On Programming Languages for Probabilistic Modeling

a dissertation presented
by
Daniel E. Huang
to
The Department of Computer Science

in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the subject of
Computer Science

Harvard University
Cambridge, Massachusetts
May 2017

Thesis advisor: Professor Greg Morrisett Daniel E. Huang

On Programming Languages for Probabilistic Modeling

Abstract

The problem of probabilistic modeling and inference, at a high-level, can be viewed as constructing a (*model*, *query*, *inference*) tuple, where an *inference* algorithm implements a *query* on a *model*. Notably, the derivation of inference algorithms can be a difficult and error-prone task. Hence, researchers have explored how ideas from *probabilistic programming* can be used to simplify the construction of these tuples. For instance, given a probabilistic model expressed in some probabilistic modeling language and a query, the derivation of an inference algorithm can be automated by implementing an appropriate program transformation. Such a modeling language will typically provide continuous distributions. Hence, there is no direct connection with traditional, discrete notions of computation. Nevertheless, there is an intuitive understanding of probabilistic programs in terms of sampling.

In this dissertation, I investigate aspects of probabilistic programming languages (PPLs) in two parts. In the first part, I show how (Type-2) computable distributions can be used to give both distributional and (algorithmic) sampling semantics to a high-level, PCF-like language extended with continuous distributions. One motivation for using computable distributions, as opposed to more generally measures, is so that we can think of a Turing-complete PPL as expressing computable distributions. In the second part, I describe a compiler that generates Markov Chain Monte Carlo (MCMC) inference sampling algorithms for a language called AugurV2, which expresses a class of fixed-structure, parametric models. To manage the large

design space of MCMC algorithms and implementations, I propose a sequence of intermediate languages that enable a compiler to gradually and successively refine a declarative description of a probabilistic model and a query for posterior samples into an executable inference algorithm. The compilation strategy produces composable MCMC algorithms for execution on the CPU and GPU. I will also show how to use the semantics based on computable distributions to justify aspects of AugurV2's design and implementation.

Contents

This is the dedication.

Acknowledgments

Lorem ipsum dolor sit amet, sdf

1

Introduction

Consider the problem of clustering a set of datapoints in D-dimensional Euclidean space $\mathbb{R}^D$ into K clusters. One approach is to construct a *probabilistic (generative) model* to explain how we believe the observations are generated. For instance, one explanation might be that there are (1) K cluster centers choosen randomly according to a Normal distribution and (2) that each datapoint (independently of each other datapoint) is Normally distributed around a randomly choosen cluster center. This describes what is known as a Gaussian Mixture Model (GMM).

A probabilistic model induces a probability distribution, which we can *query* for quantities of interest. For example, the query "what is the most likely cluster assignment of each observation under the GMM model?" formulates our original problem of clustering a collection of points in probabilistic terms. To answer such a query, practioners implement an *inference algorithm*. Inference is analytically intractable in general. Consequently, practitioners devote a significant amount of time to developing and implementing approximate inference algorithms to answer a query for a given model.

More generally, we can think of the problem of probabilistic modeling and inference as constructing a (*model*, *query*, *inference*) tuple, where the *inference* algorithm implements a *query* on a *model*. Probabilistic models have applications in a wide range of fields, including in biology [25] and robotics [95]. One reason for their success is that probabilistic models can express both modeling assumptions (*e.g.*, causality or locality) and uncertainty (*e.g.*, noise) uniformly in the language of probability and statistics. However, the richness of the modeling framework also increases the difficulty of *probabilistic inference*, *i.e.*, the task of discovering patterns in the observed data according to the model. To simplify this task, researchers have explored how ideas from *probabilistic programming* can be applied. For instance, given a probabilistic model expressed in some probabilistic modeling language and a query, the derivation of an inference algorithm can be automated by implementing an appropriate program transformation. Due to the promise of this approach, many *probabilistic programming languages* (PPLs) have been proposed in the literature to explore the various points of the design space [94, 65, 39, 105].

In this dissertation, I study aspects of PPLs in two parts. In the first part, I investigate the *semantics* of a probabilistic modeling language extended with continuous distributions. In

particular, I show how (Type-2) computable distributions can be used to give both distributional and (algorithmic) sampling semantics to a PCF-like language extended with a probability monad called λ_{CD}. One reason for using computable distributions, as opposed to more generally measures, is so that we can interpret a Turing-complete probabilistic modeling language as expressing computable distributions. Hence, we can ground our study of these langauges on traditional notions of computation. In the second part, I investigate the *compilation* of a probabilistic modeling language and a query for posterior samples into Markov Chain Monte Carlo (MCMC) inference algorithms. In particular, I describe a system called AugurV2 whose domain-specific modeling language can be used to express fixed-structure Bayesian networks. The AugurV2 modeling language is a restriction of λ_{CD}, and hence, shares a common semantic foundation based on (Type-2) computable distributions. To guide the compilation process, I propose a sequence of intermediate languages that enable a compiler to gradually and successively refine a *declarative* description of a probabilistic model and a query for posterior samples into an *executable* inference algorithm. The compiler generates composable MCMC algorithms for execution on the CPU and GPU.

Remark (Scope). I focus exclusively on PPLs designed for expressing *probabilistic generative models* and for performing *Bayesian inference* in this dissertation. Thus, unless explicitly stated, I will interchangeably use the terms *probabilistic generative model* and *probabilitic model*, and *probabilistic inference* and *Bayesian inference*. In general, the terms *probabilistic modeling* and *probabilistic inference* can refer to non-generative probabilistic models (*e.g.*, Markov Random Fields) and other inference paradigms (*e.g.*, Frequentist).

Remark (Audience). The dissertation is written for an audience that is familiar with basic programming langauges concepts. Thus, I will assume more background in programming lan-

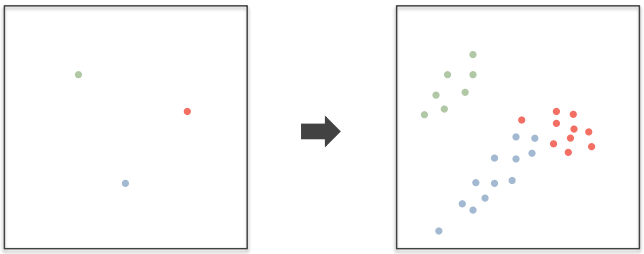
guage theory (*e.g.*, as in Gunter [42]) compared to background on probability and statistics.

1.1 Probabilistic Modeling and Inference

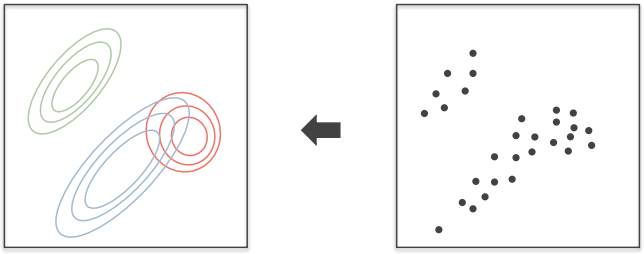
To set the stage for probabilistic programming, we begin with a short introduction to *probabilistic (generative) models* and *Bayesian inference*. Throughout this section, we will use the GMM we saw earlier (a model that clusters N points in D-dimensional Euclidean space into K clusters) to illustrate some of the ideas.

Figure 1.1a provides an illustration of a 2-dimensional ($2D$) GMM with 3 clusters as a probabilistic generative model. Intuitively, a *probabilistic generative model* describes a data-generation process, where we use distributions to encode assumptions about the structure of the process. The data-generation process begins on the left, where we sample 3 cluster locations (colored red, green, and blue). Then we proceed to the right, where we (1) select the cluster location (*i.e.*, pick a color) and then (2) sample a data point from a multivariate Gaussian distribution centered at the appropriate cluster location.

At a high-level, probabilistic inference is concerned with the reverse process, *i.e.*, determining the hidden or underlying structure of the data-generation process that produces the observations. Figure 1.1b illustrates the task of determining the locations of the 3 clusters given observed data for a GMM. On the right, we have N data points that we wish to cluster, as indicated by the absence of color. Moving to the left, we have an example of the results of probabilistic inference, which gives a distribution on the locations of the 3 clusters. In the rest of the section, we will introduce representations of probabilistic models (Section 1.1.1) and the difficulties of probabilistic inference (Section 1.1.2) more concretely, and then introduce the probabilistic programming approach (Section 1.2).



(a) An illustration of a Gaussian Mixture Model's generative process, starting with creating cluster means and shapes (left), and ending with colored datapoints (right), where the color indicates the cluster.



(b) An illustration of the results of probabilistic inference (left) to determine the distribution on cluster centers given observed data (right) for a GMM.

Figure 1.1: A Gaussian Mixture Model (GMM) and the results of probabilistic inference.

1.1.1 Abstractions for Probabilistic Models

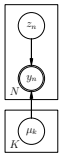
To precisely define a probabilistic model, we can use abstractions developed by statisticians and machine learning researchers. Three standard abstractions include random variables, probabilistic graphical models (PGMs) such as Bayesian networks, and densities. Figure 1.2 encodes a GMM using each of the three abstractions. As we will see, the representations encode different aspects of the same GMM.

Random variables Random variable notation (1.2a) can be read intuitively in terms of sampling. For example, the statement

$$\mu_k \sim \mathcal{N}(\mu_0, \Sigma_0) \text{ for } 1 \leq k \leq K$$

$$\mu_k \sim \mathcal{N}(\mu_0, \Sigma_0) \;\text{ for } 1 \le k \le K$$
$$z_n \sim \mathcal{D}(\pi_1, \ldots, \pi_K) \;\text{ for } 1 \le n \le N$$
$$y_n \mid z_n, \mu \sim \mathcal{N}(\mu_{z_n}, \Sigma) \;\text{ for } 1 \le n \le N$$

(a) random variables



(b) Bayesian network (plate notation)

$$p(\mu, z, y) = \prod_{k=1}^{K} p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu_k) \prod_{n=1}^{N} p_{\mathcal{D}(\pi)}(z_n) \prod_{n=1}^{N} p_{\mathcal{N}(\mu_{z_n}, \Sigma)}(y_n)$$

(c) densities

Figure 1.2: Three different ways of encoding a K-cluster GMM for clustering N datapoints in D-dimensional Euclidean space. We write π as shorthand for refering to a vector of values $(\pi_1, \ldots, \pi_K)$. The (unbound) variables μ_0, Σ_0, π, and Σ are known as model hyper-parameters and are constants.

can be read: "the random variable μ_k takes on a value according to the multivariate Gaussian distribution with mean μ_0 and covariance Σ_0 for $1 \le k \le K$." The entire (generative) process can be read as follows:

1. $\mu_k \sim \mathcal{N}(\mu_0, \Sigma_0)$ for $1 \le k \le K$: Sample the location $\mu_k \in R^D$ of each cluster from a multivariate Gaussian distribution centered at μ_0 with covariance Σ_0.
2. $z_n \sim \mathcal{D}(\pi_1, \ldots, \pi_K)$ for $1 \le n \le N$: Sample N cluster assignments z_n, each from a discrete distribution with weights π.
3. $y_n \mid z_n, \mu \sim \mathcal{N}(\mu_{z_n}, \Sigma)$: Generate the observation y_n as a sample from a multivariate Gaussian distribution centered at the corresponding cluster (*i.e.*, centered at μ_{z_n}) and shape (*i.e.*, the covariance matrix Σ).

The probabilistic model is given by the *joint distribution* of all the random variables defined, *i.e.*, the distribution of the random vector (μ, z, y). As notation, we will refer to a vector of values $(x_1, \ldots, x_N)$ as x. For example, we write μ to refer to the vector $(\mu_1, \ldots, \mu_K)$ in the GMM.

BAYESIAN NETWORK Bayesian networks (1.2b) are a representation of a probability distribution designed specifically for performing probabilistic inference. A Bayesian network expresses *conditional independence relations*, *i.e.*, how random variables influence one another, using a directed acyclic graph (DAG). The Bayesian network in the figure uses *plate notation*, which repeats the structure of the graph the number of times indicated by the plate (lower, right-hand side). The graph representation opens the possibility to use graph-based algorithms to perform inference. For example, message passing is an algorithm that can be used to compute the marginal distribution of each node in a Bayesian network, using the graph's edges to efficiently exchange information between the appropriately connected nodes. The graph structure is also useful for Gibbs sampling, a MCMC algorithm that leverages conditional independence relations to locally sample variables in the graph.

