsymbol{\upsilon} \oplus \boldsymbol{\sigma}|_B; x)}{p(\boldsymbol{\upsilon}|\boldsymbol{\sigma}; x)} = \log \left(\bar{p}(\boldsymbol{\sigma}; x) \frac{p(\boldsymbol{\upsilon} \oplus \boldsymbol{\sigma}|_B; x)}{\sum_{C \subseteq A_{\boldsymbol{\sigma}}} p(\boldsymbol{\upsilon} \oplus (\boldsymbol{\sigma}|_C); x)} \right) = \log \bar{p}(\boldsymbol{\sigma}; x)$$

That is, the log weight is deterministic and gives the log marginal likelihood. For the REGENERATE operation, the log weight when using the optimal internal proposal family is:

$$\log w = \log \frac{p(\boldsymbol{\tau}'; x')}{p(\boldsymbol{\upsilon}|\boldsymbol{\sigma}; x')} \frac{p(\boldsymbol{\upsilon}'|\boldsymbol{\sigma}'; x)}{p(\boldsymbol{\tau}; x)} = \log \frac{\bar{p}(\boldsymbol{\sigma}; x')}{\bar{p}(\boldsymbol{\sigma}'; x)} = \log \frac{\bar{p}(\boldsymbol{\tau}|_{S^\complement}; x')}{\bar{p}(\boldsymbol{\tau}'|_{S^\complement}; x)}$$

In the special case when $A_{\boldsymbol{\tau}} \Delta A_{\boldsymbol{\tau}'} \subseteq S$ and $x = x'$, $\boldsymbol{\tau}|_{S^\complement} = \boldsymbol{\tau}'|_{S^\complement}$ so $\log w = 1$ (where Δ denotes symmetric difference of sets). By further specializing to $A_{\boldsymbol{\tau}} = A_{\boldsymbol{\tau}'}$ the REGENERATE operation with the optimal proposal family recovers Gibbs sampling [44]. Section 4.3 shows how REGENERATE can be used to construct MCMC kernels, including when when q is not the optimal proposal family. Chapter 5 discusses how optimal proposal families can be constructed automatically for generative functions specified in specialized probabilistic modeling languages.

4.2 Importance sampling with the internal proposal

Internal proposal families are a mechanism for inference logic to be encapsulated inside generative functions and traces. Internal proposal families, like the other aspects of the generative function and trace ADTs are automatically generated from probabilistic modeling code by the modeling language compiler. Users can choose to employ the internal proposal for their model instead of writing a proposal generative function by hand. Many of the inference operations described in Chapter 3 have analogues that are based in internal proposals and require much less work by the user.

For example, consider the the self-normalized importance sampling construct in Algorithm 3, for which the user writes a generative function $\mathcal{Q}$ to serve as a proposal distribution for a model $\mathcal{P}$. Algorithm 13 gives an analogous procedure for self-normalized importance sampling that uses the internal proposal family instead and is much simpler, requiring the user to specify the generative model ($\mathcal{P}$), the observed data ($\boldsymbol{\rho}$) and the number of samples (n). Gen's inference library includes an implementation of this procedure, which is the simplest entry-point for new users to Gen because of the simplicity of its interface.

Algorithm 13 Self-normalized importance sampling using the internal proposal

procedure SELF-NORM-IMPORTANCE-SAMPLING-INTERNAL($\mathcal{P}$, $\boldsymbol{\rho}$, n)
  for $i \leftarrow 1 \ldots n$ **do**
    $(\mathbf{t}^{(i)}, \log \tilde{w}_i) \leftarrow \mathcal{P}.\text{GENERATE}(\_, \boldsymbol{\rho})$
  end for
  $((w^{(1)}, \ldots, w^{(n)}), \log \widehat{z}) \leftarrow \text{NORMALIZE}(\log \tilde{w}^{(1)}, \ldots, \log \tilde{w}^{(n)})$
  return $\left(\{(\mathbf{t}^{(1)}, w^{(1)}), \ldots, (\mathbf{t}^{(n)}, w^{(n)})\}, \log \widehat{z}\right)$
end procedure

It is also possible to adapt the Algorithm 3 for use with generative functions $\mathcal{P}$ that have internal proposal families. Algorithm 14 shows the modified procedure. In this procedure, if the external proposal $\mathcal{Q}$ does not sample all random choices visited in the model, the remaining choices will be sampled by the internal proposal, resulting in an importance sam-

Algorithm 14 Self-normalized importance sampling with an internal and external proposal

procedure SELF-NORM-IMPORTANCE-SAMPLING-MIXED($\mathcal{P}$, $\mathcal{Q}$, $\boldsymbol{\rho}$, n)
  for $i \leftarrow 1 \ldots n$ **do**
    $\mathbf{s} \leftarrow \mathcal{Q}.\text{SIMULATE}()$
    $\boldsymbol{\sigma} \leftarrow \mathbf{s}.\text{CHOICES}() \oplus \boldsymbol{\rho}$
    $(\mathbf{t}^{(i)}, \log \tilde{w}^{(i)}) \leftarrow \mathcal{P}.\text{GENERATE}(\_, \boldsymbol{\sigma})$
    $\log \tilde{w}^{(i)} \leftarrow \log \tilde{w}^{(i)} - \mathbf{s}.\text{LOGPDF}()$
  end for
  $((w^{(1)}, \ldots, w^{(n)}), \log \widehat{z}) \leftarrow \text{NORMALIZE}(\log \tilde{w}^{(1)}, \ldots, \log \tilde{w}^{(n)})$
  return $\left(\{(\mathbf{t}^{(1)}, w^{(1)}), \ldots, (\mathbf{t}^{(n)}, w^{(n)})\}, \log \widehat{z}\right)$
end procedure

pling algorithm whose proposal is a combination of the internal proposal and the external proposal. Such an external proposal would result in an error if $\mathcal{P}$ was not equipped with an internal proposal, because intuitively GENERATE would not be able to fill in the choices that were not provided by the external proposal. Therefore, the presence of internal proposals greatly relaxes the constraints on inference programmers—they are able to incrementally increase the complexity and efficiency of their inference algorithms by gradually increasing the purview of their external proposals. This can be understood as *interpolating* between the approach of systems like Church [51], Venture [79], Anglican [130], and Turing [40] which are based on default proposals, and the inference programming approach of Chapter 3, which required external proposals to sample all relevant random choices in the model in the context of importance sampling and particle filtering.

4.3 Selection Metropolis-Hastings

The Markov chain Monte Carlo (MCMC) kernel constructs that were described in Chapter 3 are flexible enough to permit a wide class of custom Metropolis-Hastings and reversible jump MCMC kernels, including custom trans-dimensional moves, while automating the sampling and acceptance probability calculation. However, these constructs require the user to understand the semantics of the model and in some cases, write significant code specifying the kernel. *Selection Metropolis-Hastings* (selection MH) is a novel construct for MCMC kernels that allows the user to only *select* the set of random choices that should be updated, in the form of an *address selection* $B \subseteq A$ where A is the set of all addresses. The construct uses the internal proposal family of the model instead of requiring the user to specify a custom external proposal. Selection MH kernels can be applied in cycles or mixtures with different selections, or with other kernel types. With selection MH the user maintains significant control over the inference algorithm—the user chooses how the random choices should be partitioned for updates.

Using the new REGENERATE operation, the procedure for selection MH is quite simple, and is shown in Algorithm 15. Note that it is simply a wrapper around a call to REGENERATE that accepts or rejects the resulting trace according to the log weight $\log w$. The first

Algorithm 15 Selection Metropolis-Hastings

procedure SELECTION-MH-KERNEL$_B(\mathbf{t})$
  $(\mathbf{t}', \log w, \_) \leftarrow \mathbf{t}.\text{REGENERATE}(\mathbf{t}.\text{ARGS}(), \top, B)$
  $r \sim \text{Uniform}(0, 1)$
  if $r \leq \log w$ **then return** $\mathbf{t}'$ **else return** $\mathbf{t}$
end procedure

argument to regenerate passes the arguments from the input trace ($x' = x$), and the second argument is the change hint $\delta_X = \top$ that indicates that the new arguments are the same as the previous arguments. The third argument indicates the set of addresses for which new values should be sampled. Note that REGENERATE, and this algorithm, only support models $\mathcal{P}$ whose densities are supportive (Definition 2.1.7). For models expressed in DML, it is possible construct a model with a supportive density by guaranteeing that, for each address sampled in any execution, the support of the distribution at that distribution is constant across all executions (see Chapter 5 for details).

Selection MH in Gen's inference library This construct is provided in Gen's inference library as a variant of the `Gen.mh` function:

```
(new_trace, _) = Gen.mh(trace, selection)
```

Note that this construct requires very little code to use, because it leverages the encapsulated inference capabilities of the internal proposal. Gen includes various ways of constructing address selections that include selecting individual addresses, selecting whole namespaces, and set operations on selections. For example, to construct the selection $B = \{\texttt{foo}, \texttt{bar}\}$, we use `Gen.select(:foo, :bar)`, and to select all addresses except for `foo` and `bar` we use `Gen.complement(Gen.select(:foo, :bar))`.

Example: Selection MH and stochastic control flow Consider the (anonymous) model generative function $\mathcal{P}$ defined below, which has stochastic structure.

```
@gen function ()
  val = true
  if ({:a} ~ bernoulli(0.3))
    b ~ bernoulli(0.6)) && val
  end
  p = val ? 0.9 : 0.2
  c ~ bernoulli(p)
end
```

$\boldsymbol{\tau}$	$p(\boldsymbol{\tau})$
$\{a \mapsto \texttt{T}, b \mapsto \texttt{T}, c \mapsto \texttt{T}\}$	$0.3 \cdot 0.6 \cdot 0.9$
$\{a \mapsto \texttt{T}, b \mapsto \texttt{T}, c \mapsto \texttt{F}\}$	$0.3 \cdot 0.6 \cdot 0.1$
$\{a \mapsto \texttt{T}, b \mapsto \texttt{F}, c \mapsto \texttt{T}\}$	$0.3 \cdot 0.4 \cdot 0.2$
$\{a \mapsto \texttt{T}, b \mapsto \texttt{F}, c \mapsto \texttt{F}\}$	$0.3 \cdot 0.4 \cdot 0.8$
$\{a \mapsto \texttt{F}, c \mapsto \texttt{T}\}$	$0.7 \cdot 0.9$
$\{a \mapsto \texttt{F}, c \mapsto \texttt{F}\}$	$0.7 \cdot 0.1$

Suppose the internal proposal uses forward sampling, and that the previous trace contains $\boldsymbol{\tau} = \{a \mapsto \texttt{T}, b \mapsto \texttt{T}, c \mapsto \texttt{T}\}$. Suppose we apply SELECTION-MH with the selection $B = \{a\}$. Then, $\boldsymbol{\sigma}$ is the restriction of $\boldsymbol{\tau}$ to the complement of $\{a\}$, giving $\boldsymbol{\sigma} = \{b \mapsto \texttt{T}, c \mapsto \texttt{T}\}$. The possible samples from the internal proposal for this $\boldsymbol{\sigma}$ are listed below, along with the

resulting $\boldsymbol{\tau}'$ and the acceptance probability:

$\boldsymbol{\upsilon}$	$q(\boldsymbol{\upsilon}; x, \boldsymbol{\sigma})$	$\boldsymbol{\tau}'$	$\boldsymbol{\sigma}'$	Acceptance probability
$\{a \mapsto \texttt{T}\}$	0.3	$\{a \mapsto \texttt{T}, b \mapsto \texttt{T}, c \mapsto \texttt{T}\}$	$\{b \mapsto \texttt{T}, c \mapsto \texttt{T}\}$	1
$\{a \mapsto \texttt{F}\}$	0.7	$\{a \mapsto \texttt{F}, c \mapsto \texttt{T}\}$	$\{c \mapsto \texttt{T}\}$	0.6

Note that it is possible to use selection MH with an arbitrary set of selected addresses, including addresses that may not exist in the current trace.

Example: Selection MH in a Dirichlet process mixture model Consider modeling a collection of n real-valued data points $y_i \in \mathbb{R}$ for $i \in \{1, \ldots, n\}$ using a mixture model in which there are k mixture components. For each i, the variable $z_i \in \{1, \ldots, k\}$ gives the component to which data point i is assigned. Each mixture component j for $j \in \{1, \ldots, k\}$ has a mean parameter $\mu_j \in \mathbb{R}$ with a prior distribution that is a normal distribution with large variance. All data points y_i for which $z_i = j$ are modeled as independently and identically distributed according to a normal distribution with mean μ_j and variance σ^2. The generative function $\mathcal{P}_1 = \texttt{mixture\_data}$ below implements this model. It takes as input the cluster assignments (the vector $[z_1, z_2, \ldots, z_n]$) and the variance σ^2, and proceeds to sample the cluster mean μ_j at address $(\texttt{mu}, j)$ for each $j \in \{1, \ldots, n\}$. Next it samples each data point y_i at address $(\texttt{y}, i)$ for $i \in \{1, \ldots, n\}$ from the respective normal distribution.

```
@gen function mixture_data(z, var)
    n = length(z)
    k = maximum(clusters_assignments)

    # sample the cluster parameters from the prior
    cluster_means = Dict()
    for j in 1:k
        cluster_means[j] = ({(:mu, j)} ~ normal(0.0, 10.0))
    end

    # sample the data points
    for i in 1:n
        {(:y, i)} ~ normal(cluster_means[z[i]], sqrt(var))
    end
end
```

The generative function $\mathcal{P}_2 = \texttt{dpmm}$ below defines a Dirichlet process mixture model, using `mixture_data` as a subroutine. The code takes as arguments the number of data points (n) and the alpha parameter for the Chinese restaurant process [3] (CRP) prior on the cluster assignments. The code then loops through the data points, and samples the cluster assignments (`z`) from the CRP prior using a parametrization based on a random choice $(\texttt{new\_cluster}, i)$ for each data point. If the data point does not form a new cluster, then it samples the index of a 'friend', which is one of the previous data points, and joins the cluster assignment of the friend. This parametrization of the CRP prior was designed so that the density is supportive—the support at each $(\texttt{friend}, i)$ address is always $\{1, \ldots, i-1\}$. Note that the number of clusters is random, and can vary between 1 and n. The variance

(var) is shared for all clusters, and has an inverse-gamma prior distribution. Given the cluster assignments and the variance, we invoke mixture_data from above, which samples parameters for each cluster, and then samples each data point. The function then returns the number of clusters k so that this derived value is easily accessible from inference code with $\mathbf{t}$.RETVAL() and does not need to be recomputed from the values of the random choices.

```
@gen function dpmm(n, alpha)
    z = Dict()
    k = 0
    for i in 1:n
        prob_new_cluster = alpha / (i - 1 + alpha)
        if ({(:new_cluster, i)} ~ bernoulli(prob_new_cluster))
            z[i] = k + 1
            k += 1
        else
            friend = ({(:friend, i)} ~ uniform_discrete(1, i-1))
            z[i] = z[friend]
        end
    end
    var ~ inv_gamma(1, 1)
    {:data} ~ mixture_data(z, var)
    return k
end
```

Suppose we are given observed data of the form $\boldsymbol{\rho} = \{(\mathtt{y}, 1) \mapsto y_1, \ldots, (\mathtt{y}, n) \mapsto y_n\}$. Consider the following composite MCMC kernel, constructed using the composite kernel DSL of Section 3.4.4 primitive MCMC kernels that are each constructed using SELECTION-MH:

```
@kern function dpmm_kernel(trace)
    for j in 1:5
        trace ~ Gen.mh(trace, Gen.select(:var))
    end
    let n = Gen.get_args(trace)[1]
        for i in 1:n
            let k = Gen.get_retval(trace)
                selection = Gen.select([:data => (:mu, j) for j=1:k]...,
                                       (:new_cluster, j), (:friend, j)))
                trace ~ Gen.mh(trace, selection)
            ends
        end
    end
end
```

Each application of this kernel beings by applying selection MH to just update the variance random choice in repetitions. Then, it loops through all the data points, and applies selection MH where the selection is a block of random choices that includes the cluster assignment choices for one data point, as well as the per-cluster parameters (in this example, just the cluster means) for all clusters. Using the current version of Gen, we apply this kernel in the code below, which reads the data from a variable y (definition not shown).

```
observations = Gen.choicemap()
for i in 1:length(xs)
    observations[:data => (:y, i)] = y[i]
end
trace, = Gen.generate(dpmm, (length(y), 1.0), observations)
for iter in 1:1000
    trace, = dpmm_kernel(trace)
end
```

The inference program beings by initializing a trace that contains the observed data, and then repeatedly applying the composite kernel. Note that what MCMC kernel is being applied by this code depends on the internal proposal used by `dpmm`. Since `dpmm` is written in DML, it uses forward sampling for its internal proposal by default. This means that the selection MH kernels for the variance will be independence MH moves that use the prior distribution as the proposal. However, note that `dpmm` invokes the separate generative function `mixture_data` to sample the cluster parameters and the data, and the internal proposal of that generative function will be invoked as part of the internal proposal of `dpmm`. If `mixture_data` is implemented in DML as above, then it will also use forward sampling for its internal proposal, but `mixture_data` could also have been implemented in a different modeling language, or with a custom ADT implementation that uses a different internal proposal family. The next section shows how to implement a version of `mixture_data` that uses a more efficient internal proposal family.

4.4 A combinator for overriding the internal proposal

In Chapter 5 we will discuss how internal proposal families can be implemented as part of modeling language compilers, which generate the generative function and trace abstract data types (ADTs). However, it is also possible to *override* the internal proposal family of a generative function $\mathcal{P}$ after it has already been constructed, using another generative function $\mathcal{Q}$ to define the internal proposal family. This section defines a combinator that implements the generative function and trace ADTs for a new generative function $\mathcal{R}$ using generative functions and traces of the original generative function $\mathcal{P}$ and the generative function $\mathcal{Q}$. The resulting generative function $\mathcal{R}$ has the same probability distribution on choice dictionaries as $\mathcal{P}$ but a different internal proposal family. This combinator is intended to be used to optimize the inference algorithm implementation for a model, by replacing a generic internal proposal family that is automatically generated from a compiler with a more efficient specialized internal proposal family.

Consider a generative function $\mathcal{P} = (X_1, Y_1, p, f, \tilde{q})$ where $\tilde{q}$ is the original internal proposal family. Suppose there exists another generative function $\mathcal{Q} = (X_2, Y_2, q, f_2, r)$ where $X_2 = X_1 \times \mathcal{T}_A^\star$ and for each $x \in X$ and $\boldsymbol{\sigma} \in \mathcal{T}_A^\star$, $q(\boldsymbol{v}; x, \boldsymbol{\sigma}) > 0$ if and only if $p(\boldsymbol{v}|\boldsymbol{\sigma}) > 0$. Consider the generative function $\mathcal{R} := (X_1, Y_1, p, f, q)$. Algorithm 16 shows how to implement each of the operations for the generative function and trace ADTs, using only operations of the generative function and trace ADTs of $\mathcal{P}$ and $\mathcal{Q}$.

Algorithm 16 Combinator for overriding the internal proposal

procedure $\mathcal{R}$.SIMULATE(x)
  $\mathbf{t} \leftarrow \mathcal{P}$.SIMULATE($x$)
  $\mathbf{u} \leftarrow (\mathcal{R}, x, \mathbf{t}$.CHOICES())
  return $\mathbf{u}$
end procedure

procedure $\mathcal{R}$.GENERATE(x, $\boldsymbol{\sigma}$)
  $\mathbf{s} \leftarrow \mathcal{Q}$.SIMULATE($(x, \boldsymbol{\sigma})$)
  $\boldsymbol{\upsilon} \leftarrow \mathbf{s}$.CHOICES()
  $(\mathbf{t}, \_) \leftarrow \mathcal{P}$.GENERATE($x, \boldsymbol{\upsilon} \oplus \boldsymbol{\sigma}$)
  $\log w \leftarrow \mathbf{t}$.LOGPDF() $- \mathbf{s}$.LOGPDF()
  $\mathbf{u} \leftarrow (\mathcal{R}, x, \mathbf{t}$.CHOICES())
  return $(\mathbf{u}, \log w)$
end procedure

Require: $\mathbf{u} = (\mathcal{R}, x, \boldsymbol{\tau})$

procedure $\mathbf{u}$.LOGPDF()
  $\mathbf{t} \leftarrow (\mathcal{P}, x, \boldsymbol{\tau})$
  return $\mathbf{t}$.LOGPDF()
end procedure

procedure $\mathbf{u}$.REGENERATE(x', δ_X, S)
  $\boldsymbol{\tau} \leftarrow \mathbf{u}$.CHOICES()
  $\boldsymbol{\sigma} \leftarrow \boldsymbol{\tau}|_{S^\mathsf{c}}$
  $(\mathbf{u}', \log w_1) \leftarrow \mathcal{R}$.GENERATE($x', \boldsymbol{\sigma}$)
  $\boldsymbol{\sigma}' \leftarrow \mathbf{u}'$.CHOICES()$|_{S^\mathsf{c}}$
  $\mathbf{s} \leftarrow \mathcal{Q}$.GENERATE($(x', \boldsymbol{\sigma}'), \boldsymbol{\tau}$)
  $\log w_2 \leftarrow \mathbf{u}$.LOGPDF() $- \mathbf{s}$.LOGPDF()
  return $(\mathbf{u}', \log w_1 - \log w_2, \bot)$
end procedure

procedure $\mathbf{u}$.CHOICES()
  return $\boldsymbol{\tau}$
end procedure

procedure $\mathbf{u}$.ARGS()
  return x
end procedure

procedure $\mathbf{u}$.RETVAL()
  $\mathbf{t} \leftarrow (\mathcal{P}, x, \boldsymbol{\tau})$
  return $\mathbf{t}$.RETVAL()
end procedure

Example: Rao-Blackwellizing mixture model parameters The generative function `mixture_data` above was implemented using Gen's Dynamic Modeling Language (DML). At the time of writing, Gen's DML compiler constructs an internal proposal based on forward sampling. In particular, given only the vector of data points $\boldsymbol{\sigma} = \{(\mathtt{y}, 1) \mapsto y_1, \ldots, (\mathtt{y}, n) \mapsto y_n\}$, the internal proposal distribution samples the cluster means from their prior distribution:

$$q(\boldsymbol{\upsilon}; x, \boldsymbol{\sigma}) = \prod_{j=1}^{k(x)} p_{\text{norm}(0,10)}(\boldsymbol{\upsilon}[(\mathtt{mu}, j)])$$

Here, x is the tuple $([z_1, \ldots, z_k], \sigma^2)$ and $k(x)$ extracts length of the cluster assignment vector. Note that this distribution does not depend on the values in $\boldsymbol{\sigma}$. This is an inefficient proposal distribution because the prior distribution on the parameters is very different from the posterior for typical values of σ^2 and n. Because the normal distribution is the conjugate prior for a normal likelihood, the optimal internal proposal family for the density p of this

generative function can be analytically derived, and is:

$$p(\boldsymbol{v}; x, \boldsymbol{\sigma}) := \prod_{j=1}^{k(x)} p_{\text{norm}\left(m_j(x,\boldsymbol{\sigma}), \sqrt{v_j(x,\boldsymbol{\sigma})}\right)}(\boldsymbol{v}[(\texttt{mu}, j)])$$

$$v_j(x, \boldsymbol{\sigma}) := \left(\frac{\sum_{i=1}^{n}[z_i = j]}{\sigma^2} + \frac{1}{10^2}\right)^{-1} \quad m_j(x, \boldsymbol{\sigma}) := v_j(x, \boldsymbol{v})\left(\frac{0}{10^2} + \frac{\sum_{i=1}^{n}[z_i = j]\boldsymbol{\sigma}[(\texttt{y}, i)]}{\sigma^2}\right)$$

when $A_{\boldsymbol{v}} = \{(\texttt{mu}, 1), \ldots, (\texttt{mu}, n)\}$ and zero otherwise. While we could aim to improve Gen's DML compiler so that it can automatically derive and generate this optimal internal proposal family, improving modeling language compilers is an ongoing (and promising) research challenge, and it is likely that there remain large classes of models that are difficult to automatically analyze. Therefore, a user of Gen can use the combinator defined in Algorithm 16 to override the internal proposal of `mixture_data` with the optimal internal proposal expressed in DML. The combinator is provided in Gen. The generative function $\mathcal{Q} =$ `mixture_proposal` below defines the optimal proposal family described above.

```
@gen function mixture_proposal(z, var, constraints)
    n = length(z)
    k = maximum(z)

    # compute sufficient statistics for each cluster
    sums = [0.0 for j in 1:k]
    counts = [0.0 for j in 1:k]
    for i=1:n
        if Gen.has_value(constraints, (:x, i))
            sums[z[i]] += constraints[(:x, i)]
            counts[z[i]] += 1
        end
    end

    # sample cluster parameters from the conditional distribution
    cluster_means = Dict()
    for j in 1:k
        if Gen.has_value(constraints, (:mu, j))
            cluster_means[j] = constraints[(:mu, j)]
        else
            cond_mu, cond_sigma = cond_params(var, sums[j], counts[j])
            cluster_means[j] = ((:mu, j) ~ normal(cond_mu, cond_sigma))
        end
    end

    # sample the data points that were not included in constraints
    for i=1:n
        if !Gen.has_value(constraints, (:x, i))
            {(:x, i)} ~ normal(cluster_means[z[i]], sqrt(var))
        end
    end
end
```

The arguments to this generative function consist of (i) the arguments to the original generative function (`z` and `var`, which encode $[z_1, \ldots, z_k]$ and σ^2 respectively) and (ii) a choice dictionary `constraints` (which encodes $\boldsymbol{\sigma}$). Note that to represent a valid internal proposal family, the code must be capable of handling any possible $\boldsymbol{\sigma}$. Therefore, it checks whether addresses are present in $\boldsymbol{\sigma}$ using `Gen.has_value`, and changes its behavior accordingly—if only a subset of the data points (with addresses $(\texttt{x}, i)$) for a given cluster are provided in $\boldsymbol{\sigma}$, then the sufficient statistics used to compute the conditional distribution on the cluster parameter only accumulates over the subset that is available, and the data points that are not provided in $\boldsymbol{\sigma}$ are sampled at the end, conditioned on the cluster parameters. Similarly, some subset of the cluster means (with addresses $(\texttt{mu}, j)$) may be provided, in which case a value is not sampled. For cluster means that are not provided, the parameters of the conditional distribution (m_j and v_j in the equation) are computed by `cond_params` and the cluster mean is sampled.

To construct the new generative function $\mathcal{R}$ from the original generative function $\mathcal{P}$ = `mixture_data` and the proposal generative function $\mathcal{Q}$ = `mixture_proposal`, we invoke the following Julia function, which returns a new generative function whose ADTs are implemented using Algorithm 16.

```
mixture_data = Gen.override_internal_proposal(mixture_data, mixture_proposal)
```

We call this function a *combinator* because it combines two generative functions and returns a new generative function. We will see other examples of generative function combinators in Chapter 5. After running this code, the new generative function `mixture_data` has the same probability distribution p as the original generative function but it uses the optimal internal proposal family. Now, in the MCMC algorithm for `dpmm` in the previous section, the selection MH kernels that propose new values for both the cluster assignments and the cluster parameters use a combination of forward sampling (for the new cluster assignment) and conditional sampling (for the cluster parameters). The resulting kernel can be understood as an MH kernel that uses a naive proposal for the cluster assignment, but marginalizes out the cluster parameters when computing the acceptance probability.

Encapsulation and modularity via overriding internal proposals Note that a user could have written an efficient proposal distribution that exploited conjugacy as a generative function, and used it with the original model with the inference constructs from Chapter 3, which do not require internal proposal families. However, after replacing the internal proposal in the example above, now *any* model generative function that invokes `mixture_data` will make use of its more efficient internal proposal distribution for the random choices made by `mixture_data`. Furthermore, *any* inference algorithm that uses GENERATE or REGENERATE will function without a code change, and will benefit automatically from the efficiency gain due to this change, because the new capability has been *encapsulated*. Composing reusable components of generative models (generative functions) and encapsulating custom inference logic (internal proposal families) inside these components is a promising approach to achieving more modularity in inference algorithm implementations.

4.5 Encapsulated randomness

The internal proposal family presented above allows us to encapsulate proposal distributions within generative functions and traces, but the resulting random choices are always included in $\boldsymbol{\tau}$ and accessible via the data type operations. This section extends generative functions and traces with the ability to encapsulate random choices themselves, which makes these data types significantly more flexible.

When we have a sampler for a probability distribution, together with the ability to evaluate its probability mass (or density) pointwise, we can use the distribution in a variety of ways, including in priors, likelihoods, proposals, and variational approximations. This is why software libraries for probability distributions typically provide routines for pointwise evaluation of the probability. However, evaluating a probability distribution pointwise is often intractable when sampling from it is not. Probabilistic generative models provide good examples of this—sampling from a generative model's prior distribution on data is trivial, whereas computing the marginal probability of some observed data is the difficult (and typically intractable) problem of evaluating the marginal likelihood. The generative function and trace data types we have introduced so far require that $\log p(\boldsymbol{\tau}; x)$ can be evaluated. This section extends the data types by allowing them to use *estimates* of $\log p(\boldsymbol{\tau}; x)$ in place of $\log p(\boldsymbol{\tau}; x)$. The estimates are obtained using *encapsulated randomness* $\omega \in \Omega$ that is used internally by the data type operations but is not part of the choice dictionary $\boldsymbol{\tau}$ and cannot be read from the trace directly.

Using auxiliary variable and pseudo-marginal Monte Carlo arguments, it is possible to show that the resulting generative functions can be used within inference algorithms in the same ways that generative functions that compute $\log p(\boldsymbol{\tau}; x)$ exactly can be used, without invalidating the asymptotic properties of the algorithms. This includes as components in generative models and as components in proposal distributions. The construction given in this section allows Gen users to express, in a modular and compositional way, a wide array of pseudo-marginal Monte Carlo inference algorithms.

Definition 4.5.1 (Encapsulated randomness)**.** *A generative function $\mathcal{P}$ with* encapsulated randomness *is a tuple $\mathcal{P} = (X, Y, p, f, q, \Omega, \mathring{p}, \mathring{q})$; where (X, Y, p, f, q) is a generative function with an internal proposal family*[1]*, $\mathring{p}$ and $\mathring{q}$ are families of probability densities on $\omega \in \Omega$ (with respect to some σ-finite reference measure ν on Ω) such that $\mathring{p}(\omega; x, \boldsymbol{\tau}) > 0$ if and only if $\mathring{q}(\omega; x, \boldsymbol{\tau}) > 0$ for all $x, \boldsymbol{\tau}$ such that $p(\boldsymbol{\tau}; x) > 0$. A trace $\boldsymbol{t}$ of such a generative function is a tuple $(\mathcal{P}, x, \boldsymbol{\tau}, \omega)$ where $(\mathcal{P}, x, \boldsymbol{\tau})$ are as defined earlier, and where $\omega \in \Omega$ satisfies $\mathring{p}(\omega; x, \boldsymbol{\tau}) > 0$.*

Generative functions with encapsulated randomness behave like generative functions without encapsulated randomness, except that in each of their operations, the density on choice dictionaries $p(\boldsymbol{\tau}; x)$ is replaced with an estimate $\xi(x, \boldsymbol{\tau}, \omega)$ where $\omega \in \Omega$ is a value of the encapsulated randomness.

Definition 4.5.2 (Choice density estimate)**.** *For a generative function $\mathcal{P}$ with encapsulated randomness, the* choice density estimate *is a function ξ of $x, \boldsymbol{\tau}$ and ω such that $p(\boldsymbol{\tau}; x) > 0$*

[1] Except that here, that f is a function of x, $\boldsymbol{\tau}$, and ω for $\omega \in \Omega$ instead of just x and $\boldsymbol{\tau}$.

and $\mathring{p}(\omega; x, \boldsymbol{\tau}) > 0$ *and is given by:*

$$\xi(x, \boldsymbol{\tau}, \omega) := \frac{p(\boldsymbol{\tau}; x)\mathring{p}(\omega; x, \boldsymbol{\tau})}{\mathring{q}(\omega; x, \boldsymbol{\tau})} \tag{4.2}$$

Example: Encapsulated randomness is a generalization A generative function without encapsulated randomness is equivalent to a generative function with encapsulated randomness that has a trivial set of possible values for encapsulated randomness $\Omega = \{\perp\}$ and $\mathring{p}(\perp; x, \boldsymbol{\tau}) = \mathring{q}(\perp; x, \boldsymbol{\tau}) = 1$.

Encapsulated randomness and unbiased estimators We can gain some intuition about Ω, $\mathring{p}$ and $\mathring{q}$ by interpreting them as components of estimators of the choice density $p(\boldsymbol{\tau}; x)$ that satisfy certain properties. In particular, note that sampling the encapsulated randomness ω from $\mathring{q}$ and evaluating the choice density estimate gives an unbiased estimate of the choice density $p(\boldsymbol{\tau}; x)$:

$$\mathbb{E}_{\omega \sim \mathring{q}(\cdot; x, \boldsymbol{\tau})} \left[\xi(x, \boldsymbol{\tau}, \omega)\right] = p(\boldsymbol{\tau}; x) \tag{4.3}$$

Similarly, sampling the encapsulated randomness ω from $\mathring{p}$, and evaluating the reciprocal of the choice density estimate gives an unbiased estimate of the reciprocal of the choice density $p(\boldsymbol{\tau}; x)$:

$$\mathbb{E}_{\omega \sim \mathring{p}(\cdot; x, \boldsymbol{\tau})} \left[\frac{1}{\xi(x, \boldsymbol{\tau}, \omega)}\right] = \frac{1}{p(\boldsymbol{\tau}; x)} \tag{4.4}$$

Indeed, any non-negative and unbiased estimator of $p(\boldsymbol{\tau}; x)$ can be used as the basis of an encapsulated randomness construction. The inputs to the estimator are x and $\boldsymbol{\tau}$. The internal randomness used in the estimator is ω and $\mathring{q}(\cdot; x, \boldsymbol{\tau})$ is the distribution on this randomness, and $\xi(x, \boldsymbol{\tau}, \omega) \geq 0$ is the estimate returned for randomness ω. Then, the density $\mathring{p}$ in the encapsulated randomness construction is:

$$\mathring{p}(\omega; x, \boldsymbol{\tau}) := \xi(x, \boldsymbol{\tau}, \omega)\mathring{q}(\omega; x, \boldsymbol{\tau})/p(\boldsymbol{\tau}; x) \tag{4.5}$$

which is a normalized probability density because

$$\int_\Omega \mathring{p}(\omega; x, \boldsymbol{\tau})\nu(d\omega) = \int_\Omega \xi(x, \boldsymbol{\tau}, \omega)\mathring{q}(\omega; x, \boldsymbol{\tau})/p(\boldsymbol{\tau}; x)\nu(d\omega) = p(\boldsymbol{\tau}; x)/p(\boldsymbol{\tau}; x) = 1 \tag{4.6}$$

4.5.1 Extending the data type operations with encapsulated randomness

We now define the data type operations for generative functions with extended randomness.

Simulate Sample $\boldsymbol{\tau} \sim p(\cdot; x)$ and $\omega \sim \mathring{p}(\cdot; x, \boldsymbol{\tau})$. Return the trace $\mathbf{t} := (\mathcal{P}, x, \boldsymbol{\tau}, \omega)$

Generate Sample $\boldsymbol{\upsilon} \sim q(\cdot; x, \boldsymbol{\sigma})$ and then set $\boldsymbol{\tau} := \boldsymbol{\upsilon} \oplus (\boldsymbol{\sigma}|_B)$ and $p(\boldsymbol{\tau}; x) > 0$ then sample $\omega \sim \mathring{q}(\cdot; x, \boldsymbol{\tau})$. Return the trace $\mathbf{t} := (\mathcal{P}, x, \boldsymbol{\tau}, \omega)$ and $\log w := \log(\xi(x, \boldsymbol{\tau}, \omega)/q(\boldsymbol{\upsilon}; x, \boldsymbol{\sigma}))$.

Logpdf Given trace $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau}, \omega)$, return $\log \xi(x, \boldsymbol{\tau}, \omega)$.

Update Given $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau}, \omega)$, compute $(\boldsymbol{\tau}', \boldsymbol{\sigma}') = h_{\text{update}}(\boldsymbol{\tau}, \boldsymbol{\sigma})$ as before, then sample $\omega' \sim \mathring{q}(\cdot; x', \boldsymbol{\tau}')$. Return a new trace $\mathbf{t}' = (\mathcal{P}, x', \boldsymbol{\tau}, \omega')$ and the log weight $\log w := \log(\xi(x', \boldsymbol{\tau}', \omega')/\xi(x, \boldsymbol{\tau}, \omega))$ The other return values are the same as previously.

Regenerate Given $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau}, \omega)$. As before let $\boldsymbol{\sigma} := \boldsymbol{\tau}|_{S^{\mathsf{c}}}$ and sample $\boldsymbol{\upsilon} \sim q(\cdot; x', \boldsymbol{\sigma})$ and let $\boldsymbol{\tau}' := \boldsymbol{\upsilon} \oplus (\boldsymbol{\sigma}|_B)$ for some B such that $p(\boldsymbol{\tau}'; x') > 0$; and let $\boldsymbol{\sigma}' := \boldsymbol{\tau}'|_{S^{\mathsf{c}}}$. But also sample $\omega' \sim \mathring{q}(\cdot; x, \boldsymbol{\tau}')$. Return a new trace $\mathbf{t}' = (\mathcal{P}, x', \boldsymbol{\tau}, \omega')$ and the log weight

$$\log w := \log \frac{\xi(x', \boldsymbol{\tau}', \omega') q(\boldsymbol{\upsilon}'; x, \boldsymbol{\sigma}')}{\xi(x, \boldsymbol{\tau}, \omega) q(\boldsymbol{\upsilon}; x', \boldsymbol{\sigma})} \tag{4.7}$$

The other return values are the same as previously.

Example: A mixture of normal distributions Consider a generative function that makes a single random choice at address a, which is distributed according to a mixture of an countably infinite collection of normal distributions with means μ_i and standard deviations σ_i and proportions π_i such that $\sum_{i=1}^{\infty} \pi_i = 1$:

$$p(\boldsymbol{\tau}) := \sum_{i=1}^{\infty} \pi_i \cdot p_{\text{norm}(\mu_i, \sigma_i)}(y) \text{ for } \boldsymbol{\tau} = \{a \mapsto y\} \text{ and zero otherwise} \tag{4.8}$$

(there are no arguments x). Consider $\Omega := \{1, 2, \ldots, \}$ and $\mathring{p}$ given by:

$$\mathring{p}(\omega; \boldsymbol{\tau}) := \frac{\pi_\omega \cdot p_{\text{norm}(\mu_\omega, \sigma_\omega)}(\boldsymbol{\tau}[a])}{\sum_{i=1}^{\infty} \pi_i \cdot p_{\text{norm}(\mu_i, \sigma_i)}(\boldsymbol{\tau}[a])} \tag{4.9}$$

and any proposal distribution $\mathring{q}(\omega; \boldsymbol{\tau})$ such that $\mathring{q}(\omega; \boldsymbol{\tau}) > 0$ for all $\omega \in \Omega$. Then, the choice density estimate is:

$$\xi(x, \boldsymbol{\tau}, \omega) := \frac{p(\boldsymbol{\tau}; x) \mathring{p}(\omega; x, \boldsymbol{\tau})}{\mathring{q}(\omega; x, \boldsymbol{\tau})} = \frac{\pi_\omega p_{\text{norm}(\mu_\omega, \sigma_\omega)}(\boldsymbol{\tau}[a])}{\mathring{q}(\omega; x, \boldsymbol{\tau})} \tag{4.10}$$

For example, for $\mathring{q}(\omega; x, \boldsymbol{\tau}) := \pi_\omega$ the choice density estimate simplifies to $p_{\text{norm}(\mu_\omega, \sigma_\omega)}(\boldsymbol{\tau}[a])$. Within $\mathcal{P}.\text{GENERATE}(x, \boldsymbol{\tau})$ where $\boldsymbol{\tau} = \{a \to y\}$, this generative function will sample an index ω from the distribution π_ω, resulting in trace $\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau}, \omega)$. Subsequently, $\mathbf{t}.\text{LOGPDF}()$ will return $\log p_{\text{norm}(\mu_\omega, \sigma_\omega)}(y)$. Within $\mathcal{P}.\text{SIMULATE}(x)$, we sample ω from a discrete probability distribution with probability π_ω and then we sample $y \sim p_{\text{norm}(\mu_\omega, \sigma_\omega)}(\cdot)$, and set $\boldsymbol{\tau} := \{a \mapsto y\}$. Note that this is observationally equivalent to sampling $\boldsymbol{\tau} \sim p(\cdot; x)$ and then $\omega \sim \mathring{p}(\cdot; x, \boldsymbol{\tau})$. A subsequent call to $\mathbf{t}.\text{LOGPDF}()$ will also return $\log p_{\text{norm}(\mu_\omega, \sigma_\omega)}(y)$.

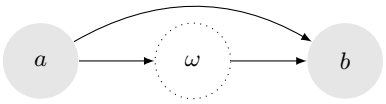
Observational equivalence for sampling encapsulated randomness in simulate Typically we construct generative functions with encapsulated randomness such that it is possible to sample $\omega \sim \mathring{q}(\cdot; x, \boldsymbol{\tau})$ efficiently but where it is not possible to sample $\omega \sim$

$\mathring{p}(\cdot; x, \boldsymbol{\tau})$ efficiently. The example above illustrates this—sampling from $\mathring{p}$ using a standard approach of computing the normalizing constant in Equation (4.9) is not possible in general because it requires summing over an infinite set of ω. While it is not possible to sample ω from $\mathring{p}$ for a specific value of $\boldsymbol{\tau}$, it is often possible to efficiently sample $(\boldsymbol{\tau}, \omega)$ *jointly* such that their joint distribution is identical to that of $\boldsymbol{\tau} \sim p(\cdot; x)$ and $\omega | \boldsymbol{\tau} \sim \mathring{p}(\cdot; x, \boldsymbol{\tau})$. As in the example above, this is often achieved by sampling ω first from the marginal distribution, and then sampling $\boldsymbol{\tau}$ from its conditional distribution given ω.

The encapsulated randomness densities need not be individually evaluable Note that it is not required to efficiently *evaluate* either $\mathring{q}(\omega; x, \boldsymbol{\tau})$ or $\mathring{p}(\omega, x, \boldsymbol{\tau})$ and implementations of the data types often do not compute these values internally. Instead, they only need to compute the choice density estimate in Equation (4.2) efficiently, which due to cancellation of factors, can be straightforward when evaluating $\mathring{q}(\omega; x, \boldsymbol{\tau})$ and $\mathring{p}(\omega, x, \boldsymbol{\tau})$ is not. This property gives the data types substantial flexibility that is exercised in the following construction.

4.5.2 Untraced random choices

Modeling languages like Gen's DML can run general-purpose code, including code for which the source code is not available and is not analyzed or instrumented by the modeling language compiler (for example, calling a Julia function). What if this black-box code code is stochastic? Encapsulated randomness can be used to formally model this setting. We call random choices that are made by a probabilistic program but not annotated with a choice dictionary address *untraced random choices*. Probabilistic programs in languages like Gen's DML can still define valid generative functions, even in the presence of untraced randomness choices, by modeling all of the untraced random choices as part of the encapsulated randomness ω, and by using forward sampling to define $\mathring{q}(\cdot; x, \boldsymbol{\tau})$. For example, consider a generative function composed of (i) sampling a random choice at address a from a distribution $p_1(\cdot)$, (ii) passing the value for a into a call to stochastic black-box code that makes untraced random choices ω sampled from $p_2(\cdot; \boldsymbol{\tau}[a])$, and (iii) sampling another random choice at address b from a distribution $p_3(\cdot; \boldsymbol{\tau}[a], \omega)$ that depends on the result of the black-box code and the value for a:



Suppose, for the sake of simpler notation, that a, ω, and b are discrete. Then, the probability distribution on choice dictionaries for the resulting generative function is:

$$p(\boldsymbol{\tau}; x) := p_1(\boldsymbol{\tau}[a]) \sum_{\omega \in \Omega} p_2(\omega; \boldsymbol{\tau}[a]) p_3(\boldsymbol{\tau}[b]; a, \omega) \tag{4.11}$$

Let $\mathring{p}$ be the conditional distribution of ω given a and b:

$$\mathring{p}(\omega; x, \boldsymbol{\tau}) := \frac{p_2(\omega; \boldsymbol{\tau}[a]) p_3(\boldsymbol{\tau}[b]; a, \omega)}{\sum_{\omega' \in \Omega} p_2(\omega'; \boldsymbol{\tau}[a]) p_3(\boldsymbol{\tau}[b]; a, \omega')} \tag{4.12}$$

Let $\mathring{q}$ be the distribution of forward sampling ω:

$$\mathring{q}(\omega; x, \boldsymbol{\tau}) := p_2(\omega; \boldsymbol{\tau}[a]) \tag{4.13}$$

Then, the choice density estimate (Equation (4.2)) simplifies to:

$$\xi(x, \boldsymbol{\tau}, \omega) = p_1(\boldsymbol{\tau}[a]) p_3(\boldsymbol{\tau}[b]; \boldsymbol{\tau}[a], \omega) \tag{4.14}$$

The DML source code below shows an example of this pattern, when `foo_simulator` is a black-box stochastic Julia function.

```
@gen function with_untraced()
    a ~ bernoulli(0.5)
    result = foo_simulator(a)
    b ~ bernoulli(if (result > 0.6) 0.1 else 0.9)
end
```

Note that for DML code to define a valid generative function, untraced random choices cannot influence control flow, because this can result in densities on choice dictionaries that are not structured (Definition 2.1.3).

4.5.3 Pseudo-marginal Monte Carlo methods and encapsulation

The encapsulated randomness of a generative function can be constructed from estimators of $p(\boldsymbol{\tau}; x)$, including Monte Carlo methods like importance sampling, annealed importance sampling, and more generally sequential Monte Carlo. When the resulting generative function is used as a generative model (or a component in a generative model) with the inference procedures of Section 3, Section 4.2, and Section 4.3, we recover many *pseudo-marginal* Monte Carlo algorithms [4] from the computational statistics literature, including particle Markov chain Monte Carlo [6]. These algorithms retain the asymptotic guarantees of regular Monte Carlo algorithms, but can be more efficient. Also, Gen's encapsulated randomness construction makes the auxiliary variable arguments that form the basis of these algorithms readily apparent.

Example: Pseudo-marginal Metropolis-Hastings The class of Metropolis-Hastings kernels can be generalized into the class of pseudo-marginal Metropolis-Hastings kernels, by using unbiased estimates of the data likelihood in place of the true data likelihood when computing the acceptance probability in the Metropolis-Hastings algorithm. Implementing Metropolis-Hastings (Section 3.4.2) using a generative function ($\mathcal{P}$) for the model that employs encapsulated randomness results in a pseudo-marginal Metropolis-Hastings kernel. Recall the acceptance probability from the Metropolis-Hastings procedure in Algorithm 5:

$$\min\left\{1, \exp(\log w - \mathbf{s}'.\text{LOGPDF}() + \mathbf{s}.\text{LOGPDF}())\right\} \tag{4.15}$$

where $\log w$ is the log weight returned by UPDATE and $\mathbf{s}$ and $\mathbf{s}'$ are the forward and backward proposal traces, respectively. When the model generative function uses encapsulated

randomness, the acceptance probability is:

$$\min\left\{1, \frac{\xi(x, \boldsymbol{\tau}', \omega')q(\boldsymbol{\sigma}; x, \boldsymbol{\tau}')}{\xi(x, \boldsymbol{\tau}, \omega)q(\boldsymbol{\sigma}'; x, \boldsymbol{\tau})}\right\} = \min\left\{1, \frac{p(\boldsymbol{\tau}'; x)\mathring{p}(\omega'; x, \boldsymbol{\tau}')q(\boldsymbol{\sigma}; x, \boldsymbol{\tau}')\mathring{q}(\omega; x, \boldsymbol{\tau})}{p(\boldsymbol{\tau}; x)\mathring{p}(\omega; x, \boldsymbol{\tau})q(\boldsymbol{\sigma}'; x, \boldsymbol{\tau})\mathring{q}(\omega'; x, \boldsymbol{\tau}')}\right\} \quad (4.16)$$

(here, p is the density on choice dictionaries $\boldsymbol{\tau}$ for the model generative function $\mathcal{P}$, q is the density on choice dictionaries $\boldsymbol{\sigma}$ for the proposal generative function $\mathcal{Q}$ and $\mathring{q}$ is the density on encapsulated randomness for $\mathcal{P}$). It is straightforward to see that this is a regular Metropolis-Hasting kernel but on an extended state space that includes $\boldsymbol{\tau}$ as well as auxiliary variables ω. Similar auxiliary variable arguments can be used to justify various Monte Carlo algorithms. When the estimator of $p(\boldsymbol{\tau}; x)$ is constructed from a sequential Monte Carlo (SMC) algorithm, pseudo-marginal Metropolis-Hastings reduces to the particle marginal Metropolis-Hastings [6] (PMMH) algorithm. The encapsulated randomness ω then contains a *particle system* of a sequential Monte Carlo algorithm, which includes all particles at all time steps and all of the ancestor choices, and $\xi(x, \boldsymbol{\tau}, \omega)$ is the (unbiased) SMC estimate of $p(\boldsymbol{\tau}; x)$ that results from a particle system ω, $\mathring{q}(\cdot; x, \boldsymbol{\tau})$ is the distribution on the particle system that represents the sampling process within SMC, and $\mathring{p}(\cdot; x, \boldsymbol{\tau})$ is the distribution on the particle system that is sampled from during the *conditional SMC update* [6] given retained particle $\boldsymbol{\tau}$.

It is possible to implement PMMH using Gen by (i) implementing the SMC estimator using Gen's inference procedures for SMC (e.g. Section 3.5), and then (ii) constructing a generative function representing part of the generative model that internally uses the resulting SMC estimator, and (iii) applying the Metropolis-Hastings procedure (Section 3.4.2) to the generative function for the resulting generative model.

4.5.4 Using encapsulated randomness inside proposal distributions

The pseudo-marginal constructions introduced in the previous section use encapsulated randomness within the generative function $\mathcal{P} = (X, Y, p, f, q, \mathring{p}, \mathring{q})$ that represents the *model*. These algorithms invoke the operations $\mathcal{P}$.GENERATE, $\mathcal{P}$.UPDATE, and $\mathcal{P}$.REGENERATE which all sample the encapsulated randomness from $\mathring{q}$, and do not invoke $\mathcal{P}$.SIMULATE, which samples the encapsulated randomness from $\mathring{p}$. Consider instead the setting when $\mathcal{P}$ is used as a *proposal* and not the model. It is also possible to employ encapsulated randomness in a generative function that is used as a proposal within importance sampling, SMC, or MCMC algorithms. These algorithms internally invoke $\mathcal{P}$.SIMULATE when $\mathcal{P}$ is the proposal, which involves sampling from $\mathring{p}$ and not $\mathring{q}$. Like pseudo-marginal algorithms, the resulting algorithms can also be justified as instances of the original algorithm templates, but on an extended space of auxiliary variables [30]. Note that the desiderata for $\mathring{q}$ and $\mathring{p}$ are somewhat modified in this setting. For example, in importance sampling when the model uses an encapsulated randomness with densities $\mathring{p}$ and $\mathring{q}$, the the efficiency of the resulting algorithms depends on $\mathrm{D}_{\mathrm{KL}}(\mathring{p}(\cdot; x, \boldsymbol{\tau})||\mathring{q}(\cdot; x, \boldsymbol{\tau}))$ because $\mathring{p}$ extends the model density and $\mathring{q}$ extends the proposal density, and because importance sampling efficiency depends on the KL divergence from the model to the proposal [21]. When the proposal in importance sampling uses encapsulated randomness with densities $\mathring{p}$ and $\mathring{q}$, the efficiency of the resulting algorithms depends on $\mathrm{D}_{\mathrm{KL}}(\mathring{q}(\cdot; x, \boldsymbol{\tau})||\mathring{p}(\cdot; x, \boldsymbol{\tau}))$, because $\mathring{p}$ extends the proposal density

and $\mathring{q}$ extends the model density.

4.6 Related Work

Selection MH and blocked Gibbs sampling The interface presented by selection MH, where users select a set of variables to be updated conditioned on the values of other variables, is similar to that of Gibbs sampling. The convenience of Gibbs sampling as a mathematical framework for constructing samplers popularized the use of MCMC for Bayesian networks [59] and motivated the development of the BUGS probabilistic programming system [48]. The utility of Gibbs sampling is furthered by use of *blocking* [63], in which groups of random variables are updated jointly, either to improve mixing, or to exploit parallelism. However, selection MH is more general than blocked Gibbs sampling—it is well-defined for models with stochastic structure, and the internal proposal may or may not sample from the exact conditional distribution. Note that selection MH does not itself perform Gibbs sampling—it relies on either the modeling language implementation or the user to derive and implement the conditional distributions, when possible.

Selection MH and subproblem MCMC in Venture The inference language of Venture [79] also includes an operator that applies a Metropolis-Hastings kernel based on forward simulation, and an operator for Gibbs sampling, to a subset of random choices defined by a *subproblem selection* [80]. Later work carefully formalized a more general notion of subproblem MCMC [55] and characterized conditions for which compositional MCMC algorithms based on subproblem selections are asymptotically sound. While these works use a representation of a trace that includes dependency structure, and defines a subproblem as the extent of a change including random choices that may go out of existence, Gen is formalized using probability distributions on choice dictionaries without dependence structure, and the *selection* includes the set of random choices that the user wants to change, corresponding to the 'minimal subproblem' in Venture. The encapsulation of these details within the trace data type allows for the internal proposal distribution to be overridden, generating heterogeneous proposal distributions that combine generic elements like forward simulation with elements that specialize to structure in sub-models. This is not supported by Venture. It may be possible to use implementations and analyses based on traces with dependence structure in the implementation of specific modeling languages that are complementary to Gen's existing general-purpose modeling languages.