DENSITY FACTORIZATION The density factorization of a GMM (1.2c) is an alternative representation of a probability distribution that is useful for performing inference. For example, the GMM model density can be used to define a *log-likelihood function*

$$\begin{aligned}\mathcal{L}_{y^*}(\mu, z) &= \log p(\mu, z, y^*) \\ &= \sum_{k=1}^{K} \log p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu_k) + \sum_{n=1}^{N} \log p_{\mathcal{D}(\pi)}(z_n) + \sum_{n=1}^{N} \log p_{\mathcal{N}(\mu_{z_n}, \Sigma)}(y_n^*)\end{aligned}$$

which is a function of the parameters (μ, z) for given data (y^*). The notation y^* indicates that it is an observed value of the random variable y. The notation $p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu_k)$ indicates the density of a (multivariate) normal distribution with mean μ_0 and covariance Σ_0 evaluated at μ_k. It is common (*e.g.*, in statistics) to write $p(\mu_k \mid \mu_0, \Sigma_0)$ to mean $p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu_k)$, although this notation loses information about which distribution is involved.

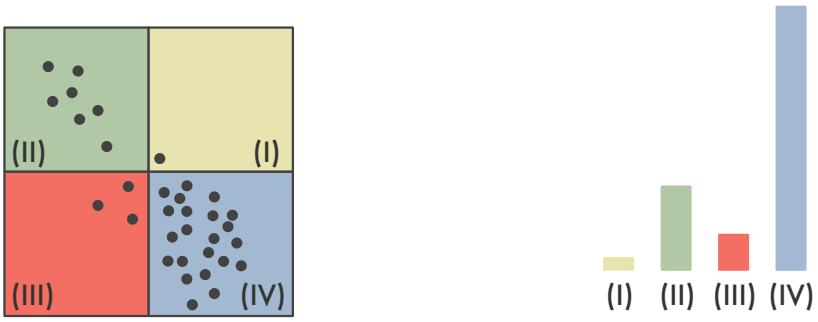


Figure 1.3: The Monte Carlo simulation approach (left) to probabilistic inference. After we simulate points from the distribution, we can construct an empirical histogram (right) by partitioning the space (*e.g.*, into quadrants) and count the number of points in each partition to determine the relative likelihood. In this example, quadrant (IV) is the most likely, while quadrant (I) is the least likely.

Given a density, inference can be cast as an optimization problem (*e.g.*, using Expectation Maximization to maximize the log-likelihood). This gives a point estimate (written $\hat{\cdot}$) of the cluster means $\hat{\mu}$ and cluster assignments $\hat{z}$. We could also use the log-likelihood in a Metropolis-Hastings (MH) sampling algorithm, a kind of Markov Chain Monte Carlo (MCMC) algorithm, to learn the posterior distribution $p(\mu, z \mid y^*)$ following a Bayesian approach. This produces a distribution over the cluster means μ and cluster assignments z instead of a single point estimate. Although densities are more convenient for expressing inference, it can be more difficult to express high-level properties about distributions. For example, to express that the sum of two Gaussians is Gaussian using densities, we would need to write a convolution

$$p_{\mathcal{N}(\mu_X+\mu_Y,\sigma_X^2+\sigma_Y^2)}(z) = \int p_{\mathcal{N}(\mu_X,\sigma_X^2)}(z)\, p_{\mathcal{N}(\mu_Y,\sigma_Y^2)}(z-t)\, dt\,.$$

1.1.2 Probabilistic Inference

Now that we have a probabilistic model, we can *query* it for quantities of interest. Below, we give two informal examples of queries and inference algorithms that can be used to answer

them (approximately).

- What are the most likely cluster locations under the GMM model given the observations? To answer this, we might apply the Expectation Maximization algorithm to solve for
$$\arg\max_{\mu} \sum_{z} \mathcal{L}_{y*}(\mu, z)$$
given observed data y^*.
- What is the distribution on the location of the first cluster given the observations under the GMM model? To answer this, we might apply Markov Chain Monte Carlo (MCMC) sampling to obtain samples from the distribution $p(\mu, z \mid y^*)$ (where again, y^* is observed data).

Note that we could use MCMC sampling to answer a variation of the first query. For instance, an algorithm could return the sample with the highest log-likelihood from a collection of samples drawn via MCMC. Thus, in general, there are many kinds of inference algorithms that can be used to approximately answer a kind of query. MCMC sampling can also be used to answer expectation queries, *i.e.*, queries for the average value according to some distribution by returning the average value of a collection of samples. Hence, in general, a family of inference algorithms (such as MCMC) can be used to approximate answers to multiple kinds of queries. In this dissertation, we focus exclusively on MCMC sampling or simulation approaches to probabilistic inference.

Figure 1.3 illustrates a (Monte Carlo) sampling approach to probabilistic inference. The idea is to approximate the shape of a distribution over a parameter space by sampling from the distribution and constructing an empirical histogram. We can construct an empirical histogram from a collection of samples by partitioning the parameter space and compute the relative likelihood of each partition as the ratio of the number of samples in a partition to the total number of samples. In the example, partition IV is the most likely, while partition I is the least likely.

Sampling becomes increasingly difficult as the number of dimensions increases. In particular, the number of "unit boxes" required to fill a D-dimensional box of with side-length 2 grows exponentially in D. Consequently, if we hold the number of samples fixed, then the average density of samples in each unit box decreases exponentially. This issue is colloquially known as the "curse of dimensionality" and affects other approaches to probabilistic inference in addition to sampling.

Probabilistic models with high-dimensional spaces are encountered frequently in practice. For example, the D-dimensional GMM we introduced earlier has $K \times D + N$ parameters—$K \times D$ parameters for the K cluster means, each of which has D parameters, and N parameters for the N cluster assignments. Notably, the number of parameters in a GMM increases with the size of the data as each new data point introduces a corresponding cluster assignment. To address the problems associated with inference in high-dimensional spaces, researchers typically take advantage of model-specific structure and use heuristics to make inference tractable. We list several below.

1. The conditional independence relationships in a model enable us to consider sampling in locally, low-dimensional spaces. For example given the locations of all the cluster means in the GMM, every cluster assignment is conditionally independent of each other. That is, sampling the N-D space of cluster assignments can be reduced to sampling N, 1-D spaces given the cluster locations. Consequently, representations of distributions such as PGMs are a staple of probabilistic modeling in practice [52].
2. First-order structure such as gradient information can be used to guide the exploration of the parameter space. For example, MCMC algorithms such as Hamiltonian Monte Carlo (HMC) [18] use the gradient of the model log-likelihood to explore the parameter space.
3. A common heuristic is to apply different inference methods to different subsets of the parameter space. For example, we may partition a model into its discrete variables and continuous variables, and apply an appropriate method to each subset. Such a heuristic would apply to a GMM, which has continuous-valued cluster locations and discrete-valued cluster assignments.

1.2 Probabilistic Programming

Ideas from *probabilistic programming* can be used to simplify the construction of (*model*, *query*, *inference*) tuples. For instance, given a model written in a probabilistic modeling language and a query, we can automate the derivation of an inference algorithm by implementing an appropriate program transformation. In the rest of the section, we will compare probabilistic modeling using a PPL with the standard setting to introduce some of the advantages of this approach as well as the challenges that arise.

Modeling language Figure 1.4 gives an example of the GMM written in a PPL with Python-like syntax. Observe that the program has similar structure to the representation of a probability distribution written in random variable notation. Indeed, the step-by-step procedure in which a probabilistic generative model can be described lends itself to being naturally expressed in a programming language. Hence, one benefit of probabilistic programming is that we can use a *formal language* to express models. Nevertheless, one challenge is to provide reasoning principles that give us assurance that encodings of probabilistic models in the modeling language correspond to the models we write with standard abstractions. An additional challenge is to reconcile the *discrete* nature of a programming language with the (possibly) *continuous* nature of models that we can write down using traditional probability and statistics.

Implementation As we have hinted at several times already, probabilistic programming enables us to phrase the problem of deriving and implementing an inference algorithm in terms of compilation. In particular, the representation of a probabilistic model in a modeling language opens up the ability to use language and compiler technology to improve the

```
def GMM(D, K, N, mu_0, Sigma_0, Sigma
    )(mu, z)(y):
  for k <- 0 until K:
    mu[k] = MvNorm(mu_0, Sigma_0)
  for n <- 0 until N:
    z[n] = Disc(pis)
    y[n] = MvNorm(mu[z[n]], Sigma)
```

(a) The encoding of a GMM in a probabilistic modeling language with Python syntax.

$$\mu_k \sim \mathcal{N}(\mu_0, \Sigma_0) \text{ for } 1 \leq k \leq K$$
$$z_n \sim \mathcal{D}(\pi_1, \ldots, \pi_K) \text{ for } 1 \leq n \leq N$$
$$y_n \mid z, \mu \sim \mathcal{N}(\mu_{z_n}, \Sigma) \text{ for } 1 \leq n \leq N$$

(b) Expressing a GMM using random variable notation.

Figure 1.4: A GMM encoded in a probabilistic modeling language with Python-like syntax (left). For comparison, the random variable notation is provided on the right.

efficiency of inference, similar to how using PGMs to represent distributions have opened up the possibility to use graph algorithms to improve the efficiency of inference. Furthermore, practioners use different representations of probability distributions for different purposes—random variables are used to summarize probabilistic models, densities are used for deriving mathematical properties useful for inference, and PGMs are used for representing conditional independence relations. A language-based approach would cast the usage of different abstractions as using different languages, and using compilation to automate the transformations between them. This breaks the construction of an inference algorithm into more modular and manageable pieces. Nevertheless, as probabilistic inference is analytically intractable in general, designing the appropriate languages and transformations so that the resulting system can perform approximate inference efficiently (for a class of models) is a huge challenge.

As a step towards making probabilistic programming more practical, we examine the semantics of probabilistic modeling languages and the implementation of a domain-specific PPL that automates inference in this dissertation. For the purposes of this dissertation, we will abbreviate the phrase (and similar phrases) "implementing a PPL that automates inference" to

simply "implementating a PPL". We overview some of the current solutions to the semantics and implementation of PPLs as well as introduce our approach now.

1.2.1 Semantics

A natural starting point is to interpet probabilistic programs as a sampler (*e.g.*, [39, 105]). As we saw previously, we can view a probabilistic program as describing a sampling procedure encoding how one believes observed data is generated. The flexibility of sampling has inspired embeddings of probabilistic primitives in full-fledged programming languages, including Scheme and Scala (*e.g.*, [39, 80]). However, languages based on sampling often avoid the basic question of what it means to sample from a continuous distribution. For instance, these languages either have semantics given by an implementation in a host-language (*e.g.*, [39]), or use an abstract machine that assumes reals and primitive continuous distributions [75].

Another approach is to ground the semantics of probabilistic programs in measure theory, which is the foundation of probability theory (*e.g.*, [17, 97]). Measure theory provides a rigorous definition of conditioning, and a uniform treatment of discrete and continuous distributions. For instance, measure-theoretic probability makes sense of why the probability of obtaining any particular sample from an (absolutely) continuous distribution is zero, as well what it means to observe a probability zero event. These situations are encountered frequently in practice, as many models incorporate both continuous distributions and observation of real-valued data. Consequently, measure-theoretic semantics have been proposed as a generalization of sampling semantics (*e.g.*, [17]). However, this approach also has its drawbacks. First, measure theory has been developed without programming language constructs in mind (*e.g.*, recursion and higher-order functions), unlike standard denotational semantics.

Hence, languages based on measure theory often omit language features (*e.g.*, [17]) or develop new meta-theory (*e.g.*, [97]). Second, measure-theoretic semantics must also develop a corresponding operational theory. For instance, Toronto *et al.* [97] give denotational semantics and later show how to implement a sound approximation of their ideal semantics because it is not directly implementable.

I propose an approach to giving semantics to probabilistic programs motivated by *Type-2 computable distributions*, which admit *Type-2 computable* sampling procedures.[1] Computable distributions have been defined and studied in the context of Type-2 (Turing) machines and algorithmic randomness (*e.g.*, [32, 33, 101]) as well as domain theory (*e.g.* [26, 27]). Their implications for probabilistic programs have also been hinted at in the literature (*e.g.*, [5, 31]). Hence, I will recast these ideas in the context of high-level probabilistic languages for Bayesian inference.

1.2.2 Compilation

A recurring theme in the implementation of a PPL is how to address the analytic intractability of inference. We list several challenges of probabilistic inference and how PPLs have addressed them.

- As inference is intractable in general, it is unlikely that a blackbox approach to inference will work well for all probabilistic models of interest. Consequently, we would like a way to derive model-specific facts useful for constructing inference algorithms. For example, the Bugs [94] system automatically detects *conjugacy relations* in the probabilistic model and uses this information to construct efficient Gibbs MCMC samplers. As another example, Stan [19] implements automatic differentiation (AD) [9], which enables the system to leverage gradient information to perform HMC sampling. Stan

[1] Because we will use the phrase "Type-2 computable" frequently, we will sometimes abbreviate it to just "computable" when it is clear from context that we are referring to Type-2 computability.

also leverages gradient information to perform Automatic Differential Variational Inference [55] (AVDI), which is a non-sampling approach to inference.

- There are many kinds of inference algorithms. Ideally, a tool should support as many as possible, given that these algorithms may exhibit different computational and statistical behaviors depending on the model to which it is applied. More generally, end users may want more fine-grained control over how to perform inference. For example, Blaise [16] provides a domain-specific graphical language for expressing models and inference algorithms. Other systems such as Edward [98] and Venture [61] explore other designs for providing fine-grained control of inference.
- Inference algorithms can be computationally expensive (*e.g.*, MCMC algorithms [18]). To improve the computational efficiency, we may want to leverage parallelism such as in Augur [99] and other systems such as Anglican [105]. Moreover, the formal representation of a model opens the possibility for compiler analysis to improve the efficiency of the generated code as done by Swift [106].

In this dissertation, I present the design and implementation of domain-specific PPL called AugurV2 that answers *posterior sampling queries* for Bayesian networks expressed in a probabilistic modeling language similar in expressive power to Bugs [94] or Stan [19] using *Markov Chain Monte Carlo* (MCMC) sampling techniques. More concretely, we can view AugurV2 as a tool for constructing (*model*, *query*, *inference*) tuples, where (1) the *model* is expressed in a domain-specific modeling language (as compared to modeling languages embedded in general-purpose languages [105, 39, 106]), (2) the *query* is for posterior samples given observed data, and (3) the *inference* is automatically derived and is restricted to a family of algorithms based on MCMC sampling. The compiler's architecture is based off of a traditional compiler design. In particular, I define a sequence of intermediate languages that structure the compilation process, which include separate languages for representing models and inference. Then, I provide the corresponding phases of compilation:

- The Frontend translates the modeling language into an intermediate representation of a model in terms of its *density factorization*.

- The Middle-end translates an intermediate representation of a model into a high-level, executable MCMC sampling algorithm.
- The Backend translates a high-level, executable inference algorithm into an inference algorithm for execution on the CPU or GPU.

Hence, I consider the *holistic* design and implementation of a resticted PPL that follows the usual approach to compiler design to manage its complexity. I do not address the compilation of more expressive PPLs, but hope that the ideas presented here can be a starting point for compiling more expressive PPLs.

1.3 Overview

In this section, I provide an overview of the contents of the dissertation, as well as collaborators and relevant publications that the dissertation draws on. The first part of the dissertation addresses the modeling aspect of probabilistic programming. The second part of the dissertation addresses the compilation of inference algorithms.

Chapter 2: Background Chapter 2 provides background on probability theory, (Type-2) computable distributions, and probabilistic inference. The emphasis will be placed on relating the three concepts with one another, as opposed to presenting them in isolation. Ideally, a PPS draws on concepts from probability theory to define models, (Type-2) computability to guide its (theoretical) implementability, and probabilistic inference to inform an efficient and correct implementation. The background is intended to provide context for definitions that will be used throughout the dissertation, rather than being a comprehensive review.

However, Chapter 2 does not contain all the background pertinent to the dissertation. In particular, Chapter 3 contains additional background relevant to giving semantics to lan-

guages, and Chapter 5 contains additional background relevant to the implementation of PPLs.

I should point out that in the discussion of Type-2 computability, the connection to realizability [14, 57, 100] is missing. To the best of my knowledge, these fields have largely proceeded independently of each other, although connections between realizability and Type-2 computability have been drawn from the realizability perspective [14]. I would like to thank Bas Spitters and Lars Birkedal for introducing me to these ideas. An informal discussion of the connections will be given in Chapter 4 after I have given a semantics to λ_{CD}.

Chapter 3: Towards Formal Languages for Probability Chapter 3 further motivates the technical issues with giving semantics to probabilistic programs. The conventional wisdom is to use measure-theoretic structure to give semantics, without regards to computability. I will further motivate the reason to consider (Type-2) computable distributions, as opposed to more generally measures. Chapter 3 also contains background on topological domains [13], semantic objects that possess both order-theoretic and (Type-2) computability structure.

Chapter 4: An Application of Computable Distributions to the Semantics of Probabilistic Programming Languages Chapter 4 contains two semantics for a PCF-like language extended with continuous distributions called λ_{CD}. The first semantics uses topological domains and is new to the dissertation. The second semantics uses standard complete partial orders (CPOs) and is based on prior published work [47], although the presentation here contains corrections for a gap discovered in that work. In particular, I would like to thank Mitch Wand for finding an error in an earlier proof of why an environment lookup func-

tion in that semantics was continuous. Appendix A provides an ad-hoc fix for this gap, which (perhaps surprisingly) relies on Type-2 computability. This motivated the definition of a second semantics based on topological domains in this dissertation. The chapter will also include an informal discussion of the connection to realizability.

Chapter 5: Towards Compilation of Probabilistic Modeling Languages Chapter 5 introduces background on MCMC sampling and more concretely explores the issues with constructing MCMC sampling algorithms.

Chapter 6: Flexible Compilation for Probabilistic Programs Chapter 6 describes a compiler design for AugurV2, a PPL that expresses fixed-structure Bayesian networks. The system described in this chapter is the second iteration of a previous system called Augur [99], which was done in collaboration with Jean-Baptiste Tristan, Joseph Tassarotti, Adam Pocock, Stephen Greene, and Guy Steele. The previous compiler generated homogenous MCMC algorithms for execution on the GPU to explore the feasibility of parallelizing inference algorithms. The key ideas developed in this system include (1) compiling at runtime so that we could use dataset sizes to determine a parallelization strategy and (2) representing conditional independence relationships symbolically. The second iteration of the system builds upon these ideas and supports composable MCMC algorithms for CPU and GPU execution. In particular, this caused a complete rearchitecture of the compiler in the form of additional intermediate languages, transformations, and runtime libraries to support more MCMC algorithms and their composition. Parts of this work will appear at Programming Language Design and Implementation, 2017.

Chapter 7: The AugurV2 Language Chapter 7 introduces the AugurV2 language and its semantics via compilation to λ_{CD}. In particular, we will see that all AugurV2 programs admit a density factorization, which justifies its implementation via MCMC sampling. We will also see how (Type-2) computability informs the design of AugurV2's conditioning primitive. Notably, conditioning is not (Type-2) computable in general [5].

Chapter 8: Conclusion Chapter 8 concludes the dissertation and suggest avenues for future work.

Remark (Contributions). In the first part of the dissertation, my contribution is simply one of synthesizing known results on (Type-2) computable distributions and existing theory in the context of a high-level probabilistic modeling language. In the second part of the dissertation, I present the design and implementation of a domain-specific PPL called AugurV2. In many ways, this also simply synthesizes many ideas on the implementation of PPLs found in the literature with one another, ideas from traditional compiler design, and uses the semantics from the first part to guide its design. Hence, this dissertation can be seen as an attempt to bring together known ideas from a broad range of fields, including in machine learning, computability, and language semantics, in a (hopefully) coherent fashion. Nevertheless, I would like to emphasize that it took a large amount of effort on my part to synthesize these ideas and try to connect them. Indeed, these fields are largely presented independently of one another.

2

Background

In this chapter, we provide background on (1) measure-theoretic probability (*e.g.*, see Billingsley [15]), (2) Type-2 computability (*e.g.*, see Pauly [76]) and computable distributions (*e.g.*, see Schröder [88] and Hoyrup [46]), and (3) Bayesian modeling and inference (*e.g.*, see Bayesian Data Analysis [35]). Items (1) and (2) will be most relevant for the first part of the dissertation. Items (1) and (3) will be most relevant for the second part of the dissertation. The background is intended to provide context for definitions that will be used throughout the dissertation, rather than being a comprehensive review. For a pedagogical introduction to the

material, we refer the reader to the references provided above.

2.1 Probability

Formulating probability theory in terms of measure theory enables (1) a uniform treatment of discrete and continuous distributions and (2) a rigorous definition of conditioning—in particular, conditioning on probability zero events. Due to the robustness of this formulation, measure-theoretic probability is often taken as the foundation of modern probability theory. In this section, we will concentrate our review on distributions and random variables, and delay conditioning until Section 2.3.2. Before proceeding directly to the measure-theoretic formulation, we begin with discrete and continuous distributions as presented in an applied setting.

2.1.1 Discrete and Continuous Distributions

A *discrete distribution* assigns probability to a discrete (*i.e.*, at most countable) set. It is characterized by a *probability mass function* (pmf), which maps a value in its domain X to a probability in the interval $[0, 1]$, such that the probabilities sum to 1. More concretely, a pmf $p : X \to [0, 1]$ satisfies

$$\sum_{x \in X} p(x) = 1 \,,$$

where $|X|$ is at most countable. For example, a fair coin flip resulting in 0 (for heads) or 1 (for tails) can be modeled as a *Bernoulli distribution* with pmf $p : \{0, 1\} \to \mathbb{R}$ such that $p(0) = 0.5$ and $p(1) = 0.5$.

A *continuous distribution* assigns probability to the reals $\mathbb{R}$. It is characterized by a *cumulative distribution function* (cdf), which maps a value $c \in \mathbb{R}$ to the probability of the interval

$(-\infty, c]$. For example, a uniform distribution on the interval (a, b), written $\mathcal{U}(a, b)$, has cdf

$$F(c) = \begin{cases} 0 & \text{when } c < a \\ \frac{x-a}{b-a} & \text{when } a \leq c < b \\ 1 & \text{when } b \leq c\,. \end{cases}$$

Note that a cdf gives the probability of an interval and not a single point. A *probability density function* (pdf) is the analog of a pmf in the continuous setting. When a pdf $f : \mathbb{R} \to \mathbb{R}$ exists, it is related to the cdf $F : \mathbb{R} \to [0, 1]$ as

$$F(c) = \int_{-\infty}^{c} f(x)\, dx\,.$$

For example, the uniform distribution $\mathcal{U}(a, b)$ has pdf

$$p(x) = \begin{cases} \frac{1}{b-a} & \text{when } a \leq x \leq b \\ 0 & \text{otherwise}\,. \end{cases}$$

As a matter of terminology, it is common in practice to refer to only those distributions with pdfs as continuous distributions. We will use a measure-theoretic distinction and call these *absolutely continuous distributions*. Although a pdf is an analog of a pmf, note that a pdf is not a probability. For instance, the pdf $p_{\mathcal{U}(0,1/2)}$ of the uniform distribution $\mathcal{U}(0, 1/2)$ has $p_{\mathcal{U}(0,1/2)}(1/4) = 2$, which is greater than 1 so it is not a probability. Indeed, the probability of the event $\{x\}$ for any point x according to an absolutely continuous distribution is zero.

2.1.2 Measure-theoretic Probability

Random variables are the basic object of study in measure-theoretic probability. Informally, we can think of a random variable as a *variable* that takes on an uncertain value according to a *distribution*. It is a *variable* in the sense that it obeys a substitution principle, albeit a non-standard one compared to an ordinary variable (which can be sustituted in any context). The underlying *distribution* is formulated as a *measure*. Importantly, both discrete and continuous distributions can be formulated as measures, which informally, assign probabilities to special subsets (called *events*) of a space. The base space that provides the structure required for probability is called a *measurable space*. Later in Section 2.2, we will see that computable distributions combine measure-theoretic structure with topological structure, which provides a notion of approximation suitable for modeling computability.

Once we have a notion of random variable, we can answer questions about collections of random variables (known as a *stochastic process*). For example, we may be interested in the *joint distribution* of some (finite) subcollection of random variables. As another example, for a single random variable, we may be interested in computing its *expectation*, *i.e.*, its average value. Now that we have an idea of how the overall theory is setup, we can review the definitions that we will need from measure-theoretic probability formally.