Forward sampling as an automatic proposal Forward sampling has long been used in Bayesian networks as a default proposal mechanism within importance sampling, which results in the likelihood weighting inference algorithm [66]. A likelihood weighting algorithm for a more flexible class of open-universe generative models was later introduced for the BLOG system [84]. The use of forward sampling as a generic proposal mechanism for Monte Carlo algorithms in universal probabilistic programming systems dates at least to Church [51], and has been the central MCMC proposal mechanism for all universal sampling-based probabilistic programming systems since [52, 79, 130, 40], and the only

trans-dimensional MCMC mechanism in these systems. While Gen's SML and DML modeling languages use forward simulation for their generic internal proposal mechanisms, this proposal is only provided as a *default* for users of Gen—users are able to replace forward simulation with their own custom proposals as more performance is needed. Furthermore, Gen allows users to seamlessly combine a combination of forward simulation and other more efficient proposal mechanisms, either by implementing custom internal proposals for fragments of generative models, or by combining external with internal proposals (within sequential Monte Carlo and importance sampling). Finally, Gen is intended to be extended with new modeling languages in which fragments of models with specialized structure implement their more efficient internal proposals.

Encapsulation of inference logic with models in probabilistic programming There is relatively little prior work on encapsulation of inference capabilities within components of generative models in the context of probabilistic programming systems. The implementation of the Venture system [79] included a foreign function interface (FFI) for procedures used in models. Although the intent of Venture's FFI is similar to that of Gen's abstract trace data type (and served as an inspiration for Gen), Venture's FFI provides a relatively weak form of encapsulation. Foreign functions pass expressions to Venture's main interpreter for evaluation, and changes to control flow within a foreign function require passing data back to the Venture interpreter. Venture's FFI is also significantly more complex—even forward execution requires back-and-forth communication between Venture's interpreter and the foreign function; in Gen the inference code simply calls SIMULATE. Also, whereas Venture allows a foreign function to possess an encapsulated ergodic MCMC kernel that can only be applied atomically, Gen allows generative functions to possess internal proposals, which can be combined seamlessly with the internal proposals from other parts of the model or the user's external proposals; and these proposals can be used in the context of both MCMC and SMC algorithms.

Encapsulated randomness and pseudo-marginal methods Cusumano-Towner and Mansinghka [29] described an early version of the encapsulated randomness construction of Section 4.5, and in a later paper [30] we applied it to using probabilistic programs as proposal distributions within Monte Carlo algorithms. We also applied a simplified version of the encapsulated randomness construction to estimating the KL divergence between the output distribution of two probabilistic programs (the paper introduces the algorithm as a technique for estimating the divergence between KL two inference samplers, but it is applicable to estimating the KL divergence between any two generative functions) [26]. The MetaPPL language [74] elaborates on the internal proposal family construction of this chapter and distinguishes between two classes of internal proposals—one for use when the generative function is used as a proposal and the other for use when the generative function is used as a model.

The encapsulated randomness construction allows a number of auxiliary variable algorithms described in the computational statistics literature to be implemented in a modular and compositional manner using probabilistic programming languages. Pseudo-marginal

MCMC [4] is a broad family of MCMC algorithms that use unbiased estimates of likelihoods in place of intractable likelihoods within acceptance probability calculations. Storvik [117] describe a general family of auxiliary variable constructions for Metropolis-Hastings that includes many that can be represented using encapsulated randomness and generative functions. Particle marginal Metropolis-Hastings [6] is a class of pseudo-marginal MCMC methods that use sequential Monte Carlo algorithms to estimate intractable densities or likelihoods within the acceptance probability.

Chapter 5

Compiling Generative Function and Trace Data Types from Probabilistic Modeling Code

The previous chapters have shown that a wide variety of inference algorithms can be implemented at a high-level of abstraction using generative functions and traces. This chapter focuses on how traces and generative functions are implemented. Generative functions are typically compiled from a modeling language so this chapter is primarily concerned with language design and implementation (although as we will see, generative functions can be constructed in other ways as well). Gen contains two built-in modeling languages that strike different tradeoffs between ease-of-use and performance: Gen's Dynamic Modeling Language (DML) is simple to implement and easy for users to learn, but is inefficient. Gen's Static Modeling Language (SML) has a more complex implementation, and is more complex to use, but enables significant asymptotic and constant-factor speedups for trace operations relative to DML. Note that generative functions expressed in both DML and SML can call arbitrary generative functions, including those compiled from other modeling languages. More specialized and domain-specific modeling languages allow for even further specialization and improved performance over SML. This chapter also introduces *generative function combinators* that express common patterns of repeated computation and conditional independence in models. The chapter ends with a discussion of how programmers can hand-implement generative functions and traces that are specialized to their model for further improvements in performance. Because each of these approaches to defining models generates the same abstract data types, programmers can easily move between approaches and dynamically adjust their balance between ease-of-use and performance. The lifecycle of modeling code, generative functions, and traces is shown figure 5-1.

5.1 The Dynamic Modeling Language compiler

The dynamic modeling language (DML) that was introduced in Section 2.2.1 is a Turing-universal probabilistic modeling language that is embedded in the Julia programming lan-

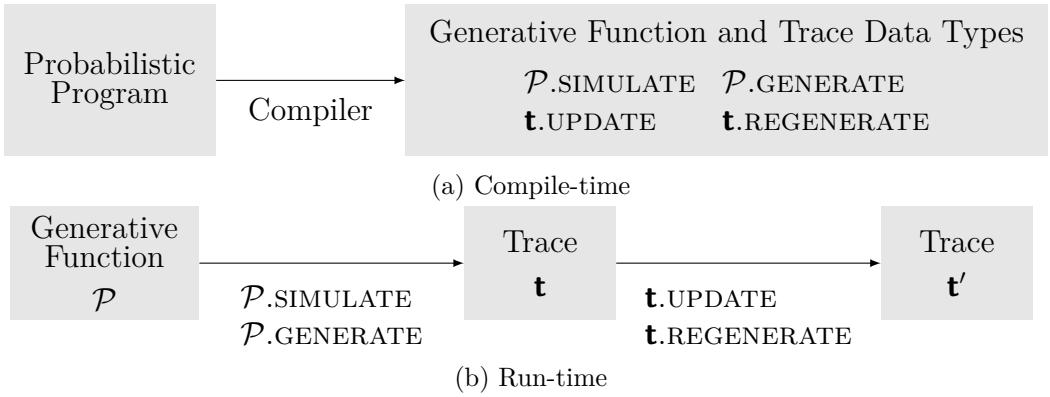


Figure 5-1: Lifecycle of probabilistic source code, generative functions, and traces

guage [12]. It includes all the control flow features of Julia itself, such as while loops and recursion. This section describes the implementations of the generative function and trace ADTs produced by the DML compiler. The compiler does not attempt to specialize the implementations of the data types to a particular model: The trace data structure is generic and based on Julia dictionaries, where keys are addresses of random choices or calls to other generative functions. All operations for the generative function and trace data types are implemented via non-standard interpretations of the Julia code in the body of the function definition. The non-standard interpretations are implemented by transforming the body of the generative function and replacing random choice expressions and generative function call expressions with calls to an effect handler that is specific to the operation being implemented. The language implementation performs no static dependency analysis or dynamic dependency tracking—every operation involves a full end-to-end execution of the body of the generative function. With the exception of random choice expressions, the body of the function is not instrumented, so the performance characteristics of the underlying Julia code are preserved (Julia type inference and Julia JIT compilation is performed on the transformed code). Therefore, its performance is asymptotically suboptimal for trace operations that involve only a subset of the random choices in the trace (e.g. updating the value of a small number of random choices, or requesting the gradient with respect to a small number of random choices). However, the DML is valuable because of (i) it provides a simple readable reference implementation of the generative function and trace data types, (ii) its similarity to Julia makes it very easy to learn and gives it predictable performance for new users of Gen, and (iii) it is expressive in that complex stochastic control flow can be specified more concisely than using the more performant modeling languages and constructs discussed later in this chapter. Finally, note that DML functions can invoke generative function compiled from other modeling languages, and vice versa, so the tradeoffs that it makes can be applied by users selectively to parts of their model.

5.1.1 Implementing generative functions and traces via effect handlers

The trace data type for DML generative functions is based on a dictionary that maps addresses to records that either contain a value and log probability density (for random choices), or an opaque trace data structure called a 'subtrace' (for calls to other generative functions, discussed in Section 5.1.2). The trace also contains the arguments, return value, and the log densities for each random choice and the total log densities for all the random choices made under each generative function call, and the total log density across all random choices. Each generative function and trace data type operation is implemented via a non-standard interpretation of the body of the function with effect handlers that perform specialized modifications to state depending on which operation is implemented.

Implementing the simulate and update operations Pseudocode for the DML implementations of the SIMULATE operation and the UPDATE operation are shown in Algorithm 17. This code does not show the the effect handlers for calling other generative functions, which will be discussed in Section 5.1.2. Each operation is implemented using two auxiliary procedures—one that initializes the state of the non-standard interpreter (SIMULATE-INIT, UPDATE-INIT) and another that implements an effect handler for random choices expressions (SIMULATE-HANDLER, UPDATE-HANDLER). The handlers take as input the state of the interpreter (`state`), the address of a random choice (a), and a probability distribution of a random choice (`dist`), do some computation and update the state of the interpreter, and then return a value for the random choice. The handlers are invoked when executing the body of the code with $\mathcal{P}$.EXEC, which takes the arguments to the generative function (x), the initial state of the interpreter (so it can be passed to handlers), and the handler that should be used. Running $\mathcal{P}$.EXEC serves to update the interpreter state, and also produces the return value of the generative function body (`ret`). Next, a trace of the generative function is constructed based on the state of the interpreter.

The state of the interpreter for SIMULATE includes a dictionary of random choice values ($\boldsymbol{\tau}$) and a dictionary of log probability densities ($\boldsymbol{\lambda}$). The effect handler for SIMULATE is straightforward—it simply samples the value of the random choice from the given distribution, stores the value in the dictionary $\boldsymbol{\tau}$, computes the log density and stores it in $\boldsymbol{\lambda}$, and then returns the value back to the running program. When constructing the trace, we use the notation $\mathbf{t} \leftarrow (\mathcal{P}, x, \boldsymbol{\tau})$ to denote the assignment of abstract semantics to the trace $\mathbf{t}$ (which, as defined in Section 2.3, consists of the generative function $\mathcal{P}$, the arguments x, and the dictionary of choices $\boldsymbol{\tau}$). We denote implementation-specific information that is recorded in the trace data structure using notation of the form $\mathbf{t}.\boldsymbol{\lambda} \leftarrow \texttt{state}.\boldsymbol{\lambda}$. This means that we store the dictionary of per-choice log densities in the trace. Similarly we store the return value of the generative function ($\mathbf{t}.y \leftarrow \texttt{ret}$); the implementation of $\mathbf{t}$.RETVAL simply returns the recorded value $\mathbf{t}.y$ and does not re-run the body of the function.

The interpreter for UPDATE is more complex. The state includes the dictionary that will contain all choices in the new trace after EXEC returns ($\boldsymbol{\tau}'$), the new dictionary of log densities ($\boldsymbol{\lambda}'$), the dictionary of new random choice values ($\boldsymbol{\sigma}$), and a dictionary that will be the dictionary $\boldsymbol{\sigma}'$ that is the third return value of UPDATE after EXEC returns. The effect handler UPDATE-HANDLER first checks if there is an entry in $\boldsymbol{\sigma}$ for the address a. If there is,

then the value for the choice is set to $\boldsymbol{\sigma}[a]$; otherwise the value is taken from the previous trace ($\boldsymbol{\tau}[a]$). If neither $\boldsymbol{\sigma}$ nor $\boldsymbol{\tau}$ contains an entry for a, then there is an error, because there is no dictionary $\boldsymbol{\tau}' = \boldsymbol{\sigma} \oplus (\boldsymbol{\tau}|_B)$ for any B for which $p(\boldsymbol{\tau}'; x') > 0$ and UPDATE is undefined ($h_{\text{update}}(\boldsymbol{\tau}, \boldsymbol{\sigma}) = \bot$). (If there were such a $\boldsymbol{\tau}'$ then the execution of the program that draws values for random choices from $\boldsymbol{\tau}'$ would result in the same sequence of program states as an execution that draws values from $\boldsymbol{\sigma} \oplus \boldsymbol{\tau}$.) If there is a new value ($\boldsymbol{\sigma}[a]$) *and* a previous value ($\boldsymbol{\tau}[a]$), then the previous value is stored in $\boldsymbol{\sigma}'[a]$. After EXEC returns, we add the old values ($\boldsymbol{\tau}[a]$) for any addresses that were not visited during the execution to $\boldsymbol{\sigma}'$. Intuitively, these addresses are removed from the trace. Finally, we compute the previous and new total log densities and the difference is $\log w$ (the actual implementation stores the total log density in the trace, but we omit this for simplicity). Note that the DML implementation of UPDATE does not use the change hint δ_X for the change from x and x', and returns the non-informative change hint $\delta_Y = \bot$ for the change from the previous return value ($\mathbf{t}.y$) to the new return value ($\mathbf{t}'.y$).

Algorithm 17 Pseudocode for DML implementations of simulate and update operations

```
procedure SIMULATE-INIT( )
    τ ← {}                          ▷ Initialize dictionary for choices
    λ ← {}    ▷ Initialize dictionary for log densities
    state ← (τ, λ)
    return state
end procedure

procedure SIMULATE-HANDLER(state, a, dist)
    val ∼ dist
    state.τ[a] ← val
    state.λ[a] ← log p_dist(val)
    return val
end procedure

procedure P.SIMULATE(x)
    state ← SIMULATE-INIT()
    ret ← P.EXEC(x, state, SIMULATE-HANDLER)
    t ← (P, x, state.τ)
    t.λ ← state.λ
    t.y ← ret
    return t
end procedure

procedure UPDATE-INIT(t = (P, x, τ), σ)
    σ' ← {}; τ' ← {}                ▷ Initialize dictionaries
    λ' ← {}  ▷ Init. dictionary for new log densities
    state ← (τ, τ', λ', σ, σ')
    return state
end procedure

procedure UPDATE-HANDLER(state, a, dist)
    if a ∈ A_state.σ then
        val ← state.σ[a]
        if a ∈ A_state.τ then
            state.σ'[a] ← state.τ[a]
        end if
    else if a ∈ A_state.τ then
        val ← state.τ[a]
    else
        error        ▷ h_update(state.τ, state.σ) = ⊥
    end if
    state.λ'[a] ← log p_dist(val)
    return val
end procedure

procedure (t = (P, x, τ)).UPDATE(x', δ_X, σ)
    state ← UPDATE-INIT(t, σ)
    ret ← P.EXEC(x', state, UPDATE-HANDLER)
    t' ← (P, x', state.τ'); t'.λ' ← state.λ'
    t'.y ← ret
    for a ∈ A_state.τ \ A_state.τ' do
        state.σ'[a] ← state.τ[a]
    end for
    ℓ1 ← Σ_{a ∈ A_t.λ} (t.λ[a])
    ℓ2 ← Σ_{a ∈ A_state.λ'} (state.λ'[a])
    return (t', ℓ2 − ℓ1, state.σ', ⊥)
end procedure
```

Implementing an internal proposal family using forward sampling Recall that the internal proposal family of a generative function (Section 4.1) is a family of probability distributions on choice dictionaries $\boldsymbol{v}$ (with densities $q(\boldsymbol{v}; x, \boldsymbol{\sigma})$) that 'fill in' a partially specified set of random choices $\boldsymbol{\sigma}$ to form a complete trace. That is, merging $\boldsymbol{v}$ with some subset of $\boldsymbol{\sigma}$ results in a dictionary $\boldsymbol{\tau}' = \boldsymbol{v} \oplus (\boldsymbol{\sigma}|_B)$ that encodes a possible complete execution of the generative function ($p(\boldsymbol{\tau}'; x) > 0$). The DML uses forward sampling for its internal proposal family. In particular, the DML implementations of the GENERATE and REGENERATE (Algorithm 18) perform forward sampling.

First, consider GENERATE, which is a more straightforward implementation of forward sampling. The state for the interpreter for this operation stores the values and log densities for each choice ($\boldsymbol{\tau}$ and $\boldsymbol{\lambda}$), and an accumulator (ℓ) for the log weight ($\log w$). The handler for random choices (GENERATE-HANDLER) first checks if address is provided in the input dictionary $\boldsymbol{\sigma}$. If it is, it retrieves the value $\boldsymbol{\sigma}[a]$ and increments the log weight. Otherwise, it samples the value from the given distribution, and does not increment the log weight.

The DML implementation of REGENERATE is more complex. The dictionary $\boldsymbol{\sigma}$ used by the internal proposal is never explicitly constructed. Recall that it is derived from the previous trace via $\boldsymbol{\sigma} = \boldsymbol{\tau}|_{B^\mathsf{c}}$ (that is, the set of all entries in the previous trace, less those in the selection B). Note that $A_{\boldsymbol{\sigma}} = A_{\boldsymbol{\tau}} \cap B^\mathsf{c}$ and therefore $A^\mathsf{c}_{\boldsymbol{\sigma}} = A^\mathsf{c}_{\boldsymbol{\tau}} \cup B$. The handler (REGENERATE-HANDLER) checks if the address is in $A^\mathsf{c}_{\boldsymbol{\tau}} \cup B$. If it is, then it samples the value. Otherwise, it retrieves the value from $\boldsymbol{\tau}$ and increments the log weight, using the value of the log density for the address from the *previous trace* (`state.`$\boldsymbol{\lambda}$). Note that although the value of the choice is the same in the previous and new trace, its density may be different because the distribution (`dist`) may have changed. Note that unlike in the implementation of UPDATE, after the execution $\mathcal{P}$.EXEC is finished, the set of choices that were *not visited* is not referenced when computing the log weight. This is possible because the contributions of the terms for these choices in $\log p(\boldsymbol{\tau}; x)$ and $\log q(\boldsymbol{v}'; x, \boldsymbol{\sigma}')$ in Equation (4.1) cancel.

Constructing generative functions with supportive probability densities Recall that REGENERATE is only supported for generative functions whose probability densities are *supportive* (Definition 2.1.7). It is possible to guarantee that the probability densities $p(\cdot; x)$ for generative functions defined in DML (and SML) are supportive by ensuring that the support of the probability density at each address a is V_a for *all executions that sample a choice at the address*. For example, consider the following two DML generative functions:

```
@gen function not_supportive(b)
    a ~ uniform_discrete(1, 10)
    b ~ uniform_discrete(1, a)
end
```

```
@gen function supportive()
    a ~ uniform_discrete(1, 10)
    p = [if (i <= a) 0.9/a else 0.1/(10-a) end
         for i in 1:10]
    b ~ categorical(p)
end
```

The distribution `uniform_discrete`(l, u) where $l \leq u$ are integers is a discrete probability distribution on the integers $\{l, \ldots, u\}$. The distribution `categorical`$(\mathbf{p})$ where $\mathbf{p} \in \mathbb{R}^k$ is a normalized vector of probabilities is a discrete probability distributions on integers $\{1, \ldots, k\}$. The generative function defined on the left is not supportive because the support

at address `b` is not constant (e.g. the support is $\{1, 2\}$ for $\boldsymbol{\tau}_1 = \{a \mapsto 2, b \mapsto 2\}$ and is $\{1, 2, 3\}$ for $\boldsymbol{\tau}_2 = \{a \mapsto 3, b \mapsto 2\}$ and $p(\boldsymbol{\tau}_1) > 0$ and $p(\boldsymbol{\tau}_2) > 0$. The generative function on the right is supportive, because the support at address a is $\{1, \ldots, 10\}$ and the support at address b is $\{1, \ldots, 10\}$ for all possible executions. Note that as in this example we can approximate a non-supportive generative function by a supportive one by extending the support but placing low probability mass on regions of the space that had zero support. Intuitively, to see the relationship between supportive densities on choice dictionaries and constant support for each address, consider running the implementation of GENERATE in Algorithm 18 for a generative function where the support of each address a is constant (V_a) in all executions, and producing a trace with choices $\boldsymbol{\tau}$. Then $\boldsymbol{\tau}$ has $\boldsymbol{\tau} \sim \boldsymbol{\sigma}$ and positive density ($p(\boldsymbol{\tau}) > 0$) since the density is the product of individual densities $p_{\texttt{dist}}(\texttt{val})$, which are all necessarily positive.

Algorithm 18 Pseudocode for DML generate and regenerate operations

procedure GENERATE-INIT($\boldsymbol{\sigma}$)
  $\boldsymbol{\tau} \leftarrow \{\}$ ▷ Initialize dictionary for choices
  $\boldsymbol{\lambda} \leftarrow \{\}$ ▷ Initialize dictionary for log densities
  $\ell \leftarrow 0$ ▷ Initialize log weight
  $\texttt{state} \leftarrow (\boldsymbol{\tau}, \boldsymbol{\lambda}, \boldsymbol{\sigma}, \log w)$
  return state
end procedure

procedure GENERATE-HANDLER(state, a, dist)
  if $a \in A_{\texttt{state}.\boldsymbol{\sigma}}$ **then**
    $\texttt{val} \leftarrow \texttt{state}.\boldsymbol{\sigma}[a]$
    $\texttt{state}.\ell \leftarrow \texttt{state}.\ell + \log p_{\texttt{dist}}(\texttt{val})$
  else
    $\texttt{val} \sim \texttt{dist}$
  end if
  $\texttt{state}.\boldsymbol{\tau}[a] \leftarrow \texttt{val}$
  $\texttt{state}.\boldsymbol{\lambda}[a] \leftarrow \log p_{\texttt{dist}}(\texttt{val})$
  return val
end procedure

procedure $\mathcal{P}$.GENERATE(x, $\boldsymbol{\sigma}$)
  $\texttt{state} \leftarrow$ GENERATE-INIT($\boldsymbol{\sigma}$)
  $\texttt{ret} \leftarrow \mathcal{P}.\text{EXEC}(x, \texttt{state}, \text{GENERATE-HANDLER})$
  $\mathbf{t} \leftarrow (\mathcal{P}, x, \texttt{state}.\boldsymbol{\tau})$
  $\mathbf{t}.\boldsymbol{\lambda} \leftarrow \texttt{state}.\boldsymbol{\lambda}$
  $\mathbf{t}.y \leftarrow \texttt{ret}$
  $\log w \leftarrow \texttt{state}.\ell$
  return $(\mathbf{t}, \log w)$
end procedure

procedure REGENERATE-INIT($\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau})$, B)
  $\boldsymbol{\tau}' \leftarrow \{\}$ ▷ Init. dictionary for new choices
  $\boldsymbol{\lambda}' \leftarrow \{\}$ ▷ Init. dictionary for new log densities
  $\boldsymbol{\lambda} \leftarrow \mathbf{t}.\boldsymbol{\lambda}$ ▷ Previous log densities of choices
  $\ell \leftarrow 0$ ▷ Initialize log weight
  $\texttt{state} \leftarrow (\boldsymbol{\tau}, \boldsymbol{\tau}', \boldsymbol{\lambda}, \boldsymbol{\lambda}', \ell)$
  return state
end procedure

procedure REGENERATE-HANDLER(state, a, dist)
  if $(a \notin \texttt{state}.\boldsymbol{\tau}) \vee (a \in \texttt{state}.B)$ **then**
    $\texttt{val} \sim \texttt{dist}$
  else
    $\texttt{val} \leftarrow \texttt{state}.\boldsymbol{\tau}[a]$
    $\texttt{state}.\ell \leftarrow \texttt{state}.\ell + \log p_{\texttt{dist}}(\texttt{val}) - \texttt{state}.\boldsymbol{\lambda}[a]$
  end if
  $\texttt{state}.\boldsymbol{\tau}'[a] \leftarrow \texttt{val}$
  $\texttt{state}.\boldsymbol{\lambda}'[a] \leftarrow \log p_{\texttt{dist}}(\texttt{val})$
  return val
end procedure

procedure $(\mathbf{t} = (\mathcal{P}, x, \boldsymbol{\tau}))$.REGENERATE($x'$, δ_X, B)
  $\texttt{state} \leftarrow$ REGENERATE-INIT($\mathbf{t}, B$)
  $\texttt{ret} \leftarrow \mathcal{P}.\text{EXEC}(x', \texttt{state}, \text{REGENERATE-HANDLER})$
  $\mathbf{t}' \leftarrow (\mathcal{P}, x', \texttt{state}.\boldsymbol{\tau}')$
  $\mathbf{t}'.\boldsymbol{\lambda} \leftarrow \texttt{state}.\boldsymbol{\lambda}'$
  $\mathbf{t}'.y \leftarrow \texttt{ret}$
  $\log w \leftarrow \texttt{state}.\ell$
  return $(\mathbf{t}', \log w, \perp)$
end procedure

Implementing the gradient operation using automatic differentiation Recall that the gradient trace operation (Section 2.3) computes gradients of the log probability density of all of the random choices ($\log p(\boldsymbol{\tau}; x)$) with respect to the arguments to the generative function and the values $\boldsymbol{\tau}[a]$ of a selected subset ($a \in B$) of the continuous

random choices. The DML implementation of this operation uses reverse-mode automatic differentiation of the Julia code that forms the body of the function. Note that the derivative with respect to the value of a random choice at address a has two additive terms—the first is the gradient due to the probability density of the choice's own distribution, and the second is the gradient of any other choices (occurring later in the execution) whose distribution parameters depend on the value at a. The gradient operation also accepts an input gradient with respect to the return value, which enables gradient to be implemented compositionally, as described next.

5.1.2 Invoking generative functions

A DML generative can invoke other generative functions, regardless of how the generative function and trace data types for the other generative function were implemented. Recall that like random choices, generative function calls are also associated with an address a. Following the sequencing composition of generative functions in Section 2.1.5, all random choices made by the callee are given addresses by the caller that are based on a (in particular, of the form (a, b) where b is the address of the choice according to the callee generative function). The DML trace data structure stores the trace for each callee generative function and the address a for the call. These stored traces for callees are called *subtraces*. Each of the data type operations for the caller DML generative function have an effect handler that is analogous to the handlers in Algorithm 17 and Algorithm 18 but is invoked at generative function call sites instead of at random choice call sites. The effect handlers for most of the operations recursively invoke the same operation, but on the callee generative function or subtrace. For example, suppose we invoke $\mathcal{P}.\textsc{generate}(x, \boldsymbol{\sigma})$ for a generative function $\mathcal{P}$ that invokes another generative function $\mathcal{R}$. Suppose the GENERATE effect handler encounters a call to generative function $\mathcal{R}$ at address a with arguments z. Then the effect handler invokes $\mathcal{R}.\textsc{generate}(z, \boldsymbol{\sigma}|_B)$ where B contains all addresses of the form (a, b) for some b.

Example: Calling an SML generative function Consider the DML generative function `foo`, which invokes the SML generative function `bar` at address `c`:

```
@gen function foo()
    a ~ normal(0, 1)
    b ~ normal(a, 1)
    c ~ bar(b)
    d ~ normal(c, 1)
    e ~ normal(d, 1)
end
```

```
@gen (static) function bar(b)
    w ~ normal(b, 1)
    x ~ normal(w, 1)
    y ~ normal(x, 1)
    z ~ normal(z, 1)
    return z
end
```

All traces of `bar` contain dictionaries $\boldsymbol{\tau}$ with address $\{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$ and all traces of `foo` contain dictionaries $\boldsymbol{\tau}$ with addresses $\{\mathtt{a}, \mathtt{b}, (\mathtt{c}, \mathtt{w}), (\mathtt{c}, \mathtt{x}), (\mathtt{c}, \mathtt{y}), (\mathtt{c}, \mathtt{z}), \mathtt{d}, \mathtt{e}\}$. Consider the operation

$$\mathtt{foo}.\textsc{generate}(\_, \{(\mathtt{c}, \mathtt{y}) \mapsto 1.2, \mathtt{e} \mapsto -4.3\})$$

Suppose the execution of the body of `foo` reaches the third line with value b. Then the

effect handler for the call to `bar` at address `m` invokes `bar`.GENERATE($b, \{\texttt{y} \mapsto 1.2\}$).