Measures The pair $(\Omega, \mathcal{F})$ is a *measurable space*, where Ω is an underlying set and $\mathcal{F}$ is a collection of subsets of Ω called a σ-algebra. A σ-algebra is closed under countable unions and complements (and hence also countable intersections). Then, a *measure* $\mu : \mathcal{F} \to [0, \infty]$ on Ω maps a *measurable set* (*i.e.*, any set in $\mathcal{F}$) to a non-negative real number, such that $\mu(\emptyset) = 0$ and μ is countably additive (*i.e.*, $\mu(\bigcup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} \mu(A_i)$ for any collection $\{A_i\}_{i \in \mathbb{N}}$ of dis-

joint sets). In the probabilistic setting, a measurable set is also called an *event*. A *probability measure* or *probability distribution* is a measure μ such that the mass of the entire space is 1 (*i.e.*, $\mu(\Omega) = 1$). Thus, a distribution maps events to probabilities.

A discrete distribution is defined on a discrete set Ω with σ-algebra 2^Ω (*i.e.*, the collection of all subsets of Ω or the powerset of Ω). For example, the Bernoulli distribution can be encoded as a measure μ with $\mu\{0\} = 0.5$ and $\mu\{1\} = 0.5$, where the probability of the other sets is determined by the measure axioms. A continuous distribution is defined on $\mathbb{R}$ with (Borel) σ-algebra $\mathcal{F} = \sigma(\{(a, b) \mid a, b \in \mathbb{R}\})$, where the operation σ closes the collection of open intervals $\{(a, b) \mid a, b \in \mathbb{R}\}$ under the σ-algebra operations. For example, the uniform distribution $\mathcal{U}(0, 1)$ has measure $\mu(a, b) = b - a$ when $(a, b) \subseteq (0, 1)$. On the real line, it is common to use *Lebesgue measure*, *i.e.*, the completion of the Borel σ-algebra.[1]

RANDOM VARIABLES Let $(\Omega, \mathcal{F}_\Omega)$ and $(X, \mathcal{F}_X)$ be measurable spaces. Then, a function $f : \Omega \to X$ is *measurable* if $f^{-1}(B)$ is measurable (*i.e.*, $f^{-1} \in \mathcal{F}_\Omega$) for all measurable B (*i.e.*, $B \in \mathcal{F}_X$). Let $(\Omega, \mathcal{F})$ be a measurable space and μ be a probability measure on $\mathcal{F}$. Then, the triple $(\Omega, \mathcal{F}, \mu)$ is a *probability space*. Let $(\Omega, \mathcal{F}_\Omega, \mu)$ be a probability space and $(X, \mathcal{F}_X)$ be a measurable space. Then, a *X-valued random variable* $\mathbf{X}$ is a measurable function $\mathbf{X} : \Omega \to X$. As convention, we will used bold-faced, capital letters such as $\mathbf{X}$ to name random variables. Let $(\Omega, \mathcal{F}, \mu)$ be a probability space. The *distribution of* $\mathbf{X}$, written $\mathbb{P}(\mathbf{X} \in \cdot)$ is the *pushforward* of μ along $\mathbf{X}$, *i.e.*, $\mathbb{P}(\mathbf{X} \in \cdot) = \mu \circ \mathbf{X}^{-1}$.

Remark (Interpretation of random variables). Let $(\Omega, \mathcal{F}, \mu)$ be a probability space, $(X, \mathcal{F}_X)$ be a measurable space, and $\mathbf{X}$ be a X-valued random variable. As terminology, the domain of

[1] As a reminder, a measure is *complete* if every subset of a measure zero set is measurable.

a random variable Ω is called the *sampling space*. The sampling space Ω can be interpreted as containing all possible quantities that determine the value of $\mathbf{X}$. For example, suppose $X \cong 2 \cong \{0, 1\}$ is the space of coin flips. Then, we can imagine Ω to be a subset of $\mathbb{R}^d$ that contains physical quantities that determine the outcome of a coin flip such as the angle and force with which it was flipped. The random variable $\mathbf{X}$ could then be interpreted as a physics simulation that converts the input angle and force into the result of a coin flip. Observe that the underlying distribution μ on Ω makes the result of the coin flip random. Notably, in probability theory, one never explicitly describes what the sample space is—a sample space is simply assumed to exist. As we will see later, this will not be the case for statisticians who are interested in *modeling* the sampling space so they can make inferences about the underlying distribution given observed data (Section 2.3).

EXPECTATION The expectation of a real-valued random variable is intuitively its average value. It is defined by a Lebesgue integral. Let $\mathbf{X}$ be a real-valued random variable with distribution μ. Then, its *expectation* is the integral of the identity function with respect to its distribution, written

$$\mathbb{E}[\mathbf{X}] = \int_X x \, d\mu \, .$$

The above is perhaps better written as

$$\mathbb{E}[\mathbf{X}] = \int_X \bar{x} \mapsto \bar{x} \, d\mu$$

to indicate that we integrate the (measurable) function $\bar{x} \mapsto \bar{x}$ with respect to the distribution μ.

JOINT DISTRIBUTIONS Let $(X, \mathcal{F}_X)$ and $(Y, \mathcal{F}_Y)$ be measurable spaces. Let $\mathbf{X}$ be a X-valued random variable and $\mathbf{Y}$ be a Y-valued random variable. Then, the *random vector* $(\mathbf{X}, \mathbf{Y})$ has a *joint distribution*, written $\mathbb{P}(\mathbf{X} \in \cdot, \mathbf{Y} \in \cdot)$. We say that $\mathbf{X}$ is independent of $\mathbf{Y}$ if its joint distribution factors, *i.e.*,

$$\mathbb{P}(\mathbf{X} \in A, \mathbf{Y} \in B) = \mathbb{P}(\mathbf{X} \in A)\mathbb{P}(\mathbf{Y} \in B)\,.$$

DENSITIES AND KERNELS Let $(X, \mathcal{F}_X)$ be a measurable space. Let μ and ν be measures on X. We say μ is *absolutely continuous* with respect to ν if $\mu(B) = 0$ whenever $\nu(B) = 0$ for any measurable B. We say that measurable $f : X \to \mathbb{R}$ is a *density* of μ if

$$\mu(B) = \int f \, d\nu$$

for μ absolutely continuous with respect to ν.[2] A common case is when X is the real line and ν is the Lebesuge measure.

Let $(X, \mathcal{F}_X)$ and $(Y, \mathcal{F}_Y)$ be measurable spaces. Let $\kappa : X \times \mathcal{F}_Y \to [0, 1]$. Then, We say that κ is a *probability kernel* if (1) κ_x is a probability measure for every $x \in X$, where $\kappa_x(B) \triangleq \kappa(x, B)$, and (2) κ_B is measurable for every $B \in \mathcal{F}_Y$, where $\kappa_B(x) \triangleq \kappa(x, B)$.

Importantly, a probability kernel generalizes the notion of a conditional density. To see this, let $\mathbf{X}$ be a X-valued random variable with distribution $\mu_{\mathbf{X}}$ and $\mathbf{Y}$ be a Y-valued random variable with distribution $\mu_{\mathbf{Y}}$. Moreover, suppose that the random vector $(\mathbf{X}, \mathbf{Y})$ has joint

[2] The Radon-Nikodym theorem gives conditions under which a density exists.

density $f_{\mathbf{X},\mathbf{Y}}$, and marginal densities $f_{\mathbf{X}}$ and $f_{\mathbf{Y}}$. Then, a *conditional density* is defined as

$$f_{\mathbf{Y}|\mathbf{X}=x}(y) = \frac{f_{\mathbf{X},\mathbf{Y}}(x,y)}{f_{\mathbf{X}}(x)}$$

such that

$$\mu_{\mathbf{Y}}(B) = \int_B f_{\mathbf{Y}|\mathbf{X}=x}(y)\, f_{\mathbf{X}}(x)\, d\mu_{\mathbf{X}}\,.$$

In particular,

$$\kappa_x(B) = \int_B f_{\mathbf{Y}|\mathbf{X}=x}\, d\mu_{\mathbf{Y}}\,,$$

where $\kappa : X \times \mathcal{F}_Y \to [0,1]$ is a probability kernel.

2.2 Computability

In this section, we provide background on Type-2 computability, also known as Type-Two Theory of Effectivity (TTE) [102]. Our goal is to review (Type-2) computable distributions [46, 33, 101], which we will use to interpret distributions in a probabilistic program. Our review of Type-2 computability will proceed in several steps, each successively more abstract.

1. We review the Turing machine model that underlies Type-2 computability and contrast it with ordinary computability theory (*i.e.*, Type-1 computability) (see Section 2.2.1).
2. We review computable metric spaces [46], which can be used to formulate computable reals (see Section 2.2.2). Importantly, (Borel) distributions can be formulated as both points of the appropriate computable metric space as well as computable sampling functions.
3. We review represented spaces [76], the most general space that has a computability theory that is derived from Turing machines (see Section 2.2.3). Notably, represented spaces form a Cartesian closed category, which will be useful when we transition to giving semantics to probabilistic programs.

2.2.1 Machine Model

Type-2 computability theory is grounded on the *same* Turing machine model that forms the foundation of standard computability theory. That is, a *Type-2 machine* is a finitely-specified machine that reads/writes symbols from a finite alphabet Σ to/from (infinite) tapes. (For the formal definition of a Turing machine, we refer the reader to standard texts such as Sipser's [89].) Hence, we will simply refer to Type-2 machines as Turing machines. The primary difference as compared to standard computability theory is the condition under which a Turing machine is said to compute an answer.

In the standard setting, a Turing machine computes on *finite-length words*, and hence, can halt with the answer on the output tape. As a reminder, a (partial) word function $f : \Sigma^* \rightharpoonup \Sigma^*$ is considered *computable* if there is a Turing machine with tape alphabet Σ that halts with $f(w)$ written on the output tape for every $w \in \mathrm{dom}(f)$, where $\Sigma^* \triangleq \{w_0 \dots w_n \mid n \in \mathbb{N}\}$ is the set of finite-length words comprised of letters from Σ. Note that any set X that can be put in surjection with Σ^* can then inherit the corresponding computability theory. For example, as Σ^* can be put in bijection with $\mathbb{N}$, we can derive a notion of a computable function $f : \mathbb{N} \rightharpoonup \mathbb{N}$. Thus, standard computability theory concerns discrete values.

In the Type-2 setting, a Turing machine computes on *streams*, and hence, has a productivity condition imposed on the symbols written to the output tape. Let $\Sigma^\omega \triangleq \{w_0 w_1 \dots \mid w_n \in \Sigma, n \in \mathbb{N}\}$ be the set of streams comprised of letters from Σ. Moreover, we write $w \sqsubset p$ to indicate that the word w is a prefix of the stream p and $p_{>|w|}$ to indicate the rest of the stream without the prefix w. Then, a (partial) stream function $f : \Sigma^\omega \rightharpoonup \Sigma^\omega$ is considered *computable* if there is a Turing machine with tape alphabet Σ that for $p \in \mathrm{dom}(f)$, eventually outputs $f(p)$ such that for any $w_1 \sqsubset w_2 \sqsubset p$, the output tape satisfies $f(w_1 p_{>|w_1|}) \sqsubset f(w_2 p_{>|w_2|}) \sqsubset$

$f(p)$. This notion of computability enforces a *finite prefix property*: finite prefixes of the input determine finite prefixes of the output. Hence, Type-2 computability concerns computing arbitrarily-accurate approximations to continuous values. In particular, the emphasis on approximation means that the structure of topological spaces will be useful for formulating these ideas. For background on topological spaces, we refer the reader to standard texts [67].

2.2.2 Computable Metric Spaces

To build towards the definition of computable distribution, we begin by considering computable reals. Intuitively, a real is computable if a Turing machine can *enumerate* its binary expansion. More formally, a real $x \in \mathbb{R}$ is *computable* if we can enumerate a fast Cauchy sequence of rationals that converges to x. Recall that a sequence $(q_n)_{n\in\mathbb{N}}$ is *Cauchy* if for every $\epsilon > 0$, there is an N such that $d(q_n, q_m) < \epsilon$ for every $n, m > N$. Thus, the elements of a Cauchy sequence become closer and closer to one another as we traverse the sequence. When $d(q_n, q_{n+1}) < 2^{-n}$ for all n, we call $(q_n)_{n\in\mathbb{N}}$ a *fast Cauchy sequence*. Hence, the representation of a computable real as a fast Cauchy sequence evokes the idea of enumerating its binary expansion.

As an example of a computable real, consider π and one possible series expansion of it below [8].

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left[\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right]$$

An algorithm can use the above series expansion and a rate of convergence to obtain a fast Cauchy sequence (*e.g.*, the BPP algorithm [8]).

A function $f : \mathbb{R} \to \mathbb{R}$ is computable if given a (fast Cauchy) sequence converging to $x \in \mathbb{R}$, there is an algorithm that outputs a (fast Cauchy) sequence converging to $f(x)$. For example,

the function $+_\pi : \mathbb{R} \to \mathbb{R}$ is computable because an algorithm can obtain a (fast Cauchy) output sequence by adding the (fast Cauchy) input sequence element-wise to a (fast Cauchy) sequence of π.