Sharing a trace with callee DML generative function To avoid collisions between addresses of random choices, DML generative functions typically supply a distinct address namespace for each call to a generative function as above. However, it is sometimes convenient to use a 'flat' address space. This is permitted only for calls to other DML generative functions, in which case there is no trace associated with the called generative functions—its random choices and generative function calls are record in the same trace as the caller, making this a type of *trace inlining*. This makes it possible to write recursive DML generative functions that walk data structures using a custom flat addressing schemes for nodes in these data structures (e.g. associating an integer with each node in a binary tree), and avoid excessively hierarchical addresses (the *recurse* combinator, discussed later in this section automates this construction). Note that the user is responsible for indicating whether or not a call to another DML function should result in a new trace or should share the caller's trace, based on whether they provide an address or not.

Gradients and compositional automatic differentiation The DML implementation of GRADIENT first executes a (deterministic) forward pass of the function, reading values of random choices from the trace instead of sampling them, and recording deterministic values and random choices. During the forward pass, at each generative function call site, the GRADIENT operation is invoked for the trace of the callee, and the result is recorded for use by a later backward pass. Note that this result includes gradients with respect to the arguments to the callee, but may also include gradients with respect to random choices made by the callee. Finally, a backward pass is performed and the results of GRADIENT for each callee are used. Note that because GRADIENT is an abstract trace data type operation, the callee's implementation can use a completely different runtime system. For example, the callee may use the TensorFlow runtime to implement GRADIENT, whereas the caller may be written in DML and use automatic differentiation of Julia code.

5.2 The Static Modeling Language compiler

Static Modeling Language Bayesian networks are a classical graphical modeling formalism based on directed acyclic graph where nodes are random variables, where the edges in the graph indicate the static conditional independencies that hold for the model's joint distribution [66]. Like Bayesian networks, Gen's Static Modeling Language (SML) uses directed acyclic graphs, but generalizes Bayesian networks along several dimensions: (i) random variables can be sampled from discrete and/or continuous distributions, (ii) deterministic nodes that compute arbitrary functions are supported, (iii) nodes in the graph can represent generative functions (which themselves encode a joint distribution over a collection of random variables, including distributions where the number of variables is itself random). SML can also be seen as a subset of DML designed to support straightforward static analysis. The central restriction of SML relative to DML is that function bodies must be static single assignment basic blocks, and addresses must be literal symbols and not ar-

bitrary dynamically computed values as in DML. These restrictions enable static inference of the address space of random choices, which supports the generation of faster specialized trace data structures; as well as static information flow and conditional independence analysis that supports efficient implementations of UPDATE and other trace operations. SML is implemented in Julia but similar modeling languages can be implemented in other host languages as well.

Consider the example SML generative function below, which models the destination of an intelligent agent as they move around an environment with fixed obstacles. There a uniform prior distribution (`random_location`) on the destination location, and the speed of the agent is also unknown. The agent's trajectory is modeled in two stages: First, a Julia function (`plan_path`) is used to plan a path from the starting location of the agent (which is an argument and is assumed known) to the destination. Then, a generative function (`noise_model`) generates the trajectory of the agent given the agent's speed and the amount of noise in the trajectory, which are both latent variables. The `(static)` annotation in the function signature indicates that the SML compiler should be used.

```
@gen (static) function agent_model(start_location, num_time_steps)
  destination ~ random_location()
  path = plan_path(start_loc, destination)
  speed ~ uniform(0, 1)
  noise ~ uniform(0, 0.1)
  trajectory ~ noise_model(path, num_time_steps, speed, noise)
end
```

Graph-based intermediate representation The SML compiler generates an intermediate representation (IR) based on a directed acyclic graph with different nodes types for (i) arguments to the generative function, (ii) random choices, (iii) Julia expressions, and (iv) calls to generative functions. Random choice nodes include the distribution family, address, and references to other nodes that are parameters to the distribution family. Julia expression nodes include an anonymous Julia function that computes their value, and references to other nodes that represent the unbound symbols in the expression. Generative function call nodes include a reference to the generative function being called and the address under which the callee's random choices will be placed. Figure 5-2 diagrams the intermediate representation for the SML function `agent_model` above. The intermediate representation is used at compile-time to generate a trace data structure specialized to the SML generative function, using a Julia record type (`struct`) with entries for each random choice and entries that contain the subtrace for each callee generative function.

Implementing the generative function and trace operations Code for each of the generative function and trace operations is generated from the graph-based intermediate representation. While code for some of the operations is generated statically, exploiting the conditional independence structure that is recorded in the graph when implementing an operation requires analyzing the graph in the context of the arguments to the operation. Consider the UPDATE operation, which takes as inputs new arguments x', a change hint δ_X for the arguments, and a dictionary $\boldsymbol{\sigma}$ of new values for random choices. The set of

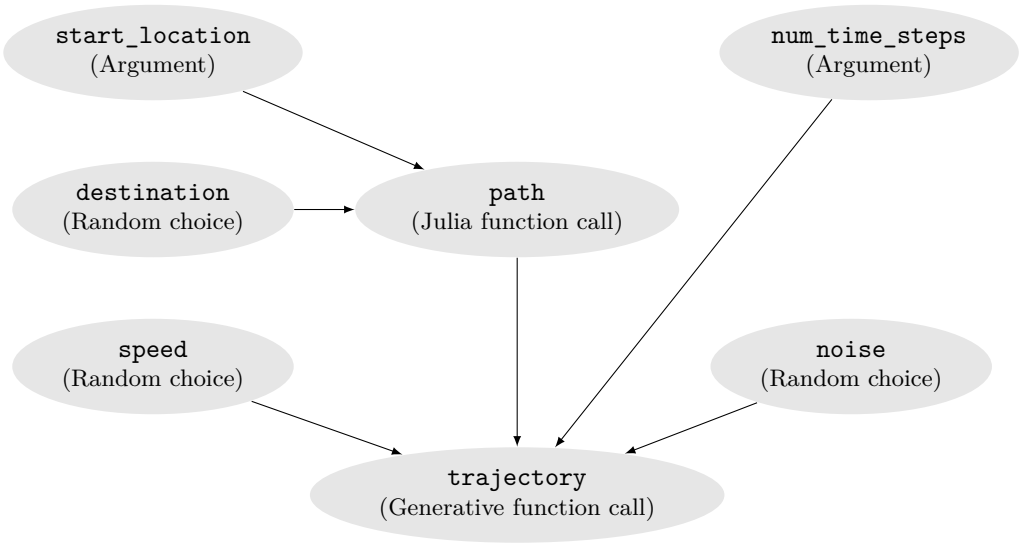


Figure 5-2: SML intermediate representation for a generative model

keys in $\boldsymbol{\sigma}$ and the change hint δ_X determine, together with the static graph structure, which statements in the program need to be re-evaluated and which do not. For example, for `agent_model`, if $\delta_X = \top$ (indicating that the arguments have not changed) and if $\boldsymbol{\sigma} = \{\texttt{noise} \mapsto 0.5\}$, it is not necessary to re-evaluate the Julia function `plan_path` because none of its inputs changed, but it is necessary to recursively invoke `update` on the subtrace of `noise_model` because one of its inputs changed. But if $\delta_X = \bot$, indicating that there may be an arbitrary change to the arguments, then both `plan_path` and `noise_model` need to be revisited. Determining what code is necessary to re-execute based on a joint analysis of the static graph, δ_X, and $A_{\boldsymbol{\sigma}}$,must happen dynamically. To reduce the overhead, we use a type system for δ_X and $A_{\boldsymbol{\sigma}}$ and only perform the analysis and code generation once for each operation applied to each unique combination of types. The types include the information that is necessary to perform the analysis—for δ_X the type indicates whether each argument may have changed or not. For $A_{\boldsymbol{\sigma}}$ the type indicates which subset of the the top-level addresses (in the example `destination`, `speed`, `noise`, or `trajectory`) are represented in $A_{\boldsymbol{\sigma}}$. The function caching and lookup based on these types are performed by Julia's JIT compiler, and the code generation is implemented using an extension to Julia's JIT compiler.

Example: Exploiting static model structure to avoid recomputation Consider an MCMC algorithm for inference over the destination given the observed trajectory, using the generative model `agent_model`. The algorithm uses a composite MCMC kernel that cycles through three Metropolis-Hastings MCMC kernels, each of which updates the value of one of the latent random choices (`destination`, `noise`, and `speed` respectively). The SML trace implementation, which is specialized to the conditional independence pattern of the model, avoids an unnecessary re-evaluation of the `plan_path` function when either the `noise` or `speed` variables are update (c). Table 5.1 shows a comparison of different implementations of this algorithm, for 1000 iterations of the composite cycle kernel. The `plan_path` procedure

is computationally intensive and dominates the running time of the algorithm. The SML implementation only re-executes `plan_path` during one of the three constituent kernels (the kernel for `destination`), whereas a DML implementation and a Turing [40] re-execute it within all three kernels because they do not exploit the independence of `speed` and `path` or `noise` and `path`. The SML gives a roughly threefold speedup over the DML implementation, and both implementations outperform the Turing implementation.

Implementation	**Running time (for 1000 steps)**
Gen (DML)	0.73s (±0.01)
Gen (SML)	2.50s (±0.08)
Turing	4.21s (±0.13)

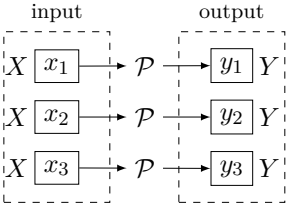
Table 5.1: Performance of different implementations of an MCMC inference algorithm

Propagating change hints through Julia code Recall that the change hint δ_X provides information about how the arguments of a generative function changed (from x to x') to the trace operations UPDATE and REGENERATE. As described above, the SML compiler uses an over-approximation of the change hint at compile time to help statically identify parts of the body that do not need to be re-executed. But the code that SML generates also propagates change hints themselves dynamically through the body of the function. For calls to generative functions, this is straightforward, because the trace operations of the callee consume a change hint and return a change hint for their return value, which can be propagated forward. Propagating change hints through Julia code requires performing a non-standard abstract interpretation of the Julia code. This is currently implemented via function overloading, but other implementation techniques are also possible. The ability to propagate change hints through Julia code to the inputs of callee generative functions is important for achieving asymptotically efficient updates using the generative function combinators described in the next section.

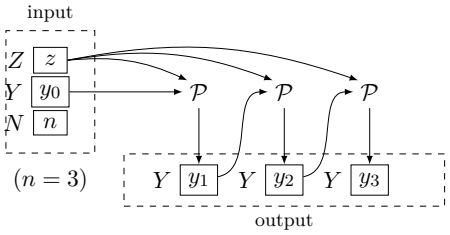
5.3 Generative function combinators for control flow

Probabilistic models with large numbers of random variables often arise in practice from patterns of repeating computation that possess conditional independencies that can be characterized statically and used to devise efficient inference algorithms. This fact is reflected in existing template-based representations like 'plates', which represent mutually conditionally independent groups of random variables and dynamic Bayesian networks, which represent repeated application of a fragment of a probabilistic model [66]. This section shows how generative functions for models with large numbers of random choices can be constructed from other generative functions via *generative function combinators*—functions that, given one or more input generative functions, return a generative function. The generative function combinators introduced in this section describe common patterns of structured control flow. Each combinator uses its own implementation of the generative function and trace data types that is specialized to the static pattern of conditional independence in the

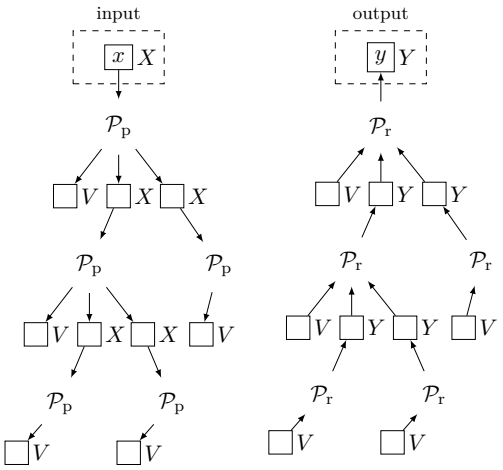
probability distributions that it represents. This section gives three examples, shown in Figure 5-3: (i) a generative function that maps another generative function independently across a vector of inputs (appropriate for modeling independently distributed data), (ii) a generative function that 'unfolds' another generative function (appropriate for modeling sequential processes like dynamic Bayesian networks), and (iii) a generative function that implements a basic pattern of recursion (appropriate for modeling processes like probabilistic context free grammars). Note that although these combinators share names with familiar higher-order functions from functional programming in some cases (e.g. 'map', 'unfold'), they are not generative functions themselves.



(a) Map generative function combinator



(b) Unfold generative function combinator



(c) Recurse generative function combinator

Figure 5-3: Generative function combinators for common control flow patterns

Map combinator The Map combinator (Figure 5-3a) constructs a generative function $\mathcal{P}' = (X', Y', p', f', q')$ that independently applies a generative function $\mathcal{P} = (X, Y, p, f, q)$ to each element of a vector of inputs $(x_1, \ldots, x_n) \in X' := \sqcup_{m=0}^{\infty} X^m$ and returns a vector of outputs $(y_1, \ldots, y_n) \in Y' := \sqcup_{m=0}^{\infty} Y^m$ where $\sqcup$ denotes disjoint union. Each application of $\mathcal{P}$ is assigned an address namespace $i \in \{1 \ldots n\}$. The trace data structure of the resulting generative function $\mathcal{P}$ stores a vector of subtraces, one for each application of $\mathcal{P}$. The implementation of UPDATE for $\mathcal{P}'$ only makes recursive calls to trace operations for the inner generative function $\mathcal{P}$ for those applications i for which the choice dictionary $\boldsymbol{\sigma}$ contains addresses under namespace i, or for which $x_i \neq x_i'$. Similarly, the implementation of REGENERATE recursively only invokes REGENERATE on subtraces for which there exists an address of the form (i, b) in the selected set B or for which $x_i \neq x_i'$. These operations use the change hint δ_X to determine which applications i have $x_i \neq x_i'$ without requiring a full scan over all i, and they can pass through change hints to individual applications if δ_X contains this information.

Unfold combinator The Unfold combinator (Figure 5-3b) constructs a generative function $\mathcal{P}' = (X', Y', p', f', q')$ that applies a generative function $\mathcal{P}$ repeatedly in sequence. Unfold generalizes dynamic Bayesian networks, a common building block of probabilistic models, by representing a Markov chain with state-space Y and transition $\mathcal{P}$ from $Y \times Z$ to Y. The generative function $\mathcal{P}$ maps $(y, z) \in Y \times Z$ to a new state $y' \in Y$. The generative function produced by this combinator takes as input the initial state $y_0 \in Y$, parameters $z \in Z$, and the number of Markov chain steps n to apply (that is $X' := Y \times Z \times \{1, 2, \ldots\}$). Each callee application is given an address namespace $i \in \{1 \ldots n\}$. Starting with initial state y_0, Unfold repeatedly applies $\mathcal{P}$ to each state and returns the vector of states $(y_1, \ldots, y_n) \in Y' := \sqcup_{m=0}^{\infty} Y^m$. The trace data structure of the resulting generative function stores a vector of subtraces, one for each application of $\mathcal{P}$. The combinator's implementation of the the abstract data operations exploits the conditional independence of applications i and j given the return value of an intermediate application l, where $i < l < j$. The change hint values δ_Y returned by the applications of $\mathcal{P}$ each i are used to determine which applications require a recursive call to the data type operations for the subtraces.

Recurse combinator The Recurse combinator (Figure 5-3c) takes two generative function, a *production function* $\mathcal{P}_{\text{p}}$ and a *reduction function* $\mathcal{P}_{\text{r}}$, and returns a generative function $\mathcal{P}' = (X, Y, p, f, q)$ that (i) recursively applies $\mathcal{P}_{\text{p}}$ to produce a tree of values, then (ii) recursively applies $\mathcal{P}_{\text{r}}$ to reduce this tree to a single return value. Recurse is used to implement recursive computation patterns including probabilistic context free grammars, which are a common building block of probabilistic models [38, 54, 65]. Letting b be the maximum branching factor of the tree, $\mathcal{P}_{\text{p}}$ maps X to $V \times (X \cup \{\perp\})^b$ and $\mathcal{P}_{\text{r}}$ maps $V \times (Y \cup \{\perp\})^b$ to Y, where $\perp$ indicates no child. The UPDATE implementation uses change hint values δ_X from the production and reduction functions to determine which subset of production and reduction applications require a recursive call to UPDATE.

A recursive DML function that is semantically equivalent to that produced by the Recurse combinator is shown on the left below. The function invokes generative functions `production` and `reduction` (definitions not shown). The right-hand side shows Julia code that generates a semantically equivalent generative function using the Recurse combinator.

```
@gen function f(x)
    (v, xs) = ({:prod} ~ production(x))
    ys = Dict()
    for i in 1:length(xs)
        ys[i] = ({i} ~ f(xs[i]))
    end
    y = ({:red} ~ reduction(v, ys))
    return y
end
```

```
f = Gen.Recurse(production, reduction)
```

A particle filtering benchmark Table 5.2 shows a performance comparison of several implementations of particle filtering (Section 3.5) for object tracking using a nonlinear state-space model. An object is assumed to move along a piecewise-linear path at con-

stant speed with Gaussian noise added to the distance traveled at each time step. The measurement model also assumes additive Gaussian noise. The task is to track the object over time along its assumed path. We evaluated two particle filtering inference algorithms implemented in Gen using the Unfold combinator. The first uses a generic proposal distribution based on forward sampling, which is the automatically generated internal proposal family (Section 4.1). The second uses custom proposal derived by manual analysis of the model and expressed in Gen's DML (Section 3.5). We compared these implementations to particle filtering implementations in Turing [40], Anglican [130], and Venture [79], none of which supported custom particle filtering proposals. The results show that the custom proposal gives accurate results in an order of magnitude less time than the generic proposal. Furthermore, the Gen implementation using the generic proposal significantly outperforms the Anglican, Turing, and Venture implementations of the same algorithm. Note that each of these implementations scales linearly in the number of time steps, whereas an algorithm implemented using only Gen DML and without Unfold would scale quadratically in the number of time steps because each time step would require a full end-to-end execution execution as part of the UPDATE operation. The running time is measured as the median runtime across multiple replicates required to achieve a mean log marginal likelihood estimate above a threshold within two nats of a gold-standard estimate.

Implementation	**Proposal distribution**	**Running time**
Gen (DML + Unfold)	Custom	4.9ms ($\pm$0.07)
Gen (DML + Unfold)	Generic	82ms ($\pm$3.6)
Anglican	Generic	275ms ($\pm$11)
Turing	Generic	1174ms ($\pm$25)
Venture	Generic	$> 10^6$ms

Table 5.2: Performance comparison of particle filtering implementations

A structure inference benchmark Table 5.3 shows a performance comparison for MCMC inference in a Gaussian process structure learning model based on a fully Bayesian adaptation [113, 109] of the model of [77]. The task is to infer the covariance function of a Gaussian process (GP) model of a time series data set, where the prior on covariance functions is defined using a probabilistic context free grammar (PCFG). The MCMC kernel is a trans-dimensional kernel that modifies the parse tree of the PCFG. We evaluated a Gen implementation of the kernel with and without the Recurse combinator, which automatically caches intermediate covariance matrix computations. The Gen DML implementation uses Julia recursion in DML to sample from the PCFG. The Gen 'SML + DML + Recurse' implementation uses SML for the top-level generative function, and a generative function constructed by Recurse for the PCFG-based prior distribution on covariance functions, and DML generative functions for the production and aggregation kernels that are passed into the Recurse combinator. The Gen DML implementation gives a 40x speedup over Venture, even without using Recurse or SML, and performs comparably to a hand-coded Julia implementation that does not cache intermediate covariance matrix computations. Using

SML and Recurse gives a 1.7x speedup over the handcoded implementation, due to the Recurse combinator's automatic reuse of cached intermediate covariance matrix computations, which is possible due to propagation of change hints within the tree of production and reduction applications. Interestingly, the DML implementation outperforms the Venture implementation, despite the fact that Venture's backend avoids unnecessary recomputation via dynamic dependency tracking and the DML implementation does not. This is explained by the high constant-factor overhead of Venture's backend relative to Gen's trace data type implementations. Because Venture uses a single *generic* backend implementation, in order to achieve asymptotically efficient updates, it implements fine-grained incremental computation for all code written in its modeling language. This leads to a complex data structure for storing random choices and a dynamic dependency graph that has high overhead. In contrast, because Gen uses *abstract* data types for traces, and allows for *interoperable* and *specialized* implementations of these data types, Gen avoids the need for fine-grained dynamic dependency tracking of arbitrary model code: Gen's combinators (in this case, Recurse) can achieve asymptotically efficient updates based on coarse-grained and static patterns of conditional independence and the DML implementation does not need to employ any dependency tracking, which vastly reduces constant-factor overhead.

Implementation	**Running time (per step)**
Gen (SML + DML + Recurse)	2.57ms (±0.09)
Julia (handcoded)	4.73ms (±0.45)
Gen (DML)	6.21ms (±0.94)
Venture	279ms (±31)

Table 5.3: Performance of different implementations of a trans-dimensional MCMC kernel

A Bayesian robust regression benchmark We also benchmarked several inference implementations applied to a Bayesian robust regression problem. The aim is to infer the slope and intercept of a linear model of data from observations that include outliers. Table 5.4 shows a comparison of Gen and Venture on an *uncollapsed* variant of the model that contains explicit discrete random choices which indicate whether each data point is an inlier or an outlier. A Gen implementation using SML and the Map combinator gives a >200x speedup over Venture for two inference algorithms (Metropolis-Hastings moves on both the parameters and the outlier indicator variables, and gradient-based optimization on the parameters combined with Metropolis-Hastings moves on the outlier indicator variables). The incremental computation optimizations enabled by the SML and the Map combinator give a roughly 100x speedup over the DML implementation, which does not exploit incremental computation and scales as $O(n^2)$ where n is the number of observed data points (n is ∼500 in the experiments). The ± values in the table indicate the inter-quartile range across many runs of each step. Table 5.5 shows a comparison of a Gen implementation of a Gaussian random walk Metropolis-Hastings algorithm with implementations in Anglican and Venture for a collapsed variant of the model, where the discrete random choices for the outlier indicators are manually eliminated from the model by summing them out. For each

implementation we measured the running time needed to exceed an accuracy threshold on held-out data. The Gen implementation is 10x faster than the Anglican implementation and many orders of magnitude faster than the Venture implementation.

Implementation	**Inference Algorithm**	**Running time (per step)**
Gen (SML + Map)	Custom Metropolis Hastings	64ms (±1)
Gen (DML)	Custom Metropolis Hastings	7,376ms (±87)
Venture	Custom Metropolis Hastings	15,910ms (±500)
Gen (SML + Map)	Gradient-Based Optimization	74ms (±2)
Gen (DML)	Gradient-Based Optimization	7,384ms (±85)
Venture	Gradient-Based Optimization	17,702ms (±234)

Table 5.4: Performance comparison of inference operations for Bayesian robust regression

Implementation	**Running time (to converge)**
Gen (SML + Map)	75.3ms
Anglican	783ms
Venture	1.3×10^6 ms

Table 5.5: Performance comparison of MCMC algorithms for Bayesian robust regression

5.4 Domain-specific generative functions

The DML and SML modeling languages and the generative function combinators discussed above are *general-purpose*—they are not specialized to any given application domain or class of models. Also, they are completely interoperable, in that generative functions compiled using any of these languages or combinators can internally invoke generative functions compiled from any of the other languages or constructs. And they can invoke essentially arbitrary pure-functional general-purpose code. While these dimensions of flexibility are useful, they are in tension with performance. High performance implementations of the generative function and trace ADT implementations require specialization to the particular structures and properties of the generative function. The DML does not exploit any such structure in its generative function and trace implementations, and the SML and combinators only exploit high-level conditional independence structure.

While DML, SML and combinators provide a reasonable baseline level of performance, it is possible improve performance further by implementing more specialized generative functions. There is a wide gamut of different potential specializations, ranging from hand-crafting a trace implementation that optimize memory layout of the trace data structure for a user's model to leveraging a specialized runtime like TensorFlow, to using dynamic programming in a restricted modeling language to construct a more efficient internal proposal distribution. We now discuss a few types of more specialized implementations of the generative function and trace ADTs that are possible.

Wrapping TensorFlow and PyTorch computations in generative functions It is possible to compile generative functions from code written in domain-specific modeling languages like TensorFlow and PyTorch that use runtime systems specialized for certain computational workloads. The resulting ADT implementations invoke the these alternate runtime systems as part of every generative function and trace operation. Constructors for generative functions based on pure-functional deterministic TensorFlow and PyTorch code have been implemented. In particular the gradient trace operation is implemented using the automatic differentiation capabilities of TensorFlow and PyTorch respectively. For DML and SML generative functions that invoke a generative function compiled from one of these other languages, the gradient operation combines automatic differentiation of Julia code with automatic differentiation of the specialized runtime system. The code below shows the construction of a TensorFlow-based generative function (`neural_net`) that is invoked by a DML generative function (`inference_model`) on the right. In particular, the DML function implements a discriminative inference model that takes as input an image, and passes the image to `neural_net`, which returns parameters of beta distributions from which the DML function samples at addresses `rot`, `left_x`, etc.