A topological space with a notion of distance is a *metric space*. As a reminder, a *metric space* (X, d) is a set X equipped with a *metric* $d : X \times X \to \mathbb{R}$. A metric induces a collection of sets called *(open) balls*, where a ball centered at $c \in X$ with radius $r \in R$ is the set of points within r of c, *i.e.*, $B(c, r) = \{x \in X \mid d(c, x) < r\}$. The topology $\mathcal{O}(X)$ associated with a metric space X is the one induced by the collection of balls. Hence, the open balls of a metric space provide a notion of distance in addition to providing a notion of approximation.

Example 1.

- $(\mathbb{N}, d_{\text{Discrete}})$ endows the naturals $\mathbb{N}$ with the discrete topology (*i.e.*, $\mathcal{O}(\mathbb{N}) = 2^{\mathbb{N}}$), where d_{Discrete} is the discrete metric (*i.e.*, $d(n, m) = 0$ if $n = m$ and $d(n, m) = 1$ otherwise for $n, m \in \mathbb{N}$).
- $(\mathbb{R}, d_{\text{Euclid}})$ endows the reals $\mathbb{R}$ with the familiar Euclidean topology, where d_{Euclid} is the standard Euclidean metric.
- $(2^\omega, d_{\text{Cantor}})$ endows the set of bit-streams 2^ω with the Cantor topology, where $d_{\text{Cantor}}(x, y) = 1/2^n$ where $n = \min\{i \mid x_i \neq y_i\}$. One can check that a basic open set of the Cantor topology is of the form $a_1 a_2 \ldots a_n 2^\omega$, where $a_i \in 2$ for $1 \leq i \leq n$. That is, basic open sets of Cantor space fix finite-prefixes.

Computable metric spaces Intuitively, a computable metric space imposes additional conditions on a metric space so that a Turing machine can enumerate successively more accurate approximations (according to the metric) to a point in the metric space. We say S is *dense* in X if for every $x \in X$, there is a sequence $(s_n)_{n \in \mathbb{N}}$ that converges to x, where $s_n \in S$ for every n. We say that (X, d) is *complete* if every Cauchy sequence comprised of

elements from X also converges to a point in X. Then, a *computable metric space* [46, Def. 2.4.1] is a tuple (X, d, S) such that (1) (X, d) is a complete metric space, (2) S is a countable, enumerable, and dense subset, and (3) $d(s_i, s_j)$ is computable for $s_i, s_j \in S$. For example, $(\mathbb{R}, d_{\text{Euclid}}, \mathbb{Q})$ is a computable metric space for the reals where we use the rationals $\mathbb{Q}$ as the approximating elements. Note that we can equivalently use dyadic rationals as the approximating elements instead of $\mathbb{Q}$.

Computable distributions A computable distribution over the computable metric space (X, d, S) can be formulated as a computable point of the computable metric space $(\mathcal{M}(X), d_\rho, \mathcal{D}(S))$, where $\mathcal{M}(X)$ is the set of Borel probability measures on a computable metric space (X, d, S), d_ρ is the Prokhorov metric (see [46, Defn. 4.1.1.]), and $\mathcal{D}(S)$ is the class of distributions with finite support at ideal points S and rational masses (see [46, Prop. 4.1.1]). The Prokhorov metric is defined as:

$$d_\rho(\mu, \nu) \triangleq \inf\{\epsilon > 0 \mid \mu(A) \leq \nu(A^\epsilon) + \epsilon \text{ for every Borel } A\} ,$$

where $A^\epsilon = \{x \mid d(x, A) < \epsilon\}$. One can check that the sequence below converges (with respect to the Prokhorov metric) to the (standard) uniform distribution $\mathcal{U}(0, 1)$.

$$\left\{0 \mapsto \frac{1}{2}, \frac{1}{2} \mapsto \frac{1}{2}\right\}, \left\{0 \mapsto \frac{1}{4}, \frac{1}{4} \mapsto \frac{1}{4}, \frac{2}{4} \mapsto \frac{1}{4}, \frac{3}{4} \mapsto \frac{1}{4}\right\}, \ldots,$$

Thus, a uniform distribution can be seen as the limit of a sequence of increasingly finer discrete, uniform distributions. Although the idea of a computable distribution as a computable point is fairly intuitive for the standard uniform distribution, it may be less insightful for

more complicated distributions.

Alternatively, we can think of a computable distribution on a computable metric space (X, d, S) in terms of sampling, *i.e.*, as a (Type-2) computable function $2^\omega \to X$. To make this more concrete, we sketch an algorithm that samples from the standard uniform. The idea is to generate a value that can be queried for more precision instead of a sample x in its entirety. Thus, a sampling algorithm will interleave flipping coins with outputting an element to the desired precision, such that the sequence of outputs $(s_n)_{n\in\mathbb{N}}$ converges to a sample.

For instance, one binary digit of precision for a standard uniform corresponds to obtaining the point $1/2$ because it is within $1/2$ of any point in the unit interval. Demanding another digit of precision produces either $1/4$ or $3/4$ according to the result of a fair coin flip. This is encoded below using the function `bisect`, which recursively bisects an interval n times, starting with $(0, 1)$, using the random bit-stream u to select which interval to recurse on.

$$\text{uniform} : (\texttt{Nat} \to \texttt{Bool}) \to (\texttt{Nat} \to \texttt{Rat})$$

$$\text{uniform} \triangleq \lambda u.\ \lambda n.\ \texttt{bisect}\ u\ 0\ 1\ n$$

In the limit, we obtain a single point corresponding to the sample.

The sampling view is (computably) equivalent to the definition of a computable distribution in terms of computable metric spaces. To state the equivalence, we need a few definitions. A *computable probability space* [46, Def. 5.0.1] (X, μ) is a pair where X is a computable metric space and μ is a computable distribution. We call a distribution μ on X *samplable* if there is a computable function $s : (2^\omega, \mu_{\text{iid}}) \to (X, \mu)$ such that s is computable on $\text{dom}(s)$ of full-measure and is measure-preserving.

Proposition 2.2.1 (Computable iff samplable, see [30, Lem. 2 and 3])**.** *A distribution $\mu \in \mathcal{M}(X)$ on computable metric space $(X, d, \mathcal{S})$ is computable iff it is samplable.*

Hence, proposition 2.2.1 gives the computable analog of the probability integral transform and inverse transform from statistics.

Remark (Sampling). Note that the definitions of a sampler and a random variable are similar. The difference between the two is somewhat philosophical—a sampler has an explicitly-specified underlying probability space (*e.g.*, 2^ω), whereas the underlying probability space for a random variable is left unspecified (see Remark 2.1.2).

2.2.3 Represented Spaces

Roughly speaking, a represented space is an abstraction that provides a more compositional approach to Type-2 computability. That is, instead of constructing a Turing machine to witness the continuity/computability of a function, we can instead write the function as a composition of continuous/computable primitives. Additionally, represented spaces and continuous/computable maps between them form a Cartesian closed category of spaces. This will be useful for extending Type-2 computability to the semantics of higher-order programming languages. We now review represented spaces following Pauly's introduction [76].

A *represented space* (X, δ_X) is a pair of a set X with a partial surjective function $\delta_X : 2^\omega \rightharpoonup X$ called a *representation*. We call $p \in 2^\omega$ a *name* of x when $\delta_X(p) = x$. A *realizer* for a function $f : (X, \delta_X) \to (Y, \delta_Y)$ is a (partial) function $F : 2^\omega \rightharpoonup 2^\omega$ such that $\delta_Y(F(p)) = f(\delta_X(p))$ for $p \in \mathrm{dom}(f \circ \delta_X)$. A function $f : X \to Y$ between represented spaces is called *computable* if it has a computable realizer. It is called *continuous* if it has a continuous realizer (with respect to the Cantor topology). Unfolding the definition of continuity of a (partial) function

$f : 2^\omega \rightharpoonup 2^\omega$ on Cantor space shows that it encodes the finite prefix property. As a reminder, this means that a machine can compute $f(p)$ to arbitrary precision after consuming a finite amount of bits of p in finite time when f is continuous.

Let (X, δ_X) and (Y, δ_Y) be represented spaces. The notation $\langle \cdot, \cdot \rangle : 2^\omega \times 2^\omega \to 2^\omega$ is a standard tupling function that dovetails the two input bit-streams.

1. The product of represented spaces, written $X \times Y$, is a represented space. It has the representation $\delta_{X \times Y} \langle p, q \rangle = (\delta_X(p), \delta_Y(q))$.
2. The exponential of represented spaces, written $\mathscr{C}(X, Y)$, is a represented space. It has representation $\delta_{X \to Y} \langle 0^n 1 p \rangle = f$ if the n-th Turing machine equipped with oracle p is a realizer of f.

Proposition 2.2.2 ([76, Sec. 3])**.** *The category with represented spaces as objects and continuous/computable maps as morphisms is Cartesian closed.*

The following result connects computability with continuity via oracle Turing machines. As a reminder, an *oracle Turing machine* is a standard Turing machine that additionally has read access to an oracle tape that contains a (possibly non-computable) bit-stream to start.

Proposition 2.2.3 ([76, Sec. 3])**.** *A function $f : (X, \delta_X) \to (Y, \delta_Y)$ is continuous (as a map between represented spaces) iff it is computable by an oracle Turing machine.*

As an example of how to use this proposition, suppose we want to show $\text{partial}_x(f) = y \mapsto f(x, y) : \mathscr{C}(X \times Y, Z) \to \mathscr{C}(Y, Z)$ is continuous for $x \in X$. Let the name of f be $0^n 1 p$, *i.e.*, $\delta_{X \to Y}(0^n 1 p) = f$. Then, we can construct a Turing machine M with oracle $\delta_X(x)$, that runs the n-th Turing machine equipped with oracle p on input $\langle \delta_X(x), q \rangle$, for some δ_Y name q on M's input tape. Hence, we have shown partial_x is computable relative to some oracle, so the map is continuous. Below, we give a few examples of represented spaces.

- $(\mathbb{S}, \delta_{\mathbb{S}})$ is a represented space. It has underlying set $\mathbb{S} = \{\bot, \top\}$ and representation $\delta_{\mathbb{S}}(\bot) = 0^\omega$ and $\delta_{\mathbb{S}}(\top) = p$ for $p \neq 0^\omega$. In particular, this encodes the notion of semi-decidability—a Turing machine semi-decides that a proposition holds (encoded as $\top$) only if it eventually outputs a non-zero bit. The space $\mathbb{S}$ is known as *Sierpinski* space.
- Let (X, d, S) be a computable metric space. Then, $(X, \delta_{\text{Metric}})$ is a represented space with representation δ_{Metric} that uses fast Cauchy sequences as names. That is, $(\delta_{\mathbb{Q}}(w_n))_{n\in\mathbb{N}} \to \delta_{\text{Metric}}(p)$ when $\delta(p) = \langle\, w_1, w_2, \dots \,\rangle$. As a special case, $(\mathbb{R}, \delta_{\mathbb{R}})$ is a represented space, where δ_R is a representation that uses fast Cauchy sequences of rationals as names.

We can associate a collection of open subsets $\mathcal{O}(X)$ to every represented space (X, δ) as $\mathcal{O}(X) \triangleq \{f^{-1}\{\top\} \mid f \in \mathscr{C}(X, \mathbb{S})\}$. In particular, this shows that testing if a point is contained in an open set is semi-decidable. However, we should be careful not to identify the notion of a continuous map between represented spaces with a topological continuous function until we introduce the concept of *admissibility*. Recall that a representation δ_X of X induces the *quotient topology* on X, *i.e.*, $U \subseteq X$ is open if $\delta_X^{-1}(U)$ is open on $\text{dom}(\delta_X)$. For our purposes, we call a represented space *admissible* if every topologically continuous function $f : Y \to X$ (*i.e.*, continuous with respect to quotient topologies) is a continuous map between represented spaces.[3] Note that a continuous map between represented spaces is already topologically continuous because pre-images are computable:

$$f^{-1}(\mathbb{1}_U) = \mathbb{1}_U \circ f\,,$$

where $\mathbb{1}_U \in \mathscr{C}(Y, \mathbb{S}) \cong \mathcal{O}(Y)$ is the characteristic function and $f \in \mathscr{C}(X, Y)$. Thus, the continuous maps between admissible represented spaces coincide with the topologically continuous maps. We repeat the examples of represented spaces above, this time describing the associated

[3] Pauly gives a different definition [76, Defn. 26] and three characterizations [76, Thm. 36]. For our purposes of relating to topological continuity, we take one of the characterizations as the definition instead.

topology.