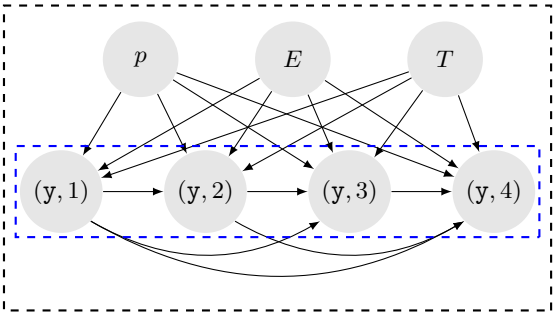
```
tf = pyimport("tensorflow")
image_flat = tf.placeholder(..)
W_conv1 = tf.Variable(..)
b_conv1 = tf.Variable(..)
...
output = tf.add(..)
neural_net = GenTF.TFFunction(
    [W_conv1, b_conv1, ..],
    [image_flat], output)
```

```
@gen function inference_model(image)
  nn_output ~ neural_net(image)
  rot ~ beta(nn_output[1], nn_output[2])
  left_x ~ beta(nn_output[3], nn_output[4])
  right_y ~ beta(nn_output[5], nn_output[6])
  right_z ~ beta(nn_output[7], nn_output[8])
  ...
end
```

A specialized generative function for discrete hidden Markov models Discrete-valued hidden Markov models [99] are a popular class of models because they admit efficient algorithms for inference over latent states and marginal likelihood evaluation via the forward algorithm and forward-filtering backward sampling. These algorithms are based on dynamic programming, and are specialized to both the graph structure of hidden Markov models (a chain) and the domain of the individual random choices (discrete). Importantly, these algorithms are *exact* and not approximate and have predictable running time (for a chain with T time steps and K possible hidden states per time step, the forward-filtering backward-sampling algorithm can be used to sample from the conditional distribution of hidden states given observations in time $O(K^2T)$). It is possible to leverage these specialized algorithms within a Gen model by implementing a generative function that uses them as the basis of *optimal* internal proposal family (Section 4.1) that samples from the conditional distributions $p(\cdot|\boldsymbol{\sigma})$ on random choices given choices $\boldsymbol{\sigma}$. It is also possible to implement a generative function that implements a *collapsed* HMM. For example, the generative function `collapsed_hmm` below takes as input the parameters of a HMM, and encodes a probability distribution on the observed variables only (with addresses $1, \ldots, n$). Each evaluation of $p(\boldsymbol{\tau}; x)$ within `collapsed_hmm` triggers a run of the forward-backward algorithm to compute the marginal likelihood. In the context of the DML generative function `hmm_based_model`

that calls `collapsed_hmm`, the observed states have addresses $(\texttt{y}, 1), \ldots, (\texttt{y}, n)$. Note that the author of `hmm_based_model` does not need to know that `collapsed_hmm` is a specialized generative function implementation that uses dynamic programming internally, because it behaves externally the same way as all other generative functions.

```
@gen function hmm_based_model(n)
    p ~ dirichlet(10)
    E ~ emission_matrix_prior()
    T ~ transition_matrix_prior()
    y ~ collapsed_hmm(p, E, T, n)
end
```



5.5 Related work

Conditional independence and probabilistic inference Conditional independence of random variables is the basis of the 'graphical models' approach to probabilistic modeling and inference [66], in which the presence of conditional independencies admits the construction of efficient inference algorithms. The graphical models paradigm includes modeling constructs for expressing conditional independence, including Bayesian networks (directed acyclic graphs where nodes are random variables), and 'plate notation' and dynamic Bayesian networks. The graphical models paradigm is most adept at representing *static* conditional independencies that always hold, but has also formalized notions of *context-specific* conditional independencies that only hold when certain random variables take specific values. Although conditional independence of random variables is usually studied in the setting of a fixed-length vector of random variables, related types of structure have been studied in more general classes of probabilistic models. Examples include open-universe probabilistic models [84, 83] and infinite contingent Bayesian networks [85], probabilistic models and inference algorithm from Bayesian nonparametric [64], and dynamic programming algorithms for stochastic context free grammars [68].

Incremental computation in probabilistic programming languages A number of sampling-based probabilistic programming systems use the universal Metropolis-Hastings (MH) MCMC inference algorithm that was introduced in Church [51] and later generalized for use with lightweight embeddings of probabilistic modeling languages in host languages [127]. The algorithm, sometimes called 'lightweight MH', benefits from incrementalization because it involves updating one random choice at a time in an execution of the program that represents the probabilistic model. A naive implementation of the algorithm involves interpreting a variant of the program end-to-end, and computing the MH acceptance probability by separately computing the numerator and denominator, even when most factors in the numerator and denominator cancel due to conditional independencies in the model. A number of approaches have been taken to incrementalize this and

related algorithms by leveraging conditional independencies. The Shred [132] tracing interpreter for Church recognizes the opportunity for specialization of the inference algorithm to traces of a certain structure (i.e. control flow path), and compiles and caches MH kernel implementations that are specialized to each structure that is encountered throughout the MH algorithm. Shred JIT compiles straight-line traces of the probabilistic model and applies a reaching analysis ('slicing') to these traces to reduce redundant computation in the MH acceptance probability calculation. The C3 implementation [103] of lightweight MH wraps call sites in the model with a cache lookup, transforms the model to continuation-passing-style (CPS), and maintains a call stack data structure. One essential difference of these two approaches with Gen's is that they are specialized to one particular inference algorithm—lightweight MH, whereas Gen's trace data type supports an open-ended set of inference algorithms via the much more flexible UPDATE operation. An crucial feature of UPDATE is that it can modify the values of multiple random choices at once—this permits use of proposal distributions that are specialized to the model, as opposed to the generic forward sampling proposal distribution that is employed within lightweight MH. Furthermore, UPDATE can be used to implement algorithms other than MCMC, including various sequential Monte Carlo [33] algorithms, annealed importance sampling [91], and numerical and stochastic local optimization. However, it should be possible to use Shred's JIT specialization approach as the basis of a more optimized trace data type for a variant of Gen's dynamic modeling language. Gen's SML trace implementation already does exploit JIT specialization of trace operations to the set of random choices being manipulated (e.g. the set of choices being changed in a call to UPDATE) but SML does not itself support control flow (control flow is instead handled by other modeling constructs including generative function combinators, or DML functions).

Although the Venture [79] probabilistic programming system also builds on ideas from Church [51] and employs incremental computation, it is distinct from Shred and C3 because its data structures are designed to simultaneously support multiple inference algorithms, including algorithms that involve updating multiple random choices at once. Venture uses a custom interpreter for a pure variant of Scheme [100] that constructs a directed acyclic graph that contains one or more nodes for every application of `eval`. Edges in this graph reflect dependencies between evaluated expressions. To compute the set of re-evaluations that are necessary for a given set of proposed-to choices, a walk of this graph is performed starting at the nodes corresponding to proposed-to-choices. This data structure is used as the basis for several inference algorithms, including a restricted form of Metropolis-Hastings that proposes to multiple random variables at once using forward sampling, and a restricted form of sequential Monte Carlo that extends a model with additional statements using forward sampling as the proposal for any new latent variables. While Venture's approach to incremental computation is more flexible than that of Shred and C3, it has high constant-factor overhead. Gen's use of incremental computation differs from that of Venture in several respects: First, Gen's trace data types are significantly more flexible than that of Venture because they allow arbitrary modifications to the values of random choices using unique user-defined addresses. Second, whereas Venture uses a single generic implementation for the trace data structure for all models, Gen specializes the trace data structure to the model. This allows Gen to perform much of the computation for identifying dependen-

cies statically during generation of its specialized trace data type implementation, instead of dynamically as in Venture, leading to very large performance improvements on benchmarks that compare implementations of the same algorithm in the two systems. The subset of the Gen current modeling constructs that do exploit incremental computation (SML and certain generative function combinators) is less expressive than Venture's modeling language. However, Gen's DML is equally expressive to Venture's modeling language and incurs much less constant-factor overhead, making it substantially more performant even for traces with reasonably large numbers of random choices, for which we would expect incremental computation to provide significant gains. Furthermore, in Gen it is possible to construct models from a combination of DML, SML, and combinators, which allows for selective use of compile-time trace specialization for certain parts of the model. It should be possible to implement a highly expressive Gen modeling language that uses a trace data type implementation based on Venture's dynamic dependency graph and is interoperable with Gen's existing modeling constructs, although it remains to be seen whether this implementation strategy can be made performant enough to compete with Gen's current modeling constructs. Finally, Venture automatically performs incremental computation in certain cases of exchangeable coupling between random choices. Gen's current modeling constructs do not automate this, but devising modeling constructs (e.g. generative function combinators) that do is an interesting area for future work. One potential challenge is that random choices that are distant in the call tree of the model can be exchangeably coupled, and updates to these choices therefore require non-local communication between parts of the trace, which may complicate compile-time specialization of traces.

Effect handlers in probabilistic programming system implementations Effect handlers are also used in the implementations of several other probabilistic programming systems, although in a different manner to how they are used in Gen. There are two other ways that effect handlers have been used in other probabilistic programming systems. First, systems including WebPPL [52], Anglican [130], and Turing [40] use a separate effect handler for each inference algorithm that is supported by their respective inference engines. Random choice statements (e.g. `sample`) and in some cases `observe` or `factor` statements in the modeling languages of these systems are associated with a special effect handler for each algorithm. Therefore, extending these systems with new inference algorithms requires understanding and extending the implementation of the probabilistic modeling language with a new effect handler.

Other systems including Pyro [13] and Edward2 [86] also use effect handlers to implement non-standard interpretations of probabilistic modeling code as part of inference, but each handler implements a lower-level behavior than those of WebPPL, Anglican, and Turing. In Pyro and Edward2, the effect handlers are intended to be composed to generate more complex behaviors [86]. This allows for new inference algorithms to be implemented more concisely in some cases than in WebPPL, Anglican, and Turing. However, implementers of new inference algorithms still need to understand these effect handlers and how they are implemented by the runtime system, and may need to implement new ones.

In contrast to the two approaches discussed above, in Gen effect handlers do not appear in user inference code. Instead, effect handlers are one particular implementation strategy

that can be used to implement the trace abstract data type. Some of the effect handlers used in the implementation of of Gen's Dynamic Modeling Language are similar to those used in Pyro. However, other Gen modeling languages do not use effect handlers at all, but instead are based on other implementation strategies including source-to-source transformation. Indeed, the purpose of Gen's abstract trace data type is to decouple language implementation details (like effect handlers) from user inference code, making inference code simpler and easier to understand and extend. The trace abstract data type also serves as a type of foreign function interface, which allows generative functions in general-purpose modeling languages to invoke generative functions compiled from much more specialized languages with higher-performance and more specialized implementations. These callee generative functions may also use their own specialized inference algorithms internally to implement the trace abstract data type more efficiently (e.g. exploiting dynamic programming in the case of factor graph and hidden Markov model generative functions). The composable effect handler architecture of Pyro and Edward2 is not designed to support this type of extensibility, and is rooted in the assumption that a single runtime system is used for the entire model. However, by committing to a small interface to the modeling language runtime (the trace abstract data type), Gen does sacrifice some potential for specialization of low-level inference code to the inference algorithm context. For example, the NumPyro [96] implementation of Pyro is able to perform optimizations like automatic vectorization on the implementations of inference algorithms, whereas Gen encapsulates the low-level operations that comprise inference algorithms within the trace abstract data type, making dynamic transformations and other optimizations of high-level inference algorithm logic less natural.

Static compilers for probabilistic programming languages Gen statically compiles model code into artifacts representing generative functions that have associated specialized trace data type implementations. While static compilers for probabilistic programming systems have been developed [131, 60], these compilers are responsible for generating inference implementations for more general-purpose inference algorithms. In contrast, Gen does not currently static compile the inference algorithm, which is composed by user and library Julia code on top of the trace data type. Instead, Gen uses static compilation primarily as a means to statically specialize trace implementations to the structure in models, agnostic to the inference algorithm. JIT compilation of some trace operations for certain modeling languages does specialize to limited inference algorithm context, but this context is limited to the set of addresses involved in the trace operation, and not the whole inference algorithm. Development of inference compilation techniques that retain the flexibility of Gen's inference programming capabilities is an interesting area of future work.

Chapter 6

Applications

The previous chapters described Gen's design and implementation. This chapter describes several applications of Gen. The applications utilize Gen's modeling flexibility—the models make use of black-box simulators, stochastic discrete structures, and both discrete and continuous random variables. The applications also utilize Gen's inference algorithm flexibility, in the form of Monte Carlo inference, hybrid Monte Carlo and neural inference, surrogate models, and custom trans-dimensional MCMC kernels.

6.1 Inference in generative models of intelligent behavior

This section describes an application of Gen to the task of interpreting the behavior of an intelligent agent. The application adopts the *inverse planning* paradigm [8], in which we make inferences about the intentions or beliefs of an intelligent agent from their behavior using probabilistic inference in a generative model that explains behavior as arising from rational pursuit of objectives given beliefs. The section presents several variants of the model and the inference algorithm, implemented using Gen. The models exercise the expressiveness of Gen's modeling languages via stochastic control flow and the use of black-box code simulators. The inference constructs used include importance sampling and MCMC with data-driven proposals (Section 3.2 and Section 3.4.2), the composite kernel DSL (Section 3.4.4), resample-move particle filtering (Section 3.5), involutive MCMC (Section 3.7), and the ability to construct specialized domain-specific ADT implementations (Chapter 5). An earlier version of the modeling approach used for this application was described by Cusumano-Towner et al. [31].

6.1.1 An algorithmic generative model of goal-directed movement

The code below defines a generative probabilistic model of a person's goal-directed movement throughout an indoor environment using Gen's DML:

```
@gen function agent_model(floorplan, start_location, num_time_steps)
    destination ~ random_location(floorplan)
    path = plan_path(floorplan, start_location, destination)
    observations ~ noise_model(path, num_time_steps)
end
```

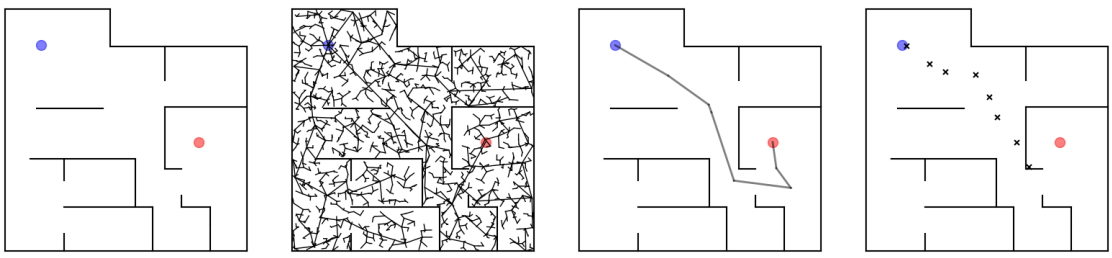
The model takes three arguments, which represent information that is assumed to be known: the floorplan of the indoor environment (including walls and other obstacles), the starting location of the person, and the number of time steps at which the person's location will be observed. The first statement samples the destination of the agent from a uniform prior distribution on the free space in the floorplan.

```
destination ~ random_location(floorplan)
```

Here, `random_location` is a custom family of probability distributions on values in $\mathbb{R}^2$. A possible value for the destination is shown as a red dot in Figure 6-1a, and the starting location is shown in blue. Next, we invoke the Julia function `plan_path`, which takes the floorplan, start location, and destination locations, and uses a path planning algorithm to generate a path from the start location to the destination location that does not intersect with any walls or obstacles in the floorplan:

```
path = plan_path(floorplan, start_location, destination)
```

Note that the ability to include essentially arbitrary executable code in generative models allows use of *algorithmic models* of complex phenomena. Here we are using a path planning algorithm to model the complex goal-directed behavior of an intelligent agent. This model assumes the agent is approximately rational and takes relatively efficient routes from its current location to its destination. It would be more difficult to express such a model mathematically, but here we can repurpose off-the-shelf code for use in our model. The algorithm internally constructs a rapidly exploring random tree [69] (Figure 6-1b) and uses this tree together with trajectory optimization to generate the path (Figure 6-1c). The internal randomness used in the path planning algorithm is *encapsulated randomness* (Section 4.5). The next line in the DML function body samples hypothetical observations



(a) (b) (c) (d)

Figure 6-1: A prior sample from a generative model of goal-directed intelligent behavior

using a generative function (`noise_model`) that simulates the person walking along the path with random perturbations as well as measurement noise (Figure 6-1d).

```
observations ~ noise_model(path, num_time_steps)
```

Unlike the sampling statement for `destination`, `noise_model` is a generative function not a distribution, and it generates *many* random choices recorded in the trace at addresses that are nested under the address namespace `:observations`. In this case, the sub-addresses are simply the integers `1` through `num_time_steps`, and each sub-address contains the hypothetical observed location at one time step. The generative function `noise_model` uses a specialized ADT implementation (Section 5.4) that implements a probabilistic variant of dynamic time warping [111] to account for variability in the person's progress along their path, combined with normally distributed sensor noise. Specifically, we assume a nominal speed for the agent along their path, and then allow the agent some probability of either taking no steps or taking an extra step at each time step, and we marginalize out these discrete random variables using the forward-backward dynamic programming algorithm.

6.1.2 A simple and generic inference implementation

The following Julia code takes as input the floorplan (`floorplan`, definition not shown), the starting location (`start_location`, definition not shown) and an input an array (`data`, definition not shown) of `num_time_steps` observed locations on the floorplan. It generates approximate conditional samples of the destination location using self-normalized importance sampling with the default internal proposal family (Algorithm 13) applied to the generative model above.

```
observations = Gen.choicemap()
for i in 1:20
    observations[:observations => i] = data[i]
end
args = (floorplan, start_location, 20)
traces, weights = Gen.importance_sampling(agent_model, args, observations, 100)
traces = [traces[categorical(weights)] for _ in 1:length(traces)]
destination_samples = [trace[:destination] for trace in traces]
```

The first lines populate a choice dictionary with the observed data at the hierarchical address `:observations => i`:

```
observations = Gen.choicemap()
for i in 1:20
    observations[:observations => i] = data[i]
end
```

Next, we invoke the `importance_sampling` function from Gen's inference library, passing the arguments to the model generative function that contain values known a-priori, to produce a weighted collection of traces that are approximately distributed according to the conditional distribution:

```
args = (floorplan, start_location, 20)
traces, weights = Gen.importance_sampling(agent_model, args, observations, 100)
```

We then extract the destination from each trace and resample in proportion to the weights:

```
traces = [traces[categorical(weights)] for _ in 1:length(traces)]
destination_samples = [trace[:destination] for trace in traces]
```

The results for an observed trajectory are shown in Figure 6-2. Each red dot shows a destination sampled approximately from the conditional distribution given the observed locations, which are shown in black. Note that there is substantial uncertainty about the destination—the conditional distribution is approximately uniform within the set of three rooms in the lower-left corner.

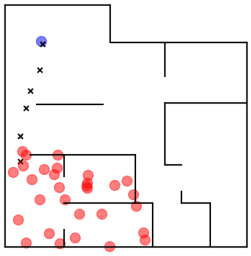


Figure 6-2: Inferences about a person's destination from their observed movement

The inference code above is one of the simplest types of inference programs that users can write using Gen. Generic inference programs like this can give reasonable results on easy inference problems, but are not efficient, and do not scale to harder inference problems. Next, we elaborate on the generative model and the inference algorithm.

6.1.3 Adding uncertainty about structure and stochastic control flow

In addition to algorithmic models, another modeling capability that is useful for cognitively inspired generative models is *uncertainty about model structure*. Gen users express uncertainty about model structure by writing models where the set of addresses of random choices that are sampled is itself random. Like `agent_model` above, the DML function `leave_model` below defines a generative model of the motion of a person moving around an indoor space, but is more complex:

```
@gen function leave_model(floorplan, start_location, num_time_steps)
      num_attempts ~ geometric(0.5)
      path = [start_location]
      previous_location = start_location
      for i in 1:num_attempts
            current_location = ({(:search_loc, i)} ~ likely_key_location())
            append!(path, plan_path(floorplan, previous_location, current_location))
            previous_location = current_location
      end
      destination ~ location_near_exit()
      append!(path, plan_path(floorplan, previous_location, destination))
      observations ~ noise_model(path, num_time_steps)
end
```

Specifically, `leave_model` models the behavior of an person who seeks to leave their apartment, but searches for their keys first. The number of times they search for their keys is random (sampled from a `geometric` distribution). The person knows where they typically store their keys, and they tend to search for their keys in these locations (`likely_key_location` is a probability distribution on locations within the floorplan that is biased towards these locations; see the left of Figure 6-3, where green and yellow indicate high probability density). According to `leave_model`, after the person finds their keys, they head for a location near the exit of the apartment (in the lower right corner, near the green dots in Figure 6-3). Whereas the previous model assumed the person had a single destination and walked an efficient route from their starting location to the destination, this model posits the existence of an unknown number of events (searching for keys) and assumes that the person walks efficient routes between these search locations and from the final search location to the exit. Therefore, this model generates more complex trajectories. The right of Figure 6-3 shows six prior samples from `leave_model`, where the start location is shown in blue, the exit location is shown in green, and purple dots are locations where the person searched for their keys. The person searches for their keys a random (geometrically-distributed) number of times before they find them, and then leaves the apartment.

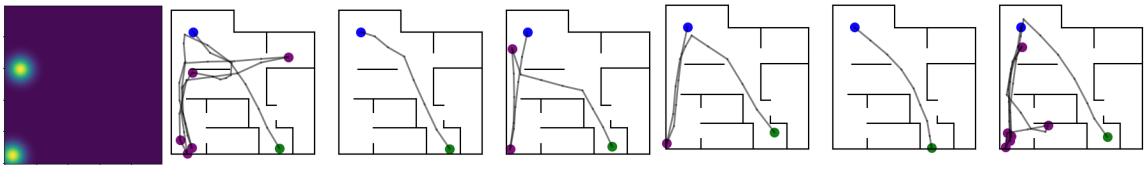


Figure 6-3: Samples from a model of intelligent behavior with stochastic structure

Combining two models using stochastic branching If we knew the person was leaving the apartment, then we could use the model `leave_model` to infer whether the person has their keys, and where and when they searched for their keys as they moved around the apartment. These are more complex explanations than were possible with `agent_model`. However, suppose we did not know whether they are leaving the apartment or not. To infer whether they are leaving, we need another generative model for their movement patterns in the condition where they are not leaving. Consider `stay_model` defined below:

```
@gen function stay_model(floorplan, start_location, num_time_steps)
    num_waypoints ~ poisson(2)
    path = [start_location]
    previous_location = start_location
    for i in 1:num_waypoints
        current_location = ({(:waypoint, i)} ~ random_location(floorplan))
        append!(path, plan_path(floorplan, previous_location, current_location))
        previous_location = current_location
    end
    observations ~ noise_model(path, num_time_steps)
end
```

This model is similar to `leave_model`, but instead of searching for their keys and then heading for the exit, this model simply posits an unknown number of 'waypoints' of unknown purpose. The number of waypoints is random and Poisson-distributed with mean 2, and the waypoints are sampled at random locations in the floorplan. Samples from the prior distribution of `stay_model` are shown in Figure 6-4a. To infer whether the person is leaving or not, we could explicitly perform Bayesian model comparison, by separately computing estimates of how well each of the two models explain a given set of observed data, and comparing these values.[1] Instead, we will exercise the flexibility of Gen's modeling languages again, and take a more efficient approach—we write a single model that *combines* the two models into one using a stochastic if-else branch. Note that both models end in the same way by sampling the observations from the noise model. Therefore, we refactor `leave_model` and `stay_model` by removing this common line of code and instead return the path from both functions. The new model is:

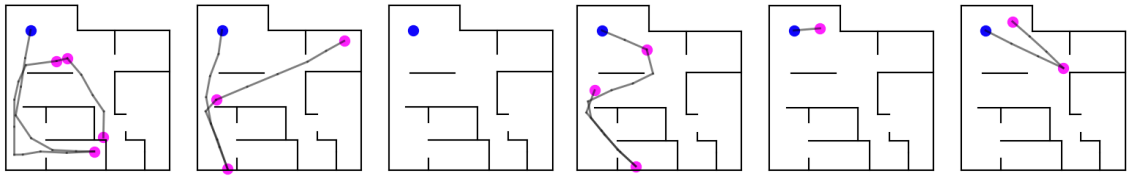
```
@gen function combined_model(floorplan, start_location, num_time_steps)
    if ({:is_leaving} ∼ bernoulli(0.1))
        path = ({:leave} ∼ leave_model(floorplan, start_location))
    else
        path = ({:stay} ∼ stay_model(floorplan, start_location))
    end
    observations ∼ noise_model(path, num_time_steps)
end
```

Prior samples from `combined_model` are shown in Figure 6-4b. Note how generative models in Gen are composable and easy to modify, combine, and refactor because they are based on functions. By doing inference in the combined model, we will infer whether the person is leaving or staying (by reading off the Boolean value at address `is_leaving` in each inferred trace), but also the specific sequence of waypoints or search locations.

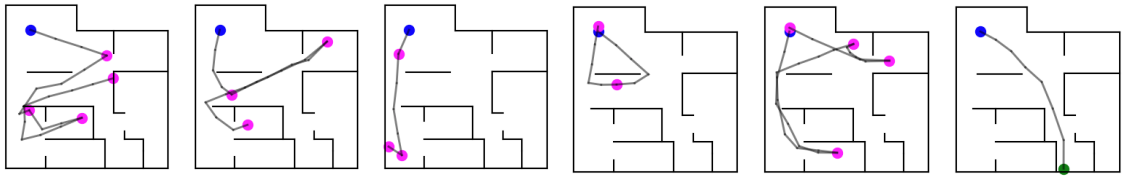
6.1.4 A sequential Monte Carlo inference algorithm

The Julia code in Figure 6-5 below is application inference code that implements an efficient sequential Monte Carlo [35] algorithm for inference in `combined_model`: We now highlight key elements in this code. First, the program creates and manipulates traces of `combined_model` using `Gen.generate` (Line 11) and `Gen.update` (Line 32), which are two of the operations supported by the trace data type, as well as a custom MCMC kernel (`custom_kernel`, on Line 25), which is a Julia function that takes a trace as input and returns a new trace as output. The program is written in a straightforward familiar Julia style, and makes use of Julia's control flow (loops), including its multi-threading features (`Threads.@threads`). The program begins by obtaining initial traces of the model where `num_time_steps` is `1`; the initial values for each random choice besides the observation are

[1] For example, averaging the importance weights returned for a model gives an estimate of the marginal likelihood of that model, and we could multiply the marginal likelihood ratio by the prior ratio (which expresses the prior probability of one model versus the other) to obtains the Bayes factor and then the posterior probabilities on the two models.



(a) Prior samples from `stay_model`. The start location is shown in blue, and waypoints (of unknown purpose) are shown in pink. There is a Poisson-distributed number of waypoints, and waypoints are sampled uniformly at random from locations in the floorplan.



(b) Prior samples from `combined_model`. With probability 0.1, the agent is leaving (and otherwise staying). In one of the samples shown here (rightmost), the agent is leaving.

Figure 6-4: Prior samples from alternate models of a person's activity and motion

produced by `Gen.generate` using the model's internal proposal distribution, which is encapsulated by the trace data type. The outer loop (Line 14) iterates over time steps, and alternates between culling low-weight traces (Lines 16-18), doing MCMC inference on each trace given the observed data so far (Lines 20-27), and extending the traces to the next time step and incorporating the new observed data point (Lines 29-32). The application code closely mirrors the pseudocode for the resample-move particle filter algorithm (Algorithm 7) which has also been added to Gen's inference library (although users may want to re-implement it for more control over the flow and logging).

Composing a custom rejuvenation MCMC kernel The part of the inference algorithm that is most heavily specialized to the model at hand is the rejuvenation MCMC kernel (`custom_kernel`). While users are free to define this kernel using plain Julia code, we define it using the composite kernel DSL (Section 3.4.4) in Figure 6-6. This composite MCMC kernel applies a sequence of primitive MCMC kernels. First it applies a selection MH (Section 4.3) kernel (using `Gen.mh`, Line 2) that proposes new values of all random choices using the model's encapsulated internal proposal. Next, it applies an involutive MCMC (Section 3.7) kernel (`switch_kernel`, Line 3) that proposes to switch branches (inverting the Boolean value `is_leaving`). This MCMC kernel will be discussed in more detail shortly. Next, the composite kernel applies a difference sequence of kernels, depending on which branch the trace has taken (Line 4).