- The Sierpinski space $(\mathbb{S}, \delta_{\mathbb{S}})$ is an admissible represented space. It has opens $\mathcal{O}(\mathbb{S}) = \{\emptyset, \{\top\}, \{\bot, \top\}\}$.
- $(X, \delta_{\text{Metric}})$ is an admissible represented space, where the topology coincides with the one induced by the basis of open balls.

Remark (Qcb spaces). For our purposes, we will consider only admissible represented spaces. In this case, we can interchangeably use represented space and qcb space [13]. A qcb space X (*i.e.*, T_0 quotient of a countably based space) can be characterized as a space with *admissible representation*. A representation δ_X of X is *admissible* if for any other representation δ'_X of X, the identify function on X has a continuous realizer [13, Defn. 3.10]. If X and Y are qcb spaces, then the topologically continuous functions between them coincide with those that have continuous realizers [13, Cor. 3.13], which gives the same characterization as an admissible represented space.

In the literature, qcb spaces arise in the study of topological domains (*e.g.*, as in Battenfeld [13]), domains that possess a notion of Type-2 computability, whereas represented spaces are used to clarify results on computability. As we will apply results from both the semantics aspect and Type-2 computability aspect, we have introduced both terms.

2.3 Bayesian Modeling and Inference

In this section, we shift gears from reviewing background useful for PPL semantics to the application of PPLs to Bayesian modeling and inference. Hence, the ideas here broadly fall under the realm of statistics. First, we review how random variables can be used to define probabilistic generative models and comment on the common case when these models have density

(Section 2.3.1). Second, we review the measure-theoretic notion of conditioning as conditional expectation and relate it to the common case of conditioning with conditional densities (Section 2.3.2).

2.3.1 Bayesian Modeling

We begin by introducing the overall setup of a parameterized statistical model using measure-theoretic language. Then, we will go over it using applied statistics notation. Throughout this section, let $(\Theta, \mathcal{F}_\Theta)$ and $(Y, \mathcal{F}_Y)$ be measurable spaces. We refer to the measurable space Θ as the *parameter space* and Y as the *observation space*. Now, consider the (measurable) functions (1) $\mathbf{Y} : \Theta \to Y$ and (2) $f : Y \to \Theta$.

Measure-theoretic treatment We can interpret the former as a Y-valued random variable $\mathbf{Y}$, where the parameter space Θ is $\mathbf{Y}$'s sampling space. As a reminder, the sampling space is *implicit* in the context of probability theory. In contrast, it is *explicit* in the context of statistics in order to *model* phenomenon. In particular, we model the randomness in the observation space with a family of *model likelihoods* $\{\ell_\theta \mid \ell_\theta : Y \to \mathbb{R} \text{ integrable}\}_{\theta \in \Theta}$, which for a parameter θ, gives an (integrable) function ℓ_θ that compares the relative likelihoods of different observations.

We can interpret the latter as a *statistical inference procedure* that estimates the $\theta \in \Theta$ that produces the observation $y \in Y$ and quantifies the uncertainty associated with the procedure. These procedures will be the subject of Section 2.3.2. For now, we will focus on how the construction of models are affected by the choice of inference procedure, concentrating on Frequentist and Bayesian inference. As a reminder, this dissertation will focus exclusively on the application of a PPL to Bayesian inference, but it is still worthwhile to also review the

Frequentist setting.

In a Frequentist setting, one treats the observed data as random. That is, one considers the probability space $(Y, \mathcal{F}_Y, \mu_{\theta_0})$, where μ_{θ_0} is the distribution on the observation space Y and θ_0 is a deterministic, but unknown parameter value. Writing this in random variable notation, we have

$$\mathbf{Y} \sim \mu_{\theta_0} .$$

The random variable $\mathbf{Y}$ can be interpreted as a measure-preserving function from the parameter space to the observation space, where we have stated what the distribution is on the observation space. Hence, $\mathbb{P}(\mathbf{Y} \in B) = \mu_{\theta_0}(B)$ for measurable B.

The interpretation of a model in the Frequent setting is that μ_{θ_0} is the true distribution that our observations come from. However, we do not know the value of the true parameter θ_0. The Frequentist methodology provides a set of tools for estimating the parameter, written $\hat{\theta}$, where the uncertainty in the estimate is quantified with respect to the true distribution on data μ_{θ_0}.

In a Bayesian setting, one treats the parameter as random. That is, one consider the probability space $(\Theta, \mathcal{F}_\Theta, \nu)$, where ν is a distribution on the parameter space Θ. Writing this in random variable notation, we have

$$\mathbf{\Theta} \sim \nu$$

$$\mathbf{Y} \mid \mathbf{\Theta} = \theta \sim \mu_\theta ,$$

We will comment on the notation "$\mathbf{Y} \mid \mathbf{\Theta} = \theta$" in Section 2.3.2. For now, we read it as: "Given that $\mathbf{\Theta}$ takes on the value θ, $\mathbf{Y}$ has distribution μ_θ." Intuitively, it defines a family

of random variables. We can interpret the random variable $\mathbf{\Theta}$ as a measure-preserving function from some unspecified sample space to θ, where we have stated the distribution on Θ. Hence, $\mathbb{P}(\mathbf{\Theta} \in B) = \nu(B)$ for measurable B. We can interpret $\{\mathbf{Y} \mid \mathbf{\Theta} = \theta\}_{\theta \in \Theta}$ as a family of measure-preserving functions from the parameter space to Y. Hence, $\mathbb{P}(\mathbf{Y} \mid \mathbf{\Theta} = \theta \in B) = \mu_\theta(B)$ for measurable B.

The interpretation of a model in the Bayesian setting is that ν describes our subjective, prior beliefs about the distribution on the parameter space, the *prior distribution*, that generates an observation. As we will see, the Bayesian paradigm provides a methodology for updating our prior beliefs given observed data to obtain a *posterior distribution*. In particular, we can use the posterior distribution as our new prior distribution when making inferences about additional observations.

Applied treatment In an applied setting, one typically has densities. Hence, we will go over the Bayesian setup again, mostly to explain the notational conventions associated with probabilistic modeling with densities.

We repeat the Bayesian model written with random variable notation below for convenience.

$$\mathbf{\Theta} \sim \nu$$

$$\mathbf{Y} \mid \mathbf{\Theta} = \theta \sim \mu_\theta \, .$$

In the case we have densities, it is typical to write the following model with several abuses of

notation, as below.

$$\theta \sim p(\theta)$$

$$y \mid \theta \sim p(y \mid \theta)\,.$$

First, we have overloaded the variable used to name values a random variable takes on with the random variable itself. For example, we have overloaded θ to be $\boldsymbol{\Theta}$. Second, the notation $p(\theta)$ indicates a density. In this case, p is not being applied to a variable θ. Rather, the variable θ indicates that it is a density on the space Θ. Hence, a perhaps less confusing way of writing it out would be

$$\theta \mapsto p_{\boldsymbol{\Theta}}(\theta)$$

or simply $p_{\boldsymbol{\Theta}}$ to indicate that $p_{\boldsymbol{\Theta}}$ is the density associated with the random variable $\boldsymbol{\Theta}$.

It is also common to express a Bayesian model according to its density factorization. For example, we would write

$$p(y \mid \theta)\, p(\theta)$$

as the model *distribution*. In particular, it is typical to conflate the notion of density and distribution in practice. Finally, we review some terms used to describe probabilistic models in practice.

A *probabilistic generative model* describes a joint distribution $p(\theta, y)$ on the product of the parameter space and observation space. It is possible to generate synthetic data from the description of the model—hence, its name. In contrast, a *discriminative model* describes the conditional distribution $p(y \mid \theta)$. Hence, the model does not fully describe how to create syn-

thetic data. A Bayesian model with distribution $p(\theta, y) = p(y \mid \theta)\, p(\theta)$ can be interpreted as a generative model where we sample from the prior $p(\theta)$ and likelihood $p(y \mid \theta)$ to obtain a synthetic dataset.

Perhaps the most robust feature of Bayesian modeling is the ability to perform *hierarchical modeling*. The idea is that in addition to putting priors on the parameters to our sampling distribution, we can also put priors on our priors. Let Φ and Θ be parameter spaces, and Y be the observation space. Then, a (2-level) hierarchical Bayesian model has the form

$$p(y, \theta, \phi) = p(y \mid \theta)\, p(\theta \mid \phi)\, p(\phi)\,.$$

Thus, we model uncertainty on a parameter θ, in addition to modeling uncertainty on our observation y. Note that the trivial factoring $p(y \mid \phi, \theta)\, p(\phi \mid \theta)\, p(\theta)$ always holds, but is rather uninformative. Much of the power of probabilistic modeling comes from how it (non-trivially) factors.

When all distributions in a Bayesian model with joint distribution $p(y, \theta) = p(y \mid \theta)\, p(\theta)$ have a functional form (*i.e.*, we can write down a formula for its density), it is called a *parametric model*. Depending on the context, a *non-parametric model* has one of two meanings. In statistics, it is common to call a joint distribution whose density does not have known functional form non-parametric (see kernel density estimation [cite]). In machine learning, a non-parametric distribution is used to refer to a distribution on a countable number of parameters (see non-parametric priors such as the Dirichlet process [93]). In either case, inference in the non-parametric setting is more difficult than in the parametric setting.

The number of parameters in a Bayesian model does not need to be known ahead of time. When the number of parameters is fixed and the functional form of the density does not de-

pend on what value the parameters take on, we call it a *fixed-structure model*.

2.3.2 Conditioning and Bayesian Inference

Conditional expectation Let $(\Omega, \mathcal{F}, \mu)$ be a probability space. Let $\mathbf{X}$ be a real-valued random variable on the probability space and $\mathbf{Y}$ be a random variable that is measurable with respect to the sub-σ-algebra $\mathcal{H} \subseteq \mathcal{F}$, where $\mathcal{H} \triangleq \{\mathbf{Y}^{-1}(B) \mid B \text{ Borel}\}$. The *conditional expectation* of $\mathbf{X}$ given $\mathbf{Y}$, written $\mathbb{E}[\mathbf{X} \mid \mathbf{Y}]$, is any $\mathbf{Y}$-measurable function h (*i.e.*, random variable) such that the equation below holds for every $B \in \mathcal{H}$.

$$\int_B h \, d\mu = \int_B \mathbf{X} \, d\mu$$

Any such measurable h is called a *version* of the conditional expectation. Note that the conditional expectation $\mathbb{E}[\mathbf{X} \mid \mathbf{Y}]$ is a family of $\mathbf{Y}$-measurable functions, but that each function in this family will differ only on a set of measure zero.

Regular conditional probability A *regular conditional probability* is a probability kernel κ such that $\mathbb{P}(X \in A, Y \in B) = \int \kappa(x, B) \, \mathbb{P}_X(dx)$, where A and B are measurable. One can show that all random variables on a separable complete metric space admit a regular conditional probability.[4] Recall that a computable metric space is a separable complete metric space, and hence, admit regular conditional probability.

Conditioning with densities Let $\mathbf{X}$ be a X-valued random variable and $\mathbf{Y}$ be a Y-valued random variable. Furthermore, let $\mathbb{P}(\mathbf{X} \in \cdot)$ have density $f_{\mathbf{X}}(\cdot)$ and $f_{\mathbf{Y}|\mathbf{X}=x}(\cdot)$ be a conditional

[4] More generally, see disintegration [20].

density. Then,

$$f_{\mathbf{X}|\mathbf{Y}=y}(x) = \frac{f_{\mathbf{X}}(x)\, f_{\mathbf{Y}|\mathbf{X}=x}(y)}{\int f_{\mathbf{X}}(x)\, f_{\mathbf{Y}|\mathbf{X}=x}(y)\, dx}$$

Remark (Computability of conditioning). In general, conditioning is not (Type-2) computable [5]. Nevertheless, in practice, it is (Type-2) computable. We will discuss this more in depth in Chapter 4 after we have introduced a library for writing computable distributions. In particular, we will be able to write a program using the libary that encodes a counterexample given by Ackerman *et al.* [5] of a computable distribution whose conditional is not computable. We will also explain using the library when conditioning is (Type-2) computable, again following Ackerman *et al.*'s results on the computability of conditioning.

3

Towards Formal Languages for Probability

In this chapter, we will explore some of the difficulties with giving semantics to probabilistic languages extended with continuous distributions. Then, we review some semantic constructs, including topological domains (see Battenfeld *et al.*'s summary [13]), that we will use later to give semantics to a core probabilistic language.