A specialized reversible jump move for efficiently switching branches Consider the kernel `switch_kernel`, which is responsible for proposing a switch of the branch. This kernel is constructed using the involutive MCMC construction of Section 3.7. Specifically, it is composed from an auxiliary generative function (code not shown) and a trace transform

```
# Given: data (an array of noisy measurements of the 2D location)
num_particles = 128
traces = Vector{Trace}(undef, num_particles)
weights = Vector{Float64}(undef, num_particles)

# obtain initial traces, which include the observation for the first time step
observations = Gen.choicemap(); observations[:observations => 1] = data[1]
arguments = (floorplan, start, 1)
for i in 1:num_particles
    traces[i], weights[i] = Gen.generate(combined_model, arguments, observations)
end

for t in 2:length(data)

    # duplicate traces with high weights, delete traces with low weights
    normalized_weights = normalize(weights)
    traces = [traces[categorical(normalized_weights)] for i in 1:num_particles]

    # apply rejuvenation MCMC kernels to each trace in parallel
    Threads.@threads for i in 1:num_particles
        for iteration in 1:20

            # the custom MCMC kernel is defined with Gen composite kernel DSL
            traces[i] = custom_kernel(traces[i])
        end
    end

    # extend traces to the next time step, incorporating new data
    observations = Gen.choicemap(); observations[:observations => t] = data[t]
    new_arguments = (floorplan, start, 1)
    traces[i], weights[i] = Gen.update(traces[i], new_arguments, observations)
end
```

Figure 6-5: Gen implementation of an SMC algorithm for inferring a person's destination

(`switch_bijection`, defined in Figure 6-8).

```
switch_kernel(trace) = Gen.mh(trace, q_switch, (), switch_bijection)
```

The bijective trace transform `switch_bijection` in Figure 6-8 defines a transformation on traces of the model that switches the branch (Lines 3-5) and then populates the values of random choices in one branch using the values of random choices in the other branch. Note that most other universal probabilistic programming systems lack the ability to reuse values in one branch when switching branches. Instead, they rely on a *generic* mechanism for filling in values in the new branch: The new branch is executed from scratch, ignoring the values in the old branch. Although this generic approach to MCMC inference over structure converges to the correct distribution *if repeated infinitely many times*, generic proposals of this form amount to random guessing and are unlikely to be accepted, making

```
1  @kern function custom_kernel(trace)
2      trace ~ Gen.mh(trace, Gen.complement(Gen.select(:observations)))
3      trace ~ switch_kernel(trace)
4      if trace[:is_leaving]
5          trace ~ add_remove_search_attempt_kernel(trace)
6          trace ~ leave_random_walks_kernel(trace)
7      else
8          trace ~ add_remove_waypoint_kernel(trace, points)
9          trace ~ stay_random_walks_kernel(trace)
10     end
11     return trace
12 end
```

Figure 6-6: A composite MCMC kernel that combines several types of primitive kernels

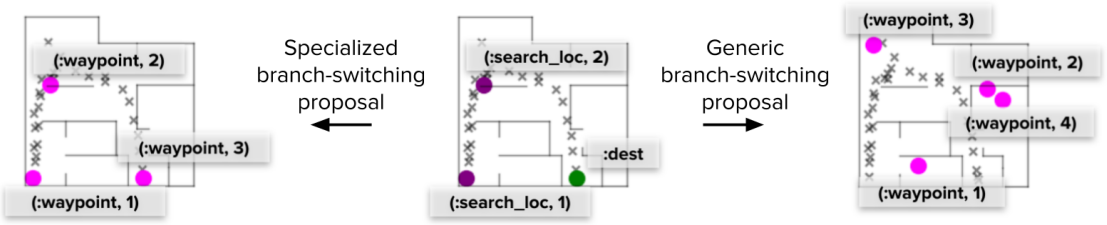


Figure 6-7: Contrasting generic and specialized MCMC moves for changing control flow

inference over structure inefficient. Recognizing that 'waypoints' in the 'stay' model have a related semantics to 'search-locations' in the 'leave' model, our `switch_bijection` reinterprets waypoints as search locations and vice-versa, making proposals much more likely to be accepted. Effectively, we are *reusing* the inference already done for one control flow path to jumpstart inference for the new control flow path. Figure 6-7 shows a state in which `is_leaving` is true (middle) and new states with `is_leaving` set to false that are proposed by our specialized kernel `switch_kernel` (left) and a generic kernel (right), respectively. The specialized kernel proposes a hypothesis that explains the data well and is likely to be accepted, whereas the generic kernel proposes hypotheses for the 'stay' model from the prior that are likely to be rejected.

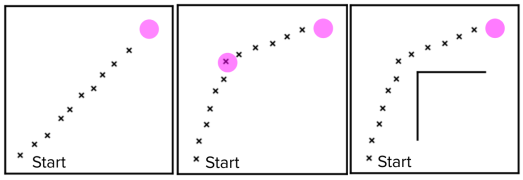
Proposing new waypoints based on a heuristic detector Consider the involutive MCMC kernels `add_remove_waypoint_kernel` and `add_remove_search_attempt` that are invoked on Line 5 and Line 8 of `custom_kernel` respectively. These kernels add or delete waypoints from the trace. Different techniques can be used to propose new waypoint locations in these kernels. The generic proposal mechanism that forms the basis for most existing universal probabilistic programming systems (forward sampling) would propose values for waypoints uniformly at random, just as when switching branches. However, as shown in Figure 6-9, we can instead use a simple heuristic to find points along the observed trajectory where the trajectory appears to change direction. These points will often be

```
@transform switch_bijection (model_in, aux_in) to (model_out, aux_out) begin

    # switch branches
    is_leaving = @read(model_in[:is_leaving], :disc)
    @write(model_out[:is_leaving], !is_leaving, :disc)

    if is_leaving

        # populate choices in leave branch, using old choices in stay branch
        @copy(aux_in[:reuse_dest], aux_out[:set_dest_from_waypoint])
        num_attempts = @read(model_in[:num_search_attempts], :disc)
        if @read(aux_in[:reuse_dest], :disc)
            num_waypoints = num_attempts + 1
            @copy(model_in[:destination], model_out[(:waypoint, num_waypoints)])
        else
            num_waypoints = num_attempts
            @copy(model_in[:destination], aux_out[:destination])
        end
        @write(model_out[:num_waypoints], num_waypoints, :disc)
        for i in 1:num_attempts
            @copy(model_in[(:search_loc, i)], model_out[(:waypoint, i)])
        end
    else

        # populate choices in stay branch, using old choices in leave branch
        ..
    end
end
```

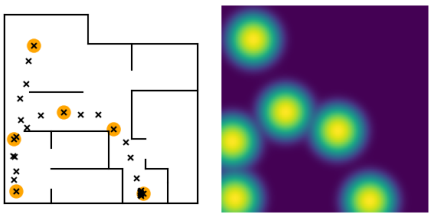
Figure 6-8: A Gen trace transform that switches control flow branches

waypoints, because the person is unlikely to change direction unless there is a waypoint. Note that this heuristic is not perfect—the heuristic does not know about the floorplan, so it cannot distinguish between turns that are explained by the floorplan and those that are not. But Monte Carlo inference algorithms including importance sampling (Section 3.2), MCMC (Section 3.4.2, Section 3.7) and particle filtering (Section 3.5) that use data-driven proposals can filter out the incorrect proposals using by assigning a low importance weight or by rejecting a proposal outright. Furthermore data-driven proposals can be learned or tuned automatically via supervised learning on simulated or real training data (Section 3.3).

Using a coarse-grained surrogate model as the basis of a proposal The data-driven proposal for waypoints described above is based on a non-probabilistic heuristic. We can also use a probabilistic heuristic to propose new waypoints. In particular, we use probabilistic inference in a *simplified* version of the generative model as a proposal distribution for inference in the original generative model. Instead of modeling the precise location of an agent using a path planning algorithm, we can simply model what room



(a) Changes in apparent direction are often indicative of a waypoint (pink) because we assume that the agent tends to take direct paths to their next waypoint. But this heuristic can produce false positives because it does not take into account prior knowledge about the obstacles.

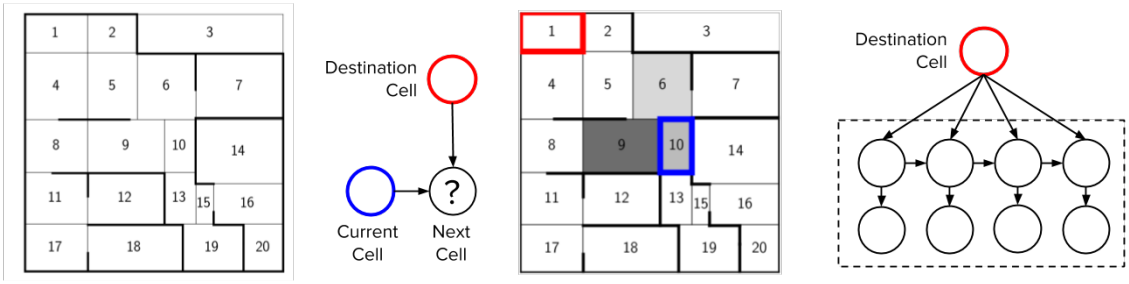


(b) Left: Heuristic waypoint detections from from the Ramer-Douglas-Peucker [36] line simplification algorithm. Right: The density of a data-driven proposal distribution that proposes a waypoint by randomly picking one of the heuristic detection and adding Gaussian noise to it.

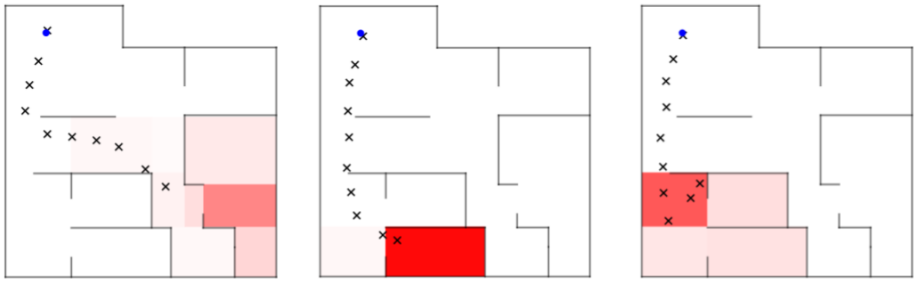
Figure 6-9: A data-driven proposal based on a heuristic

they are in (Figure 6-10a) using family of hidden Markov models (in this case with 20 discrete states). Because exact inference in hidden Markov models is efficient, we can perform exact inference easily in this coarse and discrete model of the application (Figure 6-10b). The simplified generative model can be trained offline on data simulated from the original generative model. Then, inferences in the simplified model can be used to aid inference in the original model in various ways, including (i) within proposal distributions in importance sampling, MCMC, and sequential Monte Carlo, and (ii) as part of a coarse-to-fine sequential Monte Carlo algorithm (Section 3.6). Here we use inference in the discrete model to propose new waypoint locations within `add_remove_waypoint_kernel` and `add_remove_search_attempt_kernel`.

Using custom implementations of Gen's abstract data types The surrogate model based on hidden Markov models can be implemented itself using Gen's built-in modeling languages like DML. However, since a hidden Markov model is a reusable modeling motif, it makes sense to implement specialized, performant, and reusable generative function and trace data types for it. Here we used a specialized implementation for collapsed discrete HMMs that internally uses an efficient implementation of the forward-backward algorithm to compute marginal likelihoods that contribute to the acceptance probabilities of the involutive MCMC kernels `add_remove_waypoint_kernel` and `add_remove_search_attempt_kernel`.



(a) Left: The continuous state space is discretized into cells. Middle: For each destination cell, there is a transition matrix that determines the distribution on the next cell given the current cell. The matrix was learned from data simulated from the original generative model, which uses a path planning algorithm. Right: The resulting generative model is a mixture of hidden Markov models.

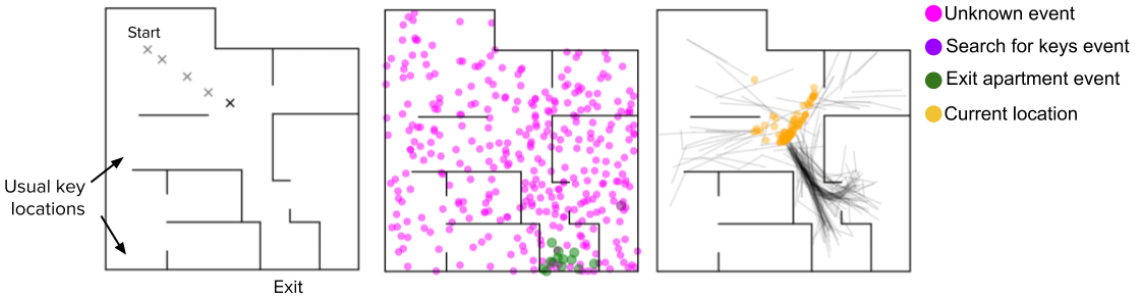


(b) Exact inference in the coarsened model via dynamic programming is efficient. The posterior distribution on the destination cell is shown for three observed data sets.

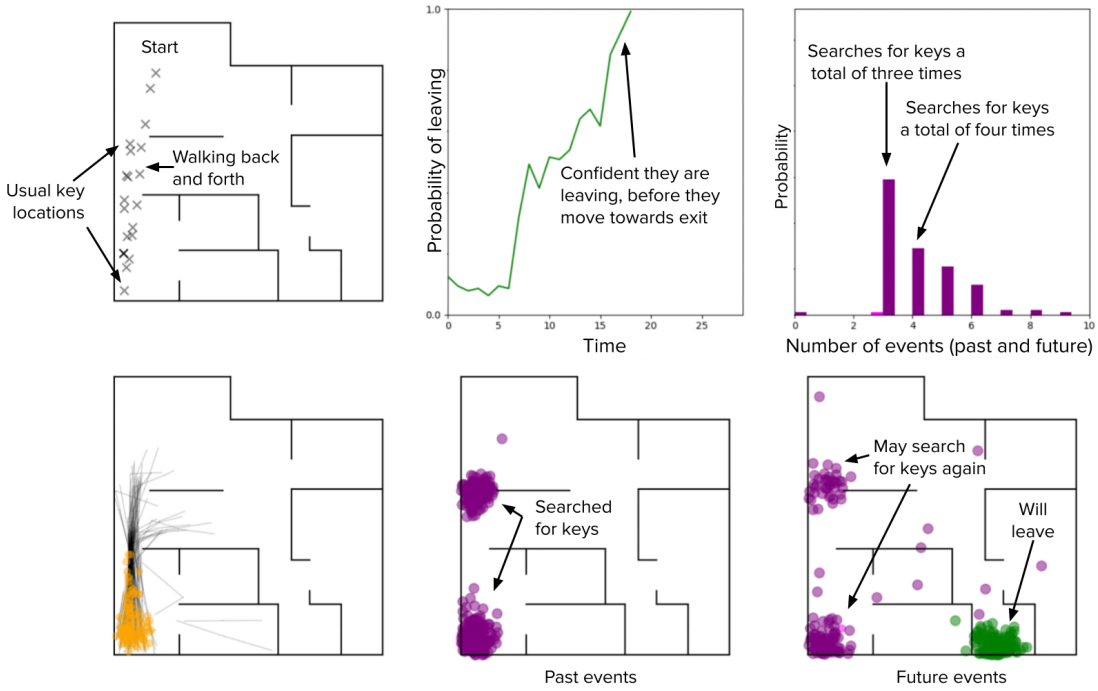
Figure 6-10: Using a coarse-grained surrogate model to aid inference in a fine-grained model

6.1.5 Symbolic reasoning from noisy data via probabilistic inference

Figure 6-11 shows inferences made using the inference code of Figure 6-5. First, note that the inferences provide much more than activity classification—the traces contain detailed information about both past and future events as well as predicting future trajectories. The inferences exhibit the combination of logical reasoning with uncertainty over symbolic structures (events) and inference from noisy data, capabilities that motivate use of universal probabilistic programming systems. However, performing inferences like this in nontrivial models requires more specialization of the inference code than allowed by existing universal probabilistic programming systems, which are largely based on generic inference algorithms.



(a) Observed data (left), predicted future events (middle), and predicted trajectories. Predicted future events for this observed data set include leaving the apartment (green), and visiting waypoints of unknown purpose (pink), but not searching for keys (purple). The inferences are intuitive—if the person did not have their keys, they would have walked towards the likely key locations, but instead they are walking directly towards the exit.



(b) An observed data set where the person first walks towards the likely key location in the lower left, and then to the other likely key location in the center-left, and then back again to the lower-left. The Gen inferences conclude that the person is probably searching for their keys and is planning to leaving the apartment, long before they actually start moving in the direction of the exit.

Figure 6-11: Inferring past events, and predicting future events and trajectories with Gen

6.2 Inferring object pose and existence from point clouds

Robust, accurate, and more human-like reasoning about scenes requires the ability to account for uncertainty due to occlusion. This section describes an application of Gen to 3D perception of objects from point cloud data under self-occlusion and full and partial occlusion by other objects. This section shows two example applications—inferring the pose of a self-occluded object with uncertainty quantification, and inference about the existence or non-existence of a fully occluded object based on Bayesian spatial reasoning about multiple objects in a scene. We take an approach based on 'vision as inverse graphics' [10], a paradigm for computer vision that is based on inverting the generative process of rendering to produce interpretable descriptions of an image in terms of discrete primitives like objects.

Bayesian inference of 6DoF object pose under self-occlusion Self-occlusion is a common source of uncertainty about object poses in 3D perception, especially for objects with articulation (e.g. human body) and objects with regularities or symmetries (e.g. a coffee mug that is rotationally symmetric, except for the handle). Figure 6-12a shows a Gen DML generative function that defines a generative model of the full positional and rotational pose (6DoF) of a rigid object and the point cloud generated by a depth camera facing the object. The pose is factored into a 3D Cartesian position, and a rotation that is implemented using a custom data type provided by the domain-specific modeling and inference library `Gen3DRot`. The data type uses a custom reference measure, and the `Gen3DRot` library provides a set of probability distributions with densities defined with respect to this reference measure, as well as a set of involutive MCMC moves (Section 3.7) for this data type. Each move proposes a specific common-sense mental rotation of the object (e.g. flipping or rotating around some axis of rotation). These moves allow construction of efficient MCMC algorithms for pose uncertainty quantification that exploit the structure of the objects. For example, for the mug used in this example, rotating around the z-axis (which is perpendicular to the base of the mug) is an efficient way to explore the posterior, which is highly concentrated along this one degree of freedom. Gen is designed to support an ecosystem of interoperable modeling and inference libraries like `Gen3DRot`. Multiple implementations of the likelihood model (`noise_model`) were tested, including a 3D and probabilistic extension of chamfer distance [9] between point clouds that marginalizes out the unknown correspondence between observed points and rendered points and outlier indicator variables for observed points.

Inference about the presence or absence of fully occluded objects Suppose a robot is tasked with assembling a piece of furniture, performing maintenance on a vehicle, or retrieving something from the kitchen. In each of these cases, the robot has a prior belief that some object (e.g. a tool, component, or kitchen item) is present in the environment. However, in unstructured environments are complex and cluttered, and the target object is likely to be fully occluded from the robot's view. That target object may be in fact absent from the environment, especially in human-robot interactive scenarios (e.g. the component or item is missing or misplaced). To act rationally in such situations, the robot will need to consider plausible poses of the object that can explain the *absence* of its percept. Also,

the robot must consider the possibility that the object is indeed not present, by weighing the lack of percept of the object against the prior probability that it is present or absent. We propose a principled proof-of-concept solution to these challenges based on Bayesian inference. In particular, we apply Bayesian inference in a generative model of point clouds with an interpretable latent representation that includes the existence and 6DoF pose of multiple objects. The generative model (Figure 6-13a) employs a 3D renderer to produce a synthetic point cloud from the latent variables, accounting for mutual occlusion of objects. The noise model is robust to outliers, which are common in real depth sensor data.

Most work on inverse graphics has focused on single objects, and therefore cannot be used to address existential uncertainty due to full occlusion. Work in this area that has considered occlusion has been limited to situations without existential uncertainty, where the set of objects that are present is known a-priori [62]. Perhaps one reason why structural and existential uncertainty has received less attention is the mathematical complexity that it introduces. Gen simplifies the implementation of Bayesian inference in the presence of structural uncertainty. Figure 6-13 shows the results of MCMC inference in Gen (inference program not shown) on several scenarios involving occlusion, using object models from the YCB 3D object data set [19]. The results illustrate Bayesian inference about the 6DoF pose and presence or absence of a fully occluded object, and investigate the dynamics of these inferences as the fraction of occluded volume in the scene is varied. In particular, the *lack of percept* of the mug in the visual field (i) reduces the posterior probability of its presence, but also (ii) informs the distribution on its 6DoF pose, if it is present. Note that as the fraction of volume in the scene that is occluded by the box increases, the posterior probability that mug is present in the environment increases. This reflects common sense—the more volume occluded by the box, the more places the mug could be hidden from view.

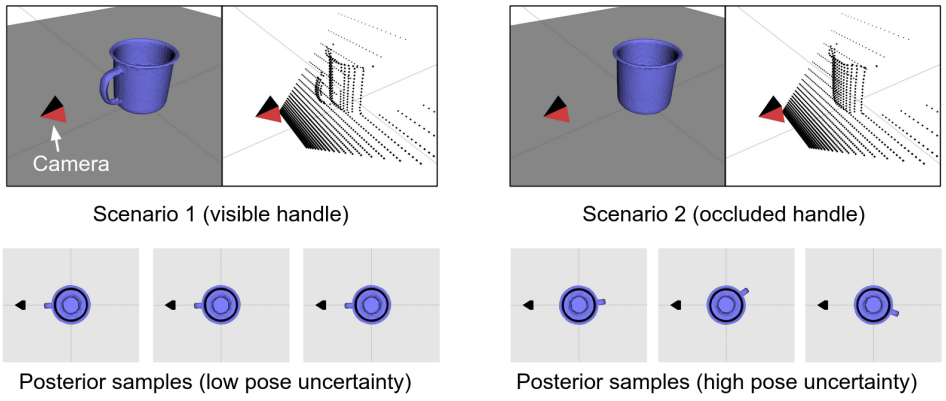
```
@gen function object_pose_prior()
    x ~ uniform(xmin, xmax)
    y ~ uniform(ymin, ymax)
    z ~ uniform(zmin, zmax)
    rot ~ Gen3DRot.uniform()
    return (x, y, z, rot)
end
```

```
@gen function mug_model()
    mug ~ object_pose_prior()
    points ~ render_depth([(mug, mug_mesh)])
    observed_points ~ noise_model(points)
end
```

(a) A generative model of the 6DoF pose of an object and point clouds produced by a depth camera. The 3D rotation (`rot`) is a quaternion sampled from the Haar measure on the group of unit quaternions. The `Gen3DRot` domain-specific modeling and inference library provides several probability distributions on 3D rotations.

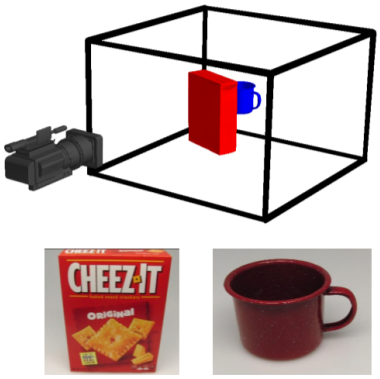
```
@kern pose_kernel(trace, addr)
   trace ~ Gen.mh(trace, position_random_walk, (0.01, addr))

   # flip over x-axis, then y-axis, then z-axis
   trace ~ Gen3DRot.flip_around_fixed_axis_mh(trace, addr=>:rot, [1,0,0])
   trace ~ Gen3DRot.flip_around_fixed_axis_mh(trace, addr=>:rot, [0,1,0])
   trace ~ Gen3DRot.flip_around_fixed_axis_mh(trace, addr=>:rot, [0,0,1])

   # rotate around z axis (perpendicular to base of mug)
   trace ~ Gen3DRot.uniform_angle_fixed_axis_mh(trace, addr=>:rot, [0,0,1])
   trace ~ Gen3DRot.small_angle_fixed_axis_mh(trace, addr=>:rot, [0,0,1], pi/16)

   # random small rotations to explore mode
   trace ~ Gen3DRot.small_angle_random_axis_mh(trace, addr=>:rot, pi/32)
   return trace
end
```

(b) A custom composite MCMC kernel for the pose of an object, parametrized by the address of the object's pose (for the model in (a), the `addr` = `mug`). The kernel uses reusable involutive MCMC moves on 3D rotations implemented in the domain-specific modeling and inference library `Gen3DRot`.



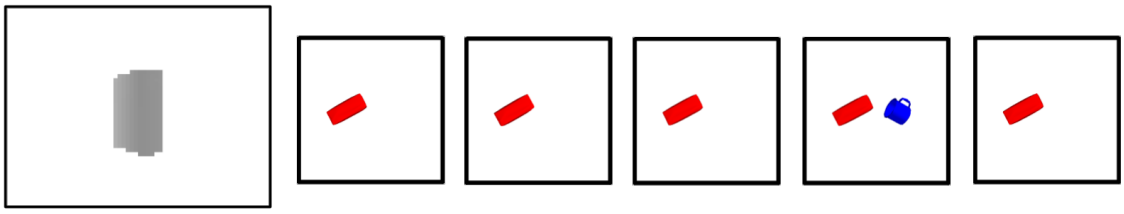
(c) Posterior inferences from a simulated point cloud. Left: Handle is visible, and the posterior is concentrated. Right: Handle is occluded, and there is uncertainty about the angle of the handle.

Figure 6-12: Using MCMC for Bayesian inference of 6DoF object pose from point clouds

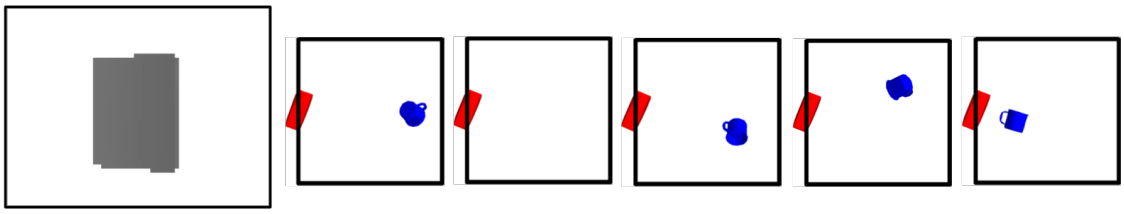



```
@gen function existential_doubt_model()
    objects = []
    if ({:box_present} ~ bernoulli(0.9))
        pose = ({:box} ~ object_pose_prior())
        push!(objects, (pose, box_mesh))
    end
    if ({:mug_present} ~ bernoulli(0.9))
        pose = ({:mug} ~ object_pose_prior())
        push!(objects, (pose, mug_mesh))
    end
    points = render_depth(objects)
    observed_points ~ noise_model(points)
end
```

(a) Left: A scenario in which a depth camera is viewing a scene that may or may not contain a cracker box and a mug. Only the cracker box is visible in the observed depth images (see below). Right: The generative model. The prior probability that the object exists is 0.9 for both objects.



(b) An observed simulated depth image and a subset of the posterior samples. The observed image has the box angled so that it occupies less of the field of view than in (c). The posterior probabilities that the mug and box are present are **0.33** and 1.0, respectively.



(c) An observed simulated depth image and a subset of the posterior samples. The observed image has the box closer to the camera, so that it occupies more of the field of view than in (b). The posterior probabilities that the mug and box are present are **0.63** and 1.0, respectively.

Figure 6-13: Inferring the presence, absence, and pose of multiple objects from point clouds

6.3 Real-time camera pose estimation

This section describes an application of Gen to real-time camera-pose estimation in an indoor environment, and inference of environment parameters; and examines the relative strengths and weaknesses of model based and neural network based inference in this setting. Specifically, the task is to estimate the height of the room, the elevation of the camera above the floor, and the pitch and roll angles of the camera in real-time from a stream of point cloud data generated by a depth camera. An online MCMC inference algorithm based on the following generative model of depth images was implemented in Gen:

```
@gen function cam_pose_prior(height)
   z ~ uniform(0.0, height)
   pitch ~ uniform(pi/4, 3*pi/4)
   roll ~ uniform(-pi/4, pi/4)
   return (0,0,z,quat(roll,pitch,0))
end
```

```
@gen function cam_pose_model()
   height ~ uniform(2.0, 3.0)
   cam_pose ~ cam_pose_prior(height)
   objects = [
    (plane, (0,0,0,quat(0,0,0))),
    (plane, (0,0,height,quat(0,0,0)))
   ]
   points = render_depth(objects, cam_pose)
   observed_points ~ noise_model(points)
end
```

The left of Figure 6-14 shows an input depth image frame from an indoor office scene where red pixels indicate missing data. The middle frame shows the inferred ceiling and floor depth pixels (blue) using the online Monte Carlo algorithm implemented with Gen. On the right is a visualization of synthetic data generated from the generative model. The generative model only models the floor and ceiling, and not any walls or other objects in the scene. The noise model uses an independent per-pixel mixture of a uniform distribution (to explain unmodeled objects and outliers) and normal distribution centered at the modeled depth at every pixel. A combination of random-walk and independent Metropolis-Hastings moves were used to obtain an initial pose estimate, and to track the pose over time by initializing the Markov chain for each new observation to the previous pose estimate. This can be interpreted as a sequential Monte Carlo algorithm (Section 3.5) with a single particle.

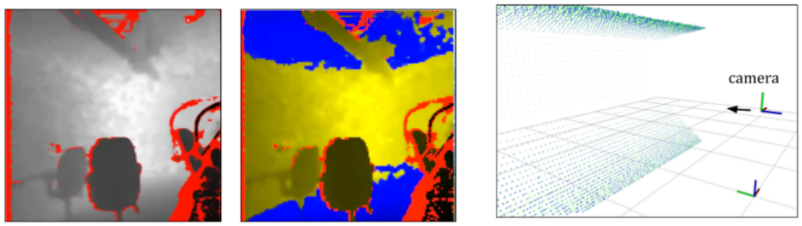
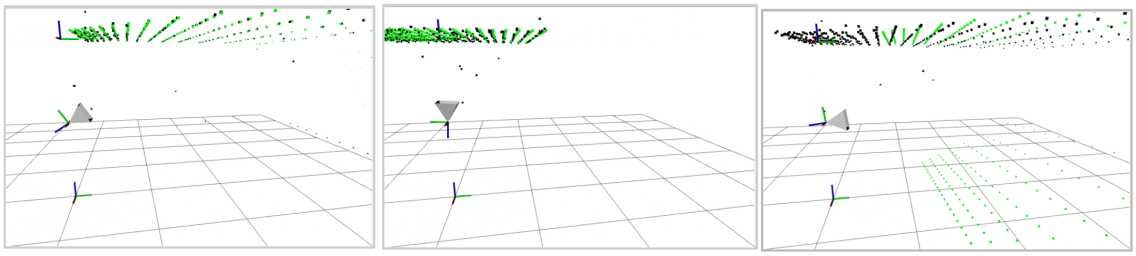


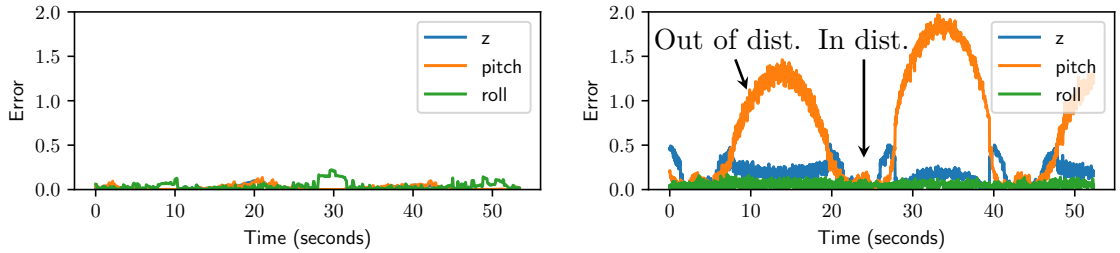
Figure 6-14: Tracking camera pose using online Monte Carlo in a generative model

It is also possible to use a 'bottom-up' neural approach to regress camera pose from depth data. To compare the tradeoffs between a neural approach and the online Monte Carlo approach, we implemented both approaches using Gen and compared them on different simulated data sets. For this experiment, the room height was fixed, so there were

three latent variables (`z`, `pitch`, and `roll`). Note that modeling the yaw is unnecessary since only the ceiling and floor are modeled, and they are modeled as infinite planes. The neural network was a multi-layer perceptron with one hidden layer with 20 hidden units implemented as a generative function in DML, and trained to regress from an observed depth image to the pose of the camera, using data simulated from the generative model (Section 3.3). The simulated data included range of pitch angles between $-\pi/4$ and $\pi/4$ (where 0 is looking straight ahead). As shown in Figure 6-15a and Figure 6-15b, the trained neural network performs reasonably for input data within its training distribution, but fails when given out-of-distribution data with pitch angles ranging from $-\pi/2$ to $\pi/2$ (that is, looking straight up at the ceiling or straight down at the floor). The model-based online Monte Carlo inference algorithm works with no re-training required. However, it is more computationally expensive than the neural network. Gen supports combining bottom-up and model-based methods. For example, when the neural network is used as a proposal within Monte Carlo, its proposals are rejected in the out-of-distribution regime where it is inaccurate. Gen is designed to support research into robust hybrid inference architectures.



(a) A neural network was trained to regress from an observed depth image to the pose of the camera, using data simulated from the generative model. Black points are observed data, green are reconstructed data. Left: The trained neural network performs well for new data that lies within its training distribution. Middle: When the distribution changes to include unfamiliar data (looking upwards), a model-based inference algorithm works with no re-training required. Right: The neural network fails for cases on which it was not trained. Gen is designed for combining bottom-up inference with model-based reasoning, which can give inference that is both robust and fast.

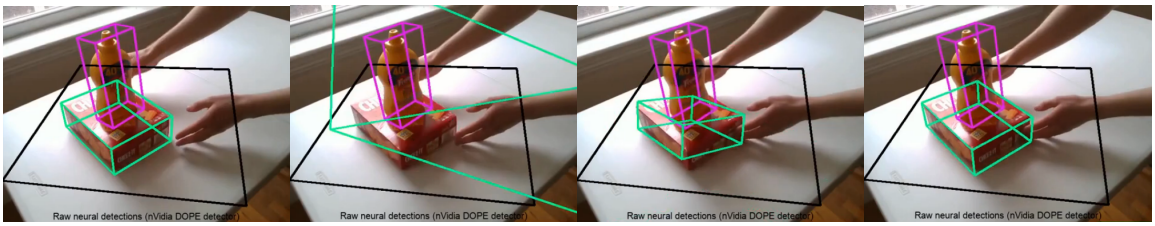


(b) Left: Error for the online Monte Carlo algorithm. Right: Error for a simple neural network. These two algorithms took comparable time (on the order of minutes) to implement using Gen; this does not not include the training time for the neural network, which was a few minutes.

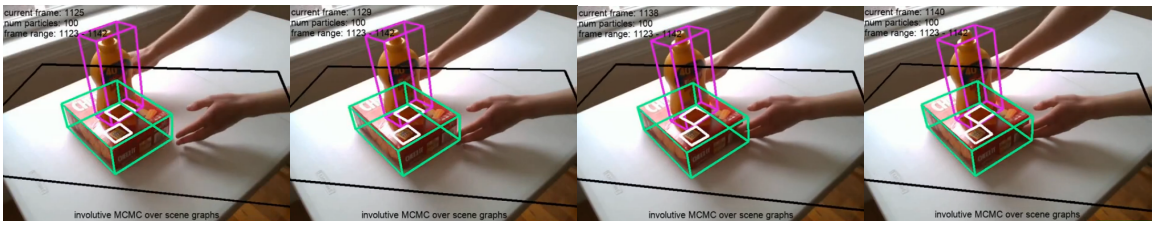
Figure 6-15: Comparing bottom-up and top-down inference approaches for pose estimation

6.4 Inferring the dynamic geometric structure of a 3D scene

This section describes an application of Gen to robust 3D scene understanding. While neural network techniques like Deep Object Pose Estimation [124] (DOPE), or heuristics like using RANSAC to fit object models to point clouds [114] can give approximate estimates of the 6DoF pose of individual objects, they do not take into account prior knowledge about the physical dynamics of scenes, or prior knowledge about the physical relationships between objects. Additional filtering or post-processing is needed to use these estimates for downstream tasks. For example, pose estimates from DOPE can be unreliable and frequently change drastically from frame to frame or drop out completely for certain frames during e.g. unfavorable or out-of-distribution lighting conditions. Developing more robust perception systems for embodied intelligence applications requires synthesizing more prior knowledge about scenes with perceptual data, and inference in probabilistic generative models of scenes is well-suited to this challenge. We used Gen to develop a probabilistic generative model and inference algorithm that synthesizes (i) heuristic 6DoF object pose estimates, (ii) prior knowledge about temporal dynamics of scenes, and (iii) prior knowledge about the physical relationships between objects to produce beliefs about the poses of objects in a scene and their physical relationships. Figure 6-16 shows example input data in the form of heuristic 6DoF pose estimates of three objects (cracker box, mustard bottle, and table surface) and the 6DoF pose estimates produced by our algorithm. For this video sequence, the algorithm is able to filter out incorrect estimates of the pose of the cracker box (green), and also infer that the mustard bottle is resting on the cracker box (white square 1) and that the cracker box is resting on the table (white square 2). The remainder of this section describes the generative model, the inference algorithm, and its implementation.



(a) Heuristic 6DoF object pose estimates at selected frames in a video



(b) For each frame in (a), the inferred 6DoF object poses and object-object contact planes

Figure 6-16: Probabilistic inference of scene graphs makes pose estimation more robust

Scene graphs The basis of our generative model is the *scene graph*, a directed tree that spans a set of coordinate frames. Our scene graph representation is inspired by the scene graph representations used for rendering in computer graphics and video games [123], as well as augmented and reality [101], and related structures in robotics [98]. These scene graphs typically parametrize a scene as a tree, rooted with a world coordinate frame, with nodes for objects and object parts. Edges in the tree encode relative 6DoF poses, that are typically generated by some lower-dimensional parametrization (e.g. rotating an elbow joint in an articulated model of a character body). In our framework, a scene graph is a directed tree, where the root node represents the world coordinate frame and each non-root represents one object. Any objects whose parent in the tree is the world coordinate frame are called *floating*, and objects whose parent is another object are called *sliding*. The 6DoF pose of an object that are floating is parametrized as a full 6DoF transformation (three dimensions of position, and a unit quaternion for orientation) from the world coordinate frame to the object's coordinate frame. The 6DoF pose of an object that is sliding is parametrized as a 3DoF relative pose transformation from the coordinate frame of one of the parent object's *faces* and one of the faces of the child object, where the faces of the two objects are sliding on one another with anti-parallel outward normal vectors. There are two degrees of translational freedom and one degree for rotation around the normal vectors. We use cuboids to model most objects. Given the scene graph, it is possible to compute the 6DoF pose of each object relative to the world coordinate frame by walking the graph from the root to the leaves.

A generative model for scene graphs and pose estimates Figure 6-17 shows a variant of the generative model, implemented in Gen's DML. The set of objects, and their mesh models, are assumed to be known a-priori. The generative model (`scene_graph_model`) first samples the structure of the scene graph, by sampling an undirected spanning tree over $n + 1$ nodes uniformly at random. The first node represents the world coordinate frame, and the remaining nodes represent objects in the scene. Then, we convert the undirected spanning tree into a directed spanning tree by forcing the world coordinate frame node to be the root of the tree. Next, the model samples the numerical parameters for each object node, which is either floating or sliding. The code for sampling the sliding object parameters assumes that both objects are cuboids and have six faces. Then, the scene graph is the tuple of the structure and parameters. Next, the model computes the 6DoF poses of each object relative to the world coordinate frame by walking the scene graph.[2] Consider the resulting prior probability distribution on world frame 6DoF poses for each object (the distribution on `world_poses`). This distribution is complex, and encodes the the prior knowledge that objects are often in face-to-face contact with one another. However, note that while interpenetration of one object by another is less likely under this prior than if each object was modeled independently, it is still not guaranteed because objects that are not connected directly in the tree may interpenetrate. Finally, a robust likelihood model models the heuristic estimates of each object's 6DoF pose relative to the world frame via

[2] When walking the scene graph, we also sample extra *slack* degrees of freedom for each sliding object (not shown). The slack variables make reversible jump MCMC inference over the structure more efficient.

a combination of zero-mean Gaussian noise in the position, von-Mises Fisher noise in the orientation, and outliers in both position and orientation.

```
@gen function float_params_prior()
  x ~ uniform(xmin, xmax)
  y ~ uniform(ymin, ymax)
  z ~ uniform(zmin, zmax)
  rot ~ Gen3DRot.uniform(),
  return (x, y, z, rot)
end
```

```
@gen function slide_params_prior()
  other_side ~ uniform_discrete(1, 6)
  my_side ~ uniform_discrete(1, 6)
  u ~ normal(0.0, side_len / 3)
  v ~ normal(0.0, side_len / 3)
  theta ~ uniform(0., 2 * pi)
  return (other_side, my_side, u, v, theta)
end
```

```
@gen function scene_graph_model(num_objects::Int)

  # generate the structure scene graph, a directed spanning tree
  num_nodes = num_objects + 1 # the world coordinate frame is a node
  undirected_spanning_tree ~ uniform_spanning_tree(num_nodes)
  structure = to_directed_tree(undirected_spanning_tree)

  # generate numerical parameters of the scene graph
  params = Dict()
  for object in 1:num_objects
     if object_is_floating(scene_graph_structure, object)
         params[object] = ({(object,:floating)} ~ float_params_prior())
     else
         params[object] = ({(object,:sliding)} ~ slide_params_prior())
     end
  end

  # construct scene graph from its structure and parameters
  scene_graph = (structure, params)

  # generate noisy measurement of each object pose relative to world frame
  world_poses = compute_world_poses(scene_graph)
  observations ~ noise_model(world_poses)
end
```

Figure 6-17: Using Gen to model the symbolic structure of a 3D scene with multiple objects

Inferring the scene graph via involutive MCMC Given a collection of labeled object pose estimates for each object produced by either DOPE (for YCB objects) or a RANSAC-based estimator that uses depth data (for the table-top), the inference algorithm produces approximate posterior samples of the scene graphs, in the form of traces of the model `scene_graph_model`. The inference algorithm is based on MCMC and is implemented using Gen. The MCMC algorithm uses a combination of data-driven proposals, random-walk moves on individual parameters, and a custom reversible jump MCMC move that is implemented using Gen's involutive MCMC construct (Section 3.7). The reversible jump proposal is illustrated in Figure 6-18. Given a graph G, with root note r and object nodes o_i for each i, the auxiliary generative function samples random node to sever from the tree (in this case o_5, shown in green), and a random other node on which to graft the first node (in this case, r, shown in pink). Note that this change to the structure is reversible, by using the same severed node but the previous parent (o_3) as the graft node. The involution transforms between the equivalent parametrizations of the pose of the severed object. The resulting inferences about contacts between objects arise naturally via Bayesian Occam's razor—if two objects are nearly in face-to-face contact, this is most parsimoniously explained via an edge between the two objects in the scene graph.

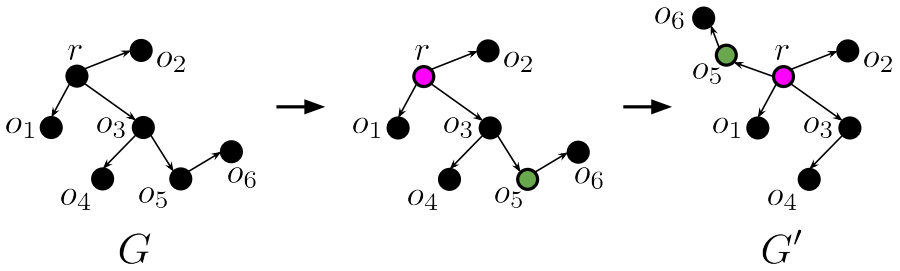


Figure 6-18: A transition kernel on scene graph structures

Adding temporal dynamics We augmented the model shown above with temporal dynamics over both the structure of the scene graph and its continuous parameters. Time is modeled as sequence of discrete steps. At each step, there is some probability that the scene graph structure changes. We model transitions on scene graph structure using a mixture of a uniform distribution on a new spanning tree (with low probability) and a local random walk on the space of spanning trees based on the same transition kernel shown in Figure 6-18. The transition model on continuous parameters is based on a random walk in parameter space. The inference algorithm for the temporal model uses sequential Monte Carlo (Section 3.5) with MCMC moves including structure-changing involutive MCMC moves used as rejuvenation kernels between new observations.

Integrating Gen and ROS This application also served as a proof-of-concept for the integration of Gen and the Robot Operating System [98] (ROS). In particular, the Gen inference algorithm was wrapped within a ROS node as part of a a real-time ROS application that also integrated a depth camera, the DOPE detector, and a visualizer.

6.5 Gaussian process structure learning for time series

Bayesian synthesis Gen's modeling languages are flexible enough to express prior distributions on programs. By combining a prior distribution on programs with a likelihood model that defines a probability distribution on data for each possible program, we obtain a generative model over program code and data. Inferring the source code of the program via approximate sampling from the conditional distribution given observed data is called *Bayesian synthesis* [109]. Bayesian synthesis can be tractable when the space of programs is defined using a small domain-specific language.

Inferring the structure and parameters of a covariance function Consider the task of time series forecasting. Time series often have symbolic explanations; the complex patterns in time series arise from a composition of simpler patterns [54]. Modeling the symbolic structures (e.g. periodicity, trends, changepoints) in the data allows for much more accurate extrapolation compared to autoregressive methods. Saad et al. [109] give a domain-specific expression language for covariance functions of a Gaussian process (GP) that builds on a class of models proposed by [54] and describe MCMC algorithms for Bayesian inference over expressions in this language, conditioned on an observed time series. The generative model for this Bayesian synthesis problem is expressed in Gen's Dynamic Modeling Language in Figure 3-19. The model first samples an expression in a domain-specific expression language that encodes the covariance function. The prior distribution on expressions is based on a probabilistic context free grammar (PCFG). Then, the model samples additional white noise to add to the data and computes the covariance matrix from the covariance function, inputs (in this case a vector of times at which data will be sampled), and finally samples the vector of output data over all times points from a multivariate normal distribution. The prior distribution on covariance matrices is defined in a DML generative function recursively generates subexpressions from the grammar of covariance functions. An alternative Gen implementation of this generative model that uses the Static Modeling Language and the Recurse Combinator is discussed in Section 5.3.

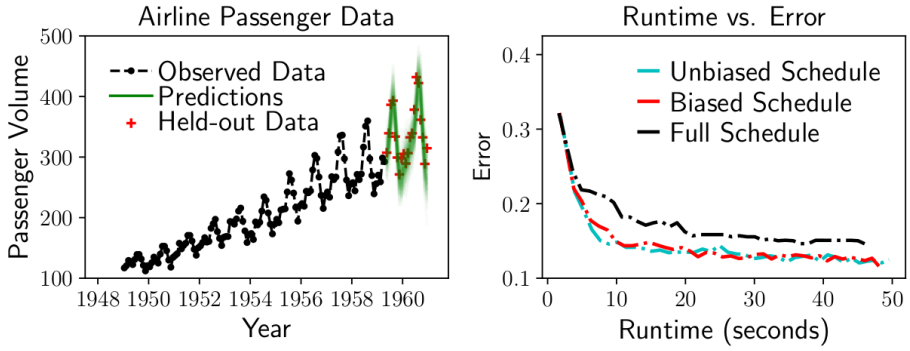


Figure 6-19: Experimenting with different MCMC schedules using Gen

Varying the MCMC kernel schedule Several variations of the MCMC inference algorithm of Saad et al. [109] were implemented using Gen's involutive MCMC construct (Section 3.7), and the code implementing these is shown and discussed in Section 3.7.5. Notably, the involutive MCMC construct gives the user fine-grained control over structure-changing MCMC moves. In this case, we use this control to explore the effect of different variants of an MCMC kernel that proposes to replace subtrees of the covariance function parse tree. Figure 6-19 shows an example data set, forecasts for this data set based on the inferred distribution on covariance functions, and a comparison in the time-accuracy profiles of three different MCMC algorithms. One algorithm (full) re-proposes the entire tree from the root at each iteration, and is less efficient. The other two schedules only re-propose subtrees (one samples subtrees uniformly in the graph, and the other biases the sampling towards subtrees at smaller depths). Note that Gen automatically computes all acceptance probabilities for these moves. Also, Gen's involutive MCMC construct prevented a bug in the acceptance probability calculation in one of the baseline hand-coded implementations of this algorithm. This bug involved accounting for the probability of selecting a subtree, which is difficult to reason about but automated by Gen's involutive MCMC construct.

Chapter 7

Conclusion

7.1 Tradeoffs in probabilistic inference systems architecture

A central insight from the research into Gen's design is that the design space of software systems for probabilistic inference is large and subject to fundamental tradeoffs, and that systems designed for practical use must intentionally and carefully balance these tradeoffs. This section explains some of these tradeoffs, situates Gen in the design space, and outlines areas for future work.

Balancing automation and user control Probabilistic inference systems must choose which of the implementation choices that arise in probabilistic inference to automate, and what choices to leave to the user. Users must also choose what level of control they need for their application when evaluating which system to use. At one extreme are modeling languages with built-in 'solvers' that allow users to declaratively specify their model, perhaps set some parameters of the algorithm, and then run the algorithm [20] (systems that automatically construct the model from input data are even further down this spectrum but out of scope of this thesis). Near the other extreme, the inference practitioner may only use a library of standard probability distributions. They will derive the updates and other primitive operations in the inference algorithm from their pencil-and-paper mathematical formulation of the model, and translate these operations directly into a numerical program.

We can better understand the design space by recognizing the different dimensions of automation and control: First, because inference algorithms are often computationally intensive and take significant time to run, some users may require control over the high-level flow of the process, or how it is distributed across parallel computing resources. Other users may want these decisions to be made for them by the system. Second, the system may use built-in algorithms [51], or may allow the user to customize a schedule of different built-in operations used in the algorithm, or may allow the user to add custom heuristics to the algorithm, or may require the user to define the algorithm themselves. Finally, the system might automate the low-level implementation details like density evaluations and gradients, or might require the user to implement the these themselves. Many recent probabilistic programming systems have emphasized automation over user control [51, 52, 130], while those in the 'programmable inference' paradigm [80] have emphasized the importance of

user control over the algorithm.

One approach to sidestepping the tradeoff between automation and user control is to design multiple levels of interfaces, with more user control at the bottom level and more automation at the top level, such that users can smoothly transition between levels as their needs evolve. This is the approach taken in Gen: At the top level, Gen's inference library includes an implementation of self-normalized importance sampling with the automatically-generated internal proposal family (Section 4.2) that requires the user to write a single line of code and provide a single parameter that governs the amount of computation to use. One level down, if the user needs to instrument the loop of the algorithm, or distribute across their infrastructure in a custom way, they can re-implement the loop of the algorithm, which invokes the GENERATE operation and requires on the order of 10 lines of code. If the user then needs more efficient inference, they can employ a custom proposal distribution written as a probabilistic program instead of the built-in proposal family (Section 3.2), and train it on simulated data (Section 3.3), which might require on the order of 100 lines of code. Or, they can employ MCMC (Section 3.4) or sequential Monte Carlo (Section 3.5 and Section 3.6) techniques, each of which has more and less automated variants. Finally, if the user wants to manually Rao-Blackwellize certain random variables in their model, they can implement custom generative function and trace data types for that part of their model (Section 5.4). Crucially the user can navigate this design space incrementally and smoothly, without having to rewrite their application from scratch, sacrifice the separation between the model and the inference code, or switch platforms.

User control is required when it is impractical to automatically meet the user's requirements. In probabilistic inference, meeting performance requirements often requires customization of the algorithm. While automatic adaptation and learning of parameters in models and inference algorithms is possible in Gen (e.g. Section 3.3), Gen does not attempt to automatically generate an inference algorithm that is customized to the model. Instead, Gen delegates this task to the user, because devising efficient custom inference algorithms often draws on prior knowledge of the problem or related problems and knowledge of relevant heuristics. Automatically generating efficient algorithms specialized to a probabilistic model is a long-term research challenge that will build on expertise in probabilistic modeling and inference, machine learning, and program synthesis. This task subsumes the design of efficient heuristics for use in proposals, schedules of MCMC kernels, reversible jump moves, annealing schedules, coarse-to-fine inference schemes, and algorithms for neural network architecture search. A truly automated system for constructing inference algorithms must search over structured, hierarchical and combinatorial spaces of inference programs. Therefore, we might define this as *inference program synthesis*. Gen lays groundwork for this research by providing a set of primitives from which inference programs are composed. One important subproblem is building probabilistic models of the efficiency of inference programs on inference problems that capture the same sort of intuitions used by human inference algorithm experts. This line of research is closely related to metareasoning [108].

Achieving good inference performance while managing complexity A central task in software architecture is designing interfaces that manage the complexity of applications without preventing them from meeting their functional requirements. Because of

the computational intensity of probabilistic inference, meeting performance requirements of applications can be challenging. Doing so with high-level inference code based on an explicit and easy-to-evolve model representation is even more challenging. Gen's abstract data types strike a particular balance between the simplicity of their interfaces and the performance that can be achieved using these interfaces.

The most important design choice in Gen is to use an explicit definitive representation of the model, in the form of a generative function, that separates the low-level computations associated with the model from the inference algorithm code. This choice is critical for managing complexity of an inference application—it makes the application much easier to maintain and modify. Additionally, by automatically compiling the generative function from a probabilistic program, we considerably reduce the surface area for bugs in low-level computations associated with the model. This choice does however have consequences for performance. For example, potential cancellations between a proposal distribution's density and a model's density cannot be exploited (unless the two are re-implemented in a custom generative function with a custom internal proposal family). Devising intermediate representations for probabilistic models that expose more information about the model and therefore allow for more optimizations than Gen's abstract data types is an interesting area for future work. However, for a practical system it will be important to ensure users can smoothly transition from an implementation based on black-box data types to one based on analysis of intermediate model representations. These two are not in conflict; indeed it should be possible to extend Gen's generative function abstract data type with operations that return intermediate representations and similarly extend Gen's inference library with functions that analyze these representations and generate optimized inference code.

Another design choice in Gen that aims to balance complexity and performance is the immutability of traces. Immutability of traces allows for a single trace data type to be used for all inference algorithms, including sequential Monte Carlo algorithms that interleave duplication of particles with rejuvenation MCMC moves (Section 3.5). For example, implementing a resample-move particle filter from scratch requires careful tracking of references to memory resources that are shared by multiple particles and careful dynamic allocation and copying. In Gen this complexity is handled by low-level persistent functional data structures [37] that are used within trace implementations. However, not all algorithms make used of the immutability of Gen's traces. For example, unconstrained optimization algorithms can be implemented using in-place updates, and algorithms that include accept and reject steps can also be implemented using in-place updates without allocations. Similarly, Gen's trace data type does not distinguish between updates that change the structure (i.e. control flow) of a trace and those that do, which have different potential low-level implementation strategies. While some performance improvements are possible (and have been implemented) via just-in-time compilation of trace operations (Section 5.2), extending Gen's trace data type with mutable variants and more generally giving users some intermediate level of control over a trace's internal data structure and representation so they can optimize their inference applications without having to write a custom generative function and trace data type is an important next step. However, these extensions should be optional both for users and generative function implementers and should be designed with an awareness of the costs of increasing system complexity.