3.1 The Basic Problem

In order to give denotational semantics to a programming language extended with continuous distributions, we should find mathematical structures that support (1) language constructs (*e.g.*, recursion and higher-order functions), (2) familiar reasoning principles about distributions, and (3) an algorithmic sampling interpretation. In this section, we illustrate some of the difficulties using a simple programming language extended with *distributions*.

For now, consider a language with finitely supported discrete distributions with rational probabilities, whose syntax is given below.

$$\tau ::= \texttt{Nat} \mid \texttt{Dist}\ \tau$$

$$e ::= x \mid z \mid \mathit{dist} \mid \oplus \overrightarrow{e} \mid \texttt{return}\ e \mid x \leftarrow e\ ;\ e$$

Types include naturals $\texttt{Nat}$ and distributions $\texttt{Dist}\ \tau$. The language contains variables x, integers z, distributions *dist*, and primitive operations $\oplus\overrightarrow{e}$ on naturals. As others have observed, probabilities form a monad (*e.g.*, see the Giry monad [38]). Hence, the language provides monadic syntax for constructing and manipulating distributions. Informally, $\texttt{return}\ e$ lifts a deterministic computation into the probability monad and monadic bind $x \leftarrow e_1\ ;\ e_2$ can be read as sampling: "draw a sample according to distribution e_1, bind the value to variable x, and continue with distribution e_2."

Following Ramsey *et al.* [83], we can interpret a type $\texttt{Dist}\ \tau$ as a pmf which maps values of type τ (written $\mathcal{V}[\![\tau]\!]$) to a probability in the interval $[0, 1]$

$$\text{(Finite discrete)} \quad \mathcal{V}[\![\texttt{Dist}\ \tau]\!] \triangleq \mathcal{V}[\![\tau]\!] \to [0, 1]\ ,$$

where $|\mathcal{V}[\![\tau]\!]|$ is finite (for simplicity) and $\sum_{x \in \mathcal{V}[\![\tau]\!]} \mathcal{V}[\![\texttt{Dist}\ \tau]\!](x) = 1$. Monadic bind $x \leftarrow e_1\ ;\ e_2$ re-weights the pmf denoted by e_2 by re-weighting according to the pmf denoted by e_1. We write the expression denotation function as $\mathcal{E}[\![\Gamma \vdash e : \tau]\!]\rho \in \mathcal{V}[\![\tau]\!]$ under a well-typed environment ρ with respect to Γ (*i.e.*, $\rho \in \mathcal{V}[\![\Gamma]\!]$).

(Bind) $\mathcal{E}[\![\Gamma \vdash x \leftarrow e_1\ ;\ e_2 : \texttt{Dist}\ \tau_2]\!]\rho \triangleq$

$$v \mapsto \sum_{\bar{x} \in \mathrm{dom}(f_1)} (\mathcal{E}[\![\Gamma, x : \tau_1 \vdash e_2 : \texttt{Dist}\ \tau_2]\!]\rho[x \mapsto \bar{x}])(v) \cdot f_1(\bar{x})$$

where $f_1 = \mathcal{E}[\![\Gamma \vdash e_1 : \texttt{Dist}\ \tau_1]\!]\rho$ is the denotation of e_1.

Because all quantities are discrete and finite, there is an obvious correspondence between the (denotational) semantics and traditional notions of computation. Nevertheless, this framework cannot be directly extended to support recursion (*e.g.*, see Ramsey *et al.* [83]). For example, we can write a program that constructs a distribution on a Boolean-valued stream in a language with recursion and discrete distributions. Notably, the probability of obtaining any stream is 0, so this program does not have a pmf.

Next, we can consider continuous distributions in the context of a probabilistic language by adding the type of reals `Real` to the language and the appropriate distributions. One approach is to interpret $\texttt{Dist}\ \tau$ as a measure-theoretic distribution (*e.g.*, [17]).

(Measure) $\mathcal{V}[\![\texttt{Dist}\ \tau]\!] \triangleq \mathcal{F} \to [0, 1]$ where $\mathcal{F}$ is a σ-algebra on $\mathcal{V}[\![\tau]\!]$

We can use monadic bind again, but now the re-weighting is accomplished by a (Lebesgue)

integral.

(Bind) $\mathcal{E}[\![\Gamma \vdash x \leftarrow e_1 \,;\, e_2 : \texttt{Dist}\ \tau_2]\!]\rho(B) \triangleq$

$$\int (\bar{x} \mapsto \mathcal{E}[\![\Gamma, x : \tau_1 \vdash e_2 : \texttt{Dist}\ \tau_2]\!]\rho[x \mapsto \bar{x}](B))\, d(\mathcal{E}[\![\Gamma \vdash e_1 : \texttt{Dist}\ \tau_1]\!]\rho)$$

where B is measurable.

Unlike the discrete case, the fact that the denotation is well-defined is less obvious. In particular, we need to show that the function

$$\bar{x} \mapsto \mathcal{E}[\![\Gamma, x : \tau_1 \vdash e_2 : \texttt{Dist}\ \tau_2]\!]\rho[x \mapsto \bar{x}](B)$$

is measurable for any measurable B so that we can integrate it. We sketch out a solution that shows the denotation function $\mathcal{E}[\![\Gamma \vdash e : \tau]\!] : \mathcal{V}[\![\Gamma]\!] \to \mathcal{V}[\![\tau]\!]$ is measurable, where $\mathcal{V}[\![\Gamma]\!]$ is the product of all the types it contains.[1]

Sketch 1. We can proceed by induction on the typing derivation. In the bind case, we can conclude from our induction hypothesis that $\mathcal{E}[\![\Gamma, x : \tau_1 \vdash e_2]\!] : \mathcal{V}[\![\Gamma \times \tau_1]\!] \to \mathcal{V}[\![\tau_2]\!]$ is measurable. Indeed, we actually have a *probability kernel* $(\bar{x}, B) \mapsto \mathcal{E}[\![\Gamma, x : \tau_1 \vdash e_2 : \texttt{Dist}\ \tau_2]\!]\rho[x \mapsto \bar{x}](B)$. It is measurable for fixed measurable B and is a measure for fixed $\bar{x}$. Finally, recall that the integral of a probability kernel with respect to a measure yields another measure. Putting all of this together, we conclude that the denotation of bind is well-defined.

The discussion above suggests that we should work in the category of measurable spaces to get an induction hypothesis that allows us to conclude that our environments are mea-

[1] $\mathcal{V}[\![\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n]\!] \cong \mathcal{V}[\![\tau_1]\!] \times \cdots \times \mathcal{V}[\![\tau_n]\!]$

surable. However, it is well known that the category of measurable spaces is not Cartesian closed [7]. Hence, we will not be able to extend the language with higher-order functions. It is also unclear how to derive order-theoretic structure to support (least) fixed-points. Toronto *et al.* show how to add recursion in this setting, but model it operationally [97]. Moreover, unlike the finite discrete case, it is unclear how to implement a sampler for all continuous distributions on a Turing machine because there are a countable number of Turing machine configurations, but an uncountable number of continuous distributions.

3.2 Semantic Constructs

We assume the reader is familiar with denotational semantics as presented in standard texts (*e.g.*, as in Gunter [42] and Winskel [104]), including constructs such as complete partials orders (CPOs). Thus, we proceed directly to topological domains, an alternative theory of domains, following Battenfeld *et al.*'s introduction [13].[2]

A topological domain captures the structure necessary to support Type-2 computability as well as to take (least) fixed-points. Unlike a CPO, a topological domain in general does not carry the Scott topology, and hence, does not consider the partial order primary. Instead, topological domains start with the topology as primary and derive the order. The main issue is ensuring that directed chains in the resulting order have least upper bounds.

Recall that we can convert a topological space into a preordered set via the *specialization preorder*, which orders $x \sqsubseteq y$ if every open set that contains x also contains y. We write S to convert a topological space into a preordered set. Intuitively, $x \sqsubseteq y$ if x contains less information than y. For a computable metric space, we can always find an open ball that separates

[2] We also refer the reader to Battenfeld's dissertation [10].

two distinct points x and y (because the distance between two distinct points is positive). Hence, the specialization preorder of a computable metric space always gives the discrete order (*i.e.*, information ordering), and hence degenerately, a CPO. For example, the specialization preorder of the computable metric space $(\mathbb{R}, d, \mathbb{Q})$ is discrete.

Topological domain theory starts with the Cartesian closed category of qcb_0 spaces (see Section 2.2.3). A qcb_0 space is called a *topological predomain* (abbreviated TP) if every ascending chain $(x_i)_{i\in\mathbb{N}}$ (with respect to the specialization preorder $\sqsubseteq$) has an upper bound x such that $(x_i)_{i\in\mathbb{N}} \to x$ (with respect to its topology) [13, Defn. 5.1]. This definition ensures that least upper bounds of increasing chains exist.

The following provides a useful characterization of qcb_0 spaces. As a reminder, a topological space $(X, \mathcal{O}(X))$ is a *monotone convergence space* if its specialization order is a CPO and every open is Scott open [13, Defn. 5.3].

Proposition 3.2.1 ([13, Prop. 5.4])**.** *A qcb_0 space is a topological predomain iff it is a monotone convergence space.*

Hence, we see that the Scott topology is in general finer than the topology associated with a topological predomain.

Analogous to standard domain theory, call a topological predomain a *topological domain* (abbreviated TD) if it has least element, written $\bot$, under its specialization order [13, Defn. 5.6]. We can take the fixed-point of a continuous function $f : D \to D$ on a topological domain D in the obvious way as $\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$.

Proposition 3.2.2 ([13, Thm. 5.7])**.** *Every continuous function $f : D \to D$ has a least fixed-point.*

Category	Objects	Morphisms
CPO	CPO	continuous
CPPO	pointed CPO	continuous
$\mathbf{CPO}_{\perp!}$	pointed CPO	strict + continuous
TP	TP	continuous
TD	TD	continuous
$\mathbf{TD}_{\perp!}$	TD	strict + continuous

Figure 3.1: Summary of categories of domains.

	CPO	$\mathbf{CPO}_{\perp}$	$\mathbf{CPO}_{\perp!}$	**TP**	**TD**	$\mathbf{TD}_{\perp!}$
$D \times E$	✓+	✓+	✓+	✓+	✓+	✓+
$D \Rightarrow E$	✓+	✓+	✓	✓+	✓+	✓
$D + E$	✓+			✓+		
$D \otimes E$		✓	✓		✓	✓
$D \Rightarrow E$		✓	✓+		✓	✓+
$D \oplus E$		✓	✓+		✓	✓+
$D_{\perp}$	✓	✓	✓	✓	✓	✓

Figure 3.2: Summary of constructs on (pointed)CPOs and topological (pre)domains. The symbol ✓ indicates that the construct exists and the symbol + additionally indicates that it corresponds to the appropriate categorical construct.

Figure 3.1 summarizes the categories of domains we have available to give semantics. As we might expect, a CPO is analogous to a topological predomain and a pointed CPO is analogous to a topological domain. Moreover, as CPOs and topological (pre)domains contain much the same categorical structure, we will overload the constructs to work with both CPOs and topological domains. For example, we will write $\Rightarrow$ to refer to both a continuous function between CPOs and a continuous function between topological domains. When it is ambiguous, we will subscript the construct with the appropriate category. For instance, we write $\Rightarrow_{\mathbf{TD}}$ to indicate the function space from **TD**.

Figure 3.2 summarizes their categorical structure. The results about CPOs are well known (*e.g.*, see [4]). The results about topological (pre)domains are summarized by Battenfeld et

al. [13].[3] We write ✓ to indicate that the construct exists and additionally + to indicate that it corresponds to the appropriate categorical construct. We write $D \times E$ for products ($D \otimes E$ for coalesced products), $D \Rightarrow E$ for continuous functions ($D \Rightarrow E$ for strict continuous functions), and $D + E$ for sums ($D \oplus E$ for coalesced sums). $D_\perp$ lifts a topological (pre)domain. In summary, topological (pre)domains possess virtually all the same categorical structure as their CPO counterparts. Now, we recall the topological structure underlying topological (pre)domains as it is not necessarily the Scott topology.