7.2 Generative and discriminative models and heuristics

A core design requirement for the architecture described in this thesis is that it can support multiple algorithmic paradigms for probabilistic inference. In particular, we distinguish between *discriminative* and *model-based* probabilistic inference. We do not use 'discriminative model' and 'generative model' here because generative models and simulators can be used offline to train discriminative models. We use 'model-based inference' to refer to processes that query an explicitly represented generative model *at inference time* and 'discriminative inference' to refer to processes that do not, and define both as producing an approximation to the conditional distribution of latent quantities given observed quantities. Discriminative inference, which is sometimes called 'bottom-up inference', is often based on discriminative probabilistic models like deep neural networks or random forests trained on real or simulated data, but can also be based heuristics, as in using least-squares within a robust Bayesian regression inference problem (Section 3.2). A discriminative inference algorithm can be interpreted as a type of 'amortized inference' [45], because it is constructed based on experience with other related inference problems.

Gen is designed to support algorithms that combine discriminative and model-based inference approaches. In Gen, both discriminative models and generative models are represented as generative functions and are typically constructed by writing probabilistic programs. Discriminative models can be used to initialize an iterative model-based sampling or search algorithm like MCMC in a generative model, or as a proposal distribution within various Monte Carlo algorithms. Generative models can also be used to generate synthetic data for training discriminative models. Using probabilistic programs to represent both statistical discriminative models like neural networks and proposal distributions based on heuristics clarifies their equivalence, and their relationship to model-based inference. In particular it suggests a methodology [30] based on fitting the stochasticity of a proposal distribution centered on the output of an arbitrary heuristic to data simulated from a generative model and then employing this proposal within Monte Carlo algorithms (Section 3.3).

Gen's abstractions also shed light on and encourage use of principled inference methodologies based on 'surrogate' generative models. Like discriminative models, surrogate generative models can be interpreted as a type of heuristic from the perspective of model-based inference. But also like discriminative models, they can be made less heuristic by training their parameters on data generated from the generative model, and can be used to accelerate asymptotically exact inference in generative models via sequential Monte Carlo and trace translators (Section 3.6). Indeed, there is a close mathematical relationship between a surrogate generative model used as an intermediate target distribution in sequential Monte Carlo and a discriminative model used as a proposal distribution in importance sampling.

A generative model can serve as a sort of *specification* for a discriminative model or a surrogate generative models, and there are well-defined metrics based on Kullback-Leibler divergence for quantifying how well a discriminative or surrogate model meets the specification (Section 3.3). Because all of these types of models are expressed explicitly as generative functions in Gen, it is straightforward to evaluate and optimize these metrics as part of a coherent engineering methodology.

7.3 Towards a mature inference engineering methodology

This thesis is intended to be a step towards an inference engineering methodology grounded in interpretable generative models that is accessible to a broader community of programmers and engineers who do not have extensive mathematical training in probabilistic modeling and inference. Progress must be made along several directions to reach this goal.

First, we need general-purpose libraries for probabilistic modeling and inference that use explicit representations of structured generative models in human-readable modeling languages, and flexibly support various algorithmic methodologies including Monte Carlo, variational, and discriminative approaches and their hybrids, at a high level of abstraction. These libraries should be based on compositional model representations with common interfaces that encourage modular modeling and inference code and reuse of modeling and inference components. Gen aims to be such a library. Several other high-level modeling and inference libraries based on Gen's data types and interoperable with Gen have been developed.

Tools for evaluating, testing, and verifying inference implementations relative to specifications defined by generative models are needed. Empirical analyses based on samples from inference implementations [120, 110], static analyses of inference implementations [73, 7, 72] that check for key invariants and identify bugs, and techniques that combine aspects of static analysis and sampling [26] all have a role to play. For performance-constrained applications like robotics where safety and uncertainty quantification are important, tools for evaluating the non-asymptotic approximation error of inference implementations and discriminative models using metrics grounded in probability theory are needed [26]. These tools need to be flexible enough to evaluate errors on specific data sets and distributions of data sets, search for data sets for which algorithms fail, and collect data used to build empirical models of inference algorithm performance.

Finally, we need to make this engineering methodology learnable without relying on mathematical knowledge of probability theory. Gen has been used to each multiple classes on applied probabilistic inference for which prior coursework in probabilistic inference was not required with encouraging results, but more work is needed to systematize practical knowledge about how to implement, debug, test, and maintain inference implementations.

Notation

a, b	Scalars and addresses in choice dictionaries (lowercase)
$\mathbf{v}, \mathbf{g}$	Vectors (bold)
$\boldsymbol{M}, \boldsymbol{J}$	Matrices (bold italicized uppercase)
A, B	Sets (uppercase)
$\varnothing$	Empty set
A^{C}	Complement of set A
$\boldsymbol{\rho}, \boldsymbol{\sigma}, \boldsymbol{\tau}, \boldsymbol{\upsilon}, \boldsymbol{\lambda}$	Choice dictionaries (bold lowercase Greek letters)
$\mathfrak{p}, \mathfrak{q}$	Probabilistic modeling language source code
$\mathcal{P}, \mathcal{Q}$	Generative functions
$\mathbf{t}, \mathbf{s}$	Traces (bold lowercase sans-serif font)
$\boldsymbol{\tau}[a]$	Value lookup in choice dictionary
$\boldsymbol{\sigma} \oplus \boldsymbol{\tau}$	Merge of two choice dictionaries
$A_{\boldsymbol{\tau}}$	Set of addresses in a choice dictionary (its domain)
$\boldsymbol{\tau}\vert_A$	Restriction of a choice dictionary to a set of addresses
$\{a \mapsto 2, b \mapsto 3\}$	Literal choice dictionary
$\mathcal{T}_A^\star$	Set of choice dictionaries with addresses that are a subset of A
$\mathcal{T}_A$	Set of choice dictionaries with addresses that are exactly A
$p(\boldsymbol{\tau}), q(\boldsymbol{\sigma}; x)$	Probability density of choice dictionary
$\text{supp}(p)$	Support of density p (the set $\{\boldsymbol{\tau} : p(\boldsymbol{\tau}) > 0\}$)
$\mu(d\boldsymbol{\tau})$, $\nu(d\boldsymbol{\tau})$	Probability measure on dictionaries
$\bar{p}(\boldsymbol{\rho})$	Marginal likelihood
$p(\boldsymbol{\sigma}\vert\boldsymbol{\rho})$	Conditional probability density
$\boldsymbol{\tau} \sim p$, $\boldsymbol{\sigma} \sim q(\cdot; \boldsymbol{\sigma})$	Sampling from the measure induced by probability density
$[\![\mathfrak{p}]\!]$	Semantics of source code (a generative function)
$[x]$	Indicator; 1 if x is true and 0 if x is false
$p_{\text{norm}(\mu,\boldsymbol{\sigma})}$	Density function for a primitive probability distribution

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Umut A. Acar. Self-adjusting computation: (an overview). In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 1–6, 2009.

[3] David J Aldous. Exchangeability and related topics. In *École d'Été de Probabilités de Saint-Flour XIII-1983*, pages 1–198. Springer, 1985.

[4] Christophe Andrieu and Gareth O. Roberts. The pseudo-marginal approach for efficient Monte Carlo computations. *Ann. Statist.*, 37(2):697–725, 04 2009.

[5] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50(1-2):5–43, 2003.

[6] Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle Markov Chain monte carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.

[7] Eric Atkinson, Cambridge Yang, and Michael Carbin. Verifying handcoded probabilistic inference procedures. *arXiv preprint arXiv:1805.01863*, 2018.

[8] Chris L Baker, Rebecca Saxe, and Joshua B Tenenbaum. Action understanding as inverse planning. *Cognition*, 113(3):329–349, 2009.

[9] H. G. Barrow, J. M. Tenenbaum, R. C. Bolles, and H. C. Wolf. Parametric correspondence and chamfer matching: Two new techniques for image matching. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'77, pages 659–663, 1977.

[10] Bruce Guenther Baumgart. *Geometric Modeling for Computer Vision.* PhD thesis, Stanford, CA, USA, 1974.

[11] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio.

Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, pages 1–7, 2010.

[12] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[13] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978, 2019.

[14] Serena Booth, Yilun Zhou, Ankit Shah, and Julie Shah. Bayes-probe: Distribution-guided sampling for prediction level sets. *arXiv preprint arXiv:2002.10248*, 2020.

[15] Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 33–46, 2016.

[16] Jörg Bornschein and Yoshua Bengio. Reweighted wake-sleep. *arXiv preprint arXiv:1406.2751*, 2014.

[17] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of Markov Chain Monte Carlo*. CRC press, 2011.

[18] Yufei Cai, Paolo G Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 145–155, 2014.

[19] Berk Calli, Arjun Singh, Aaron Walsman, Siddhartha Srinivasa, Pieter Abbeel, and Aaron M Dollar. The YCB object and model set: Towards common benchmarks for manipulation research. In *2015 International Conference on Advanced Robotics (ICAR)*, pages 510–517. IEEE, 2015.

[20] Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1): 1–32, 2017.

[21] Sourav Chatterjee, Persi Diaconis, et al. The sample size required in importance sampling. *The Annals of Applied Probability*, 28(2):1099–1135, 2018.

[22] Siddhartha Chib and Edward Greenberg. Understanding the Metropolis-Hastings algorithm. *The American Statistician*, 49(4):327–335, 1995.

[23] Mary Kathryn Cowles, Gareth O Roberts, and Jeffrey S Rosenthal. Possible biases induced by MCMC convergence diagnostics. *Journal of Statistical Computation and Simulation*, 64(1):87–104, 1999.

[24] Marco Cusumano-Towner. Inference library of the Gen probabilistic programming system. `https://github.com/probcomp/Gen.jl/blob/b9d72b/src/inference/mh.jl#L73-L108`, 2018. Accessed: 2018-12-27.

[25] Marco Cusumano-Towner and contributors. Gen: A general-purpose probabilistic programming system with programmable inference. `https://www.gen.dev`. Accessed: 2020-08-28.

[26] Marco Cusumano-Towner and Vikash K. Mansinghka. AIDE: An algorithm for measuring the accuracy of probabilistic inference algorithms. In *Advances in Neural Information Processing Systems 30*, pages 3000–3010, 2017.

[27] Marco Cusumano-Towner, Benjamin Bichsel, Timon Gehr, Martin Vechev, and Vikash K. Mansinghka. Incremental inference for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 571–585. ACM, 2018.

[28] Marco Cusumano-Towner, Alexander K Lew, and Vikash K Mansinghka. Automating involutive MCMC using probabilistic and differentiable programming. *arXiv preprint arXiv:2007.09871*, 2020.

[29] Marco F Cusumano-Towner and Vikash K Mansinghka. Encapsulating models and approximate inference programs in probabilistic modules. *arXiv preprint arXiv:1612.04759*, 2016.

[30] Marco F Cusumano-Towner and Vikash K Mansinghka. Using probabilistic programs as proposals. *arXiv preprint arXiv:1801.03612*, 2018.

[31] Marco F Cusumano-Towner, Alexey Radul, David Wingate, and Vikash K Mansinghka. Probabilistic programs for inferring the goals of autonomous agents. *arXiv preprint arXiv:1704.04977*, 2017.

[32] Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 221–236. ACM, 2019.

[33] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential Monte Carlo samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(3): 411–436, 2006.

[34] Joshua V Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A Saurous. Tensorflow distributions. *arXiv preprint arXiv:1711.10604*, 2017.

[35] Arnaud Doucet, Nando De Freitas, and Neil Gordon. An introduction to Sequential Monte Carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer, 2001.

[36] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973.

[37] James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[38] David Duvenaud, James Robert Lloyd, Roger Grosse, Joshua B. Tenenbaum, and Zoubin Ghahramani. Structure discovery in nonparametric regression through compositional kernel search. In *Proceedings of the 30th International Conference on Machine Learning*, ICML 2010, pages 1166–1174, 2013.

[39] Ted Enamorado, Benjamin Fifield, and Kosuke Imai. Using a probabilistic model to assist merging of large-scale administrative records. *American Political Science Review*, 113(2):353–371, 2019.

[40] Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: A language for flexible probabilistic inference. volume 84 of *Proceedings of Machine Learning Research*, pages 1682–1690. PMLR, 09–11 Apr 2018.

[41] Roland Gecse and Attila Kovács. Consistency of stochastic context-free grammars. *Mathematical and Computer Modelling*, 52(3-4):490–500, 2010.

[42] 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.

[43] Andreas Geiger, Martin Lauer, and Raquel Urtasun. A generative model for 3D urban scene understanding from movable platforms. In *CVPR 2011*, pages 1945–1952, 2011.

[44] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (6):721–741, 1984.

[45] Samuel Gershman and Noah Goodman. Amortized inference in probabilistic reasoning. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 36, 2014.

[46] Samuel J Gershman, Eric J Horvitz, and Joshua B Tenenbaum. Computational rationality: A converging paradigm for intelligence in brains, minds, and machines. *Science*, 349(6245):273–278, 2015.

[47] John Geweke. Getting it right: Joint distribution tests of posterior simulators. *Journal of the American Statistical Association*, 99(467):799–804, 2004.

[48] Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. A language and program for complex bayesian modelling. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 43(1):169–177, 1994.

[49] Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. A language and program for complex Bayesian modelling. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 43(1):169–177, 1994.

[50] Walter R Gilks and Carlo Berzuini. Following a moving target-Monte Carlo inference for dynamic Bayesian models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(1):127–146, 2001.

[51] Noah Goodman, Vikash Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Proceedings of the 24th Annual Conference on Uncertainty in Artificial Intelligence*, UAI 2008, pages 220–229, 2008.

[52] Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2016-10-31.

[53] Peter J Green. Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika*, 82(4):711–732, 1995.

[54] Roger B. Grosse, Ruslan Salakhutdinov, William T. Freeman, and Joshua B. Tenenbaum. Exploiting compositionality to explore a large space of model structures. In *Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence*, UAI 2012, pages 306–315, 2012.

[55] Shivam Handa, Vikash Mansinghka, and Martin Rinard. Compositional inference metaprogramming with convergence guarantees. *arXiv preprint arXiv:1907.05451*, 2019.

[56] David I Hastie and Peter J Green. Model choice using reversible jump Markov chain Monte Carlo. *Statistica Neerlandica*, 66(3):309–338, 2012.

[57] Keith Hawkins, Boris Leistedt, Jo Bovy, and David W Hogg. Red clump stars and Gaia: Calibration of the standard candle using a hierarchical probabilistic model. *Monthly Notices of the Royal Astronomical Society*, 471(1):722–729, 2017.

[58] GE Hinton, P Dayan, BJ Frey, and RM Neal. The "wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995. ISSN 0036-8075.

[59] Tomas Hrycej. Gibbs sampling in bayesian networks. *Artificial Intelligence*, 46(3): 351–363, 1990.

[60] Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. Compiling Markov chain Monte Carlo algorithms for probabilistic modeling. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 111–125, 2017.

[61] Jonathan H Huggins, Trevor Campbell, Mikołaj Kasprzak, and Tamara Broderick. Practical bounds on the error of bayesian posterior approximations: A nonasymptotic approach. *arXiv preprint arXiv:1809.09505*, 2018.

[62] Varun Jampani, Sebastian Nowozin, Matthew Loper, and Peter V Gehler. The informed sampler: A discriminative approach to Bayesian inference in generative computer vision models. *Computer Vision and Image Understanding*, 136:32–44, 2015.

[63] Claus Skaanning Jensen, Augustine Kong, and Uffe Kjaerulff. Blocking Gibbs sampling in very large probabilistic expert systems. *International Journal of Human Computer Studies*, 42(6):647–666, 1995.

[64] Charles Kemp, Joshua B Tenenbaum, Thomas L Griffiths, Takeshi Yamada, and Naonori Ueda. Learning systems of concepts with an infinite relational model. In *AAAI*, volume 3, page 5, 2006.

[65] Bjarne Knudsen and Jotun Hein. Pfold: RNA secondary structure prediction using stochastic context-free grammars. *Nucleic Acids Research*, 31(13):3423–3428, 2003.

[66] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT press, 2009.

[67] Tejas D Kulkarni, Pushmeet Kohli, Joshua B Tenenbaum, and Vikash Mansinghka. Picture: A probabilistic programming language for scene perception. In *Proceedings of the ieee conference on computer vision and pattern recognition*, pages 4390–4399, 2015.

[68] Karim Lari and Steve J Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer speech & language*, 4(1):35–56, 1990.

[69] Steven M. LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical Report TR 98-11, Computer Science Department, Iowa State University, 1998.

[70] Tuan Anh Le, Atılım Günes Baydin, and Frank Wood. Inference compilation and universal probabilistic programming. *arXiv preprint arXiv:1610.09900*, 2016.

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

[72] Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. Towards verified stochastic variational inference for probabilistic programs. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–33, 2019.

[73] Alexander K Lew, Marco F Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K Mansinghka. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019.

[74] Alexander K. Lew, Benjamin Sherman, Marco Cusumano-Towner, Michael Carbin, and Vikash Mansinghka. MetaPPL: Inference algorithms as first-class generative models. *Languages for Inference Workshop*, 2020. URL `https://popl20.sigplan.org/details/lafi-2020/14/MetaPPL-Inference-Algorithms-as-First-Class-Generative-Models`.

[75] Barbara Liskov and Stephen Zilles. Programming with abstract data types. *ACM Sigplan Notices*, 9(4):50–59, 1974.

[76] Jun S. Liu. *Monte Carlo Strategies in Scientific Computing*. Springer Publishing Company, Inc., 2001.

[77] James Robert Lloyd, David Duvenaud, Roger Grosse, Joshua Tenenbaum, and Zoubin Ghahramani. Automatic construction and natural-language description of nonparametric regression models. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[78] Anthony Lu. Venture: An extensible platform for probabilistic meta-programming. Master's thesis, Massachusetts Institute of Technology, 2016.

[79] Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: A higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

[80] Vikash K Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. Probabilistic programming with programmable inference. In *ACM SIGPLAN Notices*, volume 53, pages 603–616. ACM, 2018.

[81] David Merrell and Anthony Gitter. Inferring signaling pathways with probabilistic programming. *Proceedings of the Nineteenth European Conference of Computational Biology*, 2020.

[82] David Merrell and Anthony Gitter. Inferring signaling pathways with probabilistic programming, 2020.

[83] Brian Milch and Stuart Russell. General-purpose MCMC inference over relational structures. In *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence*, pages 349–358, 2006.

[84] Brian Milch, Bhaskara Marthi, and Stuart Russell. BLOG: Relational modeling with unknown objects. In *ICML 2004 workshop on statistical relational learning and its connections to other fields*, pages 67–73, 2004.

[85] Brian Milch, Bhaskara Marthi, David Sontag, Stuart Russell, Daniel L Ong, and Andrey Kolobov. Approximate inference for infinite contingent Bayesian networks. In *Proceedings of the 10th International Conference on Artificial Intelligence and Statistics*, AISTATS 2005, pages 238–245, 2005.

[86] Dave Moore and Maria I Gorinova. Effect handling for composable program transformations in Edward2. *arXiv preprint arXiv:1811.06150*, 2018.

[87] Quaid Morris. Recognition networks for approximate inference in BN20 networks. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, pages 370–377, 2001.

[88] Iain Murray, Ryan Adams, and David MacKay. Elliptical slice sampling. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, AISTATS 2010, pages 541–548, 2010.

[89] Lawrence M Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B Schön. Delayed sampling and automatic Rao-Blackwellization of probabilistic programs. *arXiv preprint arXiv:1708.07787*, 2017.

[90] Praveen Narayanan and Chung-chieh Shan. Symbolic disintegration with a variety of base measures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 42(2):1–60, 2020.

[91] Radford M. Neal. Annealed importance sampling. *Statistics and Computing*, 11(2): 125–139, 2001.

[92] Radford M Neal et al. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2(11):2, 2011.

[93] Kirill Neklyudov, Max Welling, Evgenii Egorov, and Dmitry Vetrov. Involutive MCMC: A Unifying Framework. *arXiv preprint arXiv:2006.16653*, 2020.

[94] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8026–8037, 2019.

[95] Avi Pfeffer. Ibal: A probabilistic rational programming language. In *IJCAI*, pages 733–740, 2001.

[96] Du Phan, Neeraj Pradhan, and Martin Jankowiak. Composable effects for flexible and accelerated probabilistic programming in NumPyro. *arXiv preprint arXiv:1912.11554*, 2019.

[97] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 315–328, 1989.

[98] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, volume 3, page 5. Kobe, Japan, 2009.

[99] L. Rabiner and B. Juang. An introduction to hidden Markov models. *IEEE ASSP Magazine*, 3(1):4–16, 1986.

[100] Jonathan Rees and William Clinger. Revised^3 report on the algorithmic language Scheme. *ACM Sigplan Notices*, 21(12):37–79, 1986.

[101] Dirk Reiners. *OpenSG: A scene graph system for flexible and efficient realtime rendering for virtual and augmented reality applications*. PhD thesis, Darmstadt University of Technology, 2002.

[102] Sylvia Richardson and Peter J Green. On Bayesian analysis of mixtures with an unknown number of components (with discussion). *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 59(4):731–792, 1997.

[103] 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*, AISTATS 2016, pages 28–37.

[104] Christian P. Robert and George Casella. *Monte Carlo Statistical Methods*. Springer Texts in Statistics. Springer-Verlag, 2005.

[105] David A Roberts, Marcus Gallagher, and Thomas Taimre. Reversible jump probabilistic programming. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics*, AISTATS 2019, pages 634–643, 2019.

[106] Andrew Roth, Jiarui Ding, Ryan Morin, Anamaria Crisan, Gavin Ha, Ryan Giuliany, Ali Bashashati, Martin Hirst, Gulisa Turashvili, Arusha Oloumi, et al. JointSNVMix: a probabilistic model for accurate detection of somatic mutations in normal/tumour paired next-generation sequencing data. *Bioinformatics*, 28(7):907–913, 2012.

[107] Stuart Russell. Rationality and intelligence: A brief update. In *Fundamental issues of artificial intelligence*, pages 7–28. Springer, 2016.

[108] Stuart Russell and Eric Wefald. Principles of metareasoning. *Artificial intelligence*, 49(1-3):361–395, 1991.

[109] Feras A. Saad, Marco Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages*, 3(POPL):37:1–37:29, 2019.

[110] Feras A Saad, Cameron E Freer, Nathanael L Ackerman, and Vikash K Mansinghka. A family of exact goodness-of-fit tests for high-dimensional discrete distributions. *arXiv preprint arXiv:1902.10142*, 2019.

[111] Hiroaki Sakoe and Seibi Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE transactions on acoustics, speech, and signal processing*, 26(1):43–49, 1978.

[112] Tetsuya Sato, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu. Formal verification of higher-order probabilistic programs. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[113] Ulrich Schaechtle, Feras Saad, Alexey Radul, and Vikash K. Mansinghka. Time series structure discovery via probabilistic program synthesis. *arXiv preprint arXiv:1611.07051*, 2016.

[114] Ruwen Schnabel, Roland Wahl, and Reinhard Klein. Efficient RANSAC for point-cloud shape detection. In *Computer Graphics Forum*, volume 26, pages 214–226. Wiley Online Library, 2007.

[115] Adam Ścibior, Zoubin Ghahramani, and Andrew D Gordon. Practical probabilistic programming with monads. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, pages 165–176, 2015.

[116] Span Spanbauer, Cameron Freer, and Vikash Mansinghka. Deep involutive generative models for neural MCMC. *arXiv preprint arXiv:2006.15167*, 2020.

[117] Geir Storvik. On the flexibility of Metropolis–Hastings acceptance probabilities in auxiliary variable proposal generation. *Scandinavian Journal of Statistics*, 38(2):342–358, 2011.

[118] Andreas Stuhlmüller, Jacob Taylor, and Noah Goodman. Learning stochastic inverses. In *Advances in Neural Information Processing Systems 26*, pages 3048–3056, 2013.

[119] Andreas Stuhlmüller, Robert XD Hawkins, N Siddharth, and Noah D Goodman. Coarse-to-fine sequential monte carlo for probabilistic programs. *arXiv preprint arXiv:1509.02962*, 2015.

[120] Sean Talts, Michael Betancourt, Daniel Simpson, Aki Vehtari, and Andrew Gelman. Validating Bayesian inference algorithms with simulation-based calibration. *arXiv preprint arXiv:1804.06788*, 2018.

[121] Luke Tierney. Markov chains for exploring posterior distributions. *The Annals of Statistics*, pages 1701–1728, 1994.

[122] Luke Tierney. A note on Metropolis-Hastings kernels for general state spaces. *Annals of Applied Probability*, 8(1):1–9, 02 1998.

[123] Robert F Tobler. Separating semantics from rendering: a scene graph based architecture for graphics applications. *The Visual Computer*, 27(6-8):687–695, 2011.

[124] Jonathan Tremblay, Thang To, Balakumar Sundaralingam, Yu Xiang, Dieter Fox, and Stan Birchfield. Deep object pose estimation for semantic robotic grasping of household objects. In *Conference on Robot Learning (CoRL)*, 2018.

[125] Zhuowen Tu and Song-Chun Zhu. Image segmentation by data-driven Markov chain Monte Carlo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24 (5):657–673, 2002.

[126] Martin J Wainwright and Michael I Jordan. Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, 1(1-2):1–305, 2008.

[127] David Wingate, Andreas Stuhlmüller, and Noah Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, AISTATS 2011, pages 770–778, 2011.

[128] Sam Witty, Alexander Lew, David Jensen, and Vikash Mansinghka. Bayesian causal inference via probabilistic program synthesis. *arXiv preprint arXiv:1910.14124*, 2019.

[129] Sam Witty, Kenta Takatsu, David Jensen, and Vikash Mansinghka. Causal inference using gaussian processes with structured latent confounders. *arXiv preprint arXiv:2007.07127*, 2020.

[130] Frank Wood, Jan Willem Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics*, AISTATS 2014, pages 1024–1032, 2014.

[131] Yi Wu, Lei Li, Stuart Russell, and Rastislav Bodik. Swift: Compiled inference for probabilistic programming languages. *arXiv preprint arXiv:1606.09242*, 2016.

[132] Lingfeng Yang, Patrick Hanrahan, and Noah Goodman. Generating efficient MCMC kernels from probabilistic programs. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics*, AISTATS 2014, pages 1068–1076.

[133] Tan Zhi-Xuan, Jordyn L Mann, Tom Silver, Joshua B Tenenbaum, and Vikash K Mansinghka. Online Bayesian goal inference for boundedly-rational planning agents. *arXiv preprint arXiv:2006.07532*, 2020.

[134] Ben Zinberg, Marco Cusumano-Towner, and Vikash K Mansinghka. Structured differentiable models of 3D scenes via generative scene graphs. *Workshop on Perception as Generative Reasoning, NeurIPS 2019, Vancouver, Canada.* URL `https://pgr-workshop.github.io/img/PGR025.pdf`.