The *sequentialization* of a topology is the canonical topology associated with a topological (pre)domain, similar to how the Scott topology is the canonical topology associated with a CPO. As a reminder, a set U is called *sequentially open* if every sequence converging to $x \in U$ is eventually contained in U.[4] A topology for which every open is sequentially open is called *sequential*. Every topological space X has a *sequentialization* $\mathbf{Seq}(X)$ defined on the same underlying set that adds all the sequentially open sets. Every countably based space (*e.g.*, a computable metric space) is sequential. We now recall the product and function spaces on topological (pre)domains.[5]

Let D and E be two topological predomains and consider $D \times E$. The underlying topology is given by the sequentialization of the product topology of D and E (as in [13, Sec. 4]). One can check that the specialization order gives the expected component-wise order.

Now, consider the function space $D \times E$. The underlying topology is given by the sequen-

[3]For results on **TP**, see [13, Thm. 5.5]. For results on **TD**, see [13, Thm. 5.9]. For results on $\mathbf{TD}_{\perp!}$, see [13, Thm. 6.1, Thm. 6.2, Prop. 6.4]

[4]The converse holds–if U is open, then a converging sequence by definition is eventually in U.

[5]Escardo *et al.* provide a more general introduction on how to create Cartesian closed categories of topological spaces using *probing* maps, of which sequentialization is a special case [28].

tialization of the compact open topology on D and E (as in [13, Sec. 4] and [12, Prop. 3.5]). As a reminder, the compact open topology $\mathscr{C}_{co}(D, E)$ has subbasic sets

$$S(K, U) = \{f \in \mathscr{C}(D, E) \mid f(K) \subseteq U\}$$

where $K \subseteq D$ is compact and $U \subseteq E$ is open. The specialization order gives the expected pointwise ordering. In addition, we get that suprema are constructed pointwise by [12, Lem. 3.13]. The function space of a topological (pre)domain differs the most from its CPO counterpart, so it is helpful to sketch out why the ordering on $D \Rightarrow E$ is pointwise.

Sketch 2. We recall a few facts. First, observe that Sierpinski space (Section 2.2.3) $\mathbb{S}$ is sequential (it is finite). Second, for any sequential space X and arbitrary space Y, $f : X \to Y$ is continuous iff $f : X \to \mathbf{Seq}(Y)$ is continuous. Thus, the collection of continuous functions $\{f \in \mathbb{S} \to X \mid f \text{ continuous}\}$ coincides with $\{f \in \mathbb{S} \to \mathbf{Seq}(X) \mid f \text{ continuous}\}$. In particular, this implies that the specialization ordering on X and $\mathbf{Seq}(X)$ are the same. Third, observe that the specialization ordering on the compact open topology is pointwise. Fourth and finally, recall that the function space on $D \Rightarrow E$ is given by the sequentialization of the compact open topology. Putting this together, we get that $D \Rightarrow E$ also has pointwise ordering.

We provide some examples of topological (pre)domains and comment on their relation to ordinary CPOs.

1. The naturals $\mathbb{N}$ with discrete topology is a topological predomain with discrete specialization order. The Scott topology on the resulting CPO is discrete, and hence, coincides.
2. The reals $\mathbb{R}$ with Euclidean topology form a topological predomain with discrete order (in particular, it is Hausdorff), but the Scott topology of $(\mathbb{R}, \sqsubseteq_{\text{Discrete}})$ is discrete.

Function	Type
$\text{const}(d) = e \mapsto d$	$D \Rightarrow (E \Rightarrow D)$
$\text{lift}(f) = \bot_D \mapsto \bot_E, d \mapsto f(d)$	$(D \Rightarrow E) \Rightarrow (D_\bot \Rightarrow E)$
$\pi_1(d, e) = d$	$D \times E \Rightarrow D$
$\pi_2(d, e) = e$	$D \times E \Rightarrow E$
$\langle f, g \rangle = d \mapsto (f(d), g(d))$	$(D \Rightarrow E) \times (D \Rightarrow F) \Rightarrow (D \Rightarrow E \times F)$
$\text{uncurry}(f) = (d, e) \mapsto f(d)(e)$	$(D \Rightarrow E \Rightarrow F) \Rightarrow (D \times E \Rightarrow F)$
$\text{curry}(f) = d, e \mapsto f(d, e)$	$(D \Rightarrow E \Rightarrow F) \Rightarrow (D \times E \Rightarrow F)$
$\text{apply}(f, d) = f(d)$	$(D \Rightarrow E \times D) \Rightarrow E$
$\text{fix}(f) = \bigsqcup_n f^n(\bot_D)$	$(D \Rightarrow D) \Rightarrow D$

Figure 3.3: Summary of operations.

3. The interval $[0, 1]$ with lower topology, *i.e.*, $\mathcal{O}([0, 1]) = \{(a, 1] \mid a \in [0, 1)\} \cup \{[0, 1]\}$ is a topological domain with order $\leq$. The Scott topology gives the lower topology and so the two coincide.

Now that we have some base examples, we can build some additional topological pre(domains using the constructs from Figure 3.2.

1. Let $L(X, \mathcal{O}(X)) = (\lfloor X \rfloor \cup \bot, \mathcal{O}(\lfloor X \rfloor) \cup \{\lfloor X \rfloor \cup \bot\})$. That is, it adds a $\bot$ element and a single open set $\lfloor X \rfloor \cup \bot$, while tagging all of X via $\lfloor X \rfloor$ to distinguish $\bot$. This corresponds to lifting a topological predomain D via $D_\bot$.
2. $\mathbb{N} \times_{\mathbf{CPO}} \mathbb{N} \cong \mathbb{N} \times_{\mathbf{TP}} \mathbb{N}$.
3. The function space $\mathbb{R} \Rightarrow_{\mathbf{TP}} \mathbb{R}$ gives the continuous functions with respect to the usual Euclidean topology. In contrast, the Scott continuous functions $\mathbb{R} \Rightarrow_{\mathbf{CPO}} \mathbb{R}$ contain all functions.

Figure 3.3 summarizes the functions (in the appropriate categories) we will use throughout the semantics. They are all standard and provided for reference.

3.3 Towards Semantics

In Chapter 4, I argue that by considering Type-2 computable distributions, we can support familiar language constructs, provide standard continuous reasoning principles, and realize program denotations as (Type-2) computable algorithms. All of the theory related to computable

distributions and their connections with domain theory that I will use has appeared at one point or another in the literature. Hence, my contribution is simply one of synthesizing these results in the context of a high-level probabilistic modeling language. Indeed, the intuitive reason for why paying attention to computability can address semantic issues for expressive probabilistic languages can be summarized by the motto: "Turing-complete probabilistic modeling languages express computable distributions." Of course, in hindsight, the motto is rather tautological!

4

An Application of Computable Distributions to the Semantics of Probabilistic Programming Languages

In this chapter, we describe a core probabilistic language called λ_{CD} that extends a PCF-like language with distributions and give λ_{CD} two semantics. The first semantics is based on topological domains whereas the second semantics is based on CPOs. Compared to the second

semantics, the first is cleaner because all the structure we need is captured categorically. The second semantics relies on an auxiliary argument involving the computability of environment lookup (see A.6.1).[1] Nevertheless, in light of our motto, it is still interesting to study how to extend a standard CPO model of PCF with distributions.

In each semantics, distributions will be assigned both a sampling interpretation and distributional interpretation. The distributional interpretation uses valuations, a topological variant of a distribution, instead of measures. Consequently, λ_{CD} will be restricted to distributions on topological spaces. Nevertheless, this includes spaces of practical interest such as reals and countable products of topological spaces. What we gain is that we can reason about probabilistic programs using the distributional semantics (see Section 4.6) and also faithfully implement the corresponding sampling semantics (see Section 4.5). Moreover, the restriction enables both semantics to support language features such as recursion and higher-order functions in a largely standard manner.

[1] I would like to thank Mitch Wand for pointing out this particular gap in the presentation of the CPO semantics.

4.1 A Core Language

The language λ_{CD} extends a PCF-like language with reals and distributions. We present the full syntax below for completeness.

$$\tau ::= \texttt{Nat} \mid \tau \to \tau \mid \tau \times \tau \mid \texttt{Real} \mid \texttt{Samp}\,\tau$$

$$e ::= x \mid n \mid \oplus e \mid \lambda x.\, e \mid e\; e \mid \texttt{if0}\; e\; \texttt{then}\; e\; \texttt{else}\; e \mid \mu x.\, e$$

$$\mid (e, e) \mid \texttt{fst}\; e \mid \texttt{snd}\; e$$

$$\mid r \mid \mathrm{op}^r(\overrightarrow{e}) \mid \mathit{dist} \mid \texttt{return}\; e \mid x \leftarrow e\; ;\; e$$

The first line of the expression syntax gives standard PCF-like expressions. For example, the syntax $\oplus e$ corresponds to primitive operations on naturals (*e.g.*, successor and predecessor). Because we are being somewhat vague about which primitive operations $\oplus$ on naturals are present in λ_{CD} and also omit booleans, we will refer to the fragment given in the first line of the expression syntax as PCF-like. The second line of the expression syntax adds pairs to the PCF-like fragment.

The third line of the expression syntax adds constants for reals r and application of primitive real functions $\mathrm{op}^r(\overrightarrow{e})$, and the constants *dist* represent primitive distributions on computable metric spaces. The expressions $\texttt{return}\; e$ and $x \leftarrow e\; ;\; e$ correspond to return and bind respectively in an appropriate probability monad.

The constants for reals and distributions can be given explicitly by their names (recall Section 2.2.3) because they are elements of the appropriate represented spaces. More concretely, if $\delta_{\mathbb{R}} : 2^\omega \rightharpoonup \mathbb{R}$ is the fast Cauchy representation of a real, then the constant r in λ_{CD} is a

$\boxed{\vdash_D \tau}$ Well-formed distribution type

$$\frac{}{\vdash_D \texttt{Nat}} \qquad \frac{}{\vdash_D \texttt{Real}} \qquad \frac{\vdash_D \tau_1 \qquad \vdash_D \tau_2}{\vdash_D \tau_1 \times \tau_2}$$

$\boxed{\Gamma \vdash e : \tau}$ Expression typing judgement

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \texttt{Nat}} \qquad \frac{\Psi(\mathrm{op}^r) = \tau_1 \times \cdots \times \tau_n \to \tau \qquad \Gamma \vdash e_i : \tau_i \text{ for } 1 \le i \le n}{\Gamma \vdash \mathrm{op}^r(e_1, \ldots, e_n) : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Nat} \qquad \Gamma \vdash e_2, e_3 : \tau}{\Gamma \vdash n : \texttt{if0}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mu x.\ e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}\ e : \tau_2} \qquad \frac{}{\Gamma \vdash r : \texttt{Real}} \qquad \frac{\Psi(\mathit{dist}) = \texttt{Samp}\ \tau \qquad \vdash_D \tau}{\Gamma \vdash \mathit{dist} : \texttt{Samp}\ \tau}$$

$$\frac{\Gamma \vdash e : \tau \qquad \vdash_D \tau}{\Gamma \vdash \texttt{return}\ e : \texttt{Samp}\ \tau} \qquad \frac{\Gamma \vdash e_1 : \texttt{Samp}\ \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \texttt{Samp}\ \tau_2 \qquad \vdash_D \tau_1, \tau_2}{\Gamma \vdash x \leftarrow e_1\ ;\ e_2 : \texttt{Samp}\ \tau_2}$$

Figure 4.1: The type-system for λ_{CD}. The expression typing judgement is parameterized by a context Ψ, which contains that types of primitive distributions and functions.

bit-stream such that $r \in \mathrm{dom}(\delta_{\mathbb{R}})$. In other words, constant reals and distributions can be named by bit-streams. Note that we do not need to restrict the constant reals or distributions present in λ_{CD} to be Type-2 computable. However, in a practical setting, we would restrict the constants provided by the language to be Type-2 computable so that they can be represented by an algorithm that computes their name.

Like PCF, λ_{CD} is a typed language. In addition to standard types, λ_{CD} includes the type of reals `Real` and the type of distributions `Samp` τ. Figure 4.1 summarizes the type-system for λ_{CD}. The expression typing judgement $\Gamma \vdash e : \tau$ is parameterized by a context Ψ (omitted in

the rules), which contains the types of primitive distributions and functions. The typing rules are largely standard.

For expressions that operate on distributions, the judgement $\vdash_D \tau$ additionally enforces that the involved types are well-formed. The distribution type `Samp` τ is well-formed if the space denoted by τ supports the operations required of a computable metric space. This includes the natural type `Nat`, the real type `Real`, and products of well-formed types $\tau_1 \times \tau_2$.[2]

4.2 Distribution Constructions

In this section, we gather results about valuations and integration. Our goal is to present these results in a form amenable for giving the semantics of λ_{CD}, as they have often appeared in the literature under a different guise (*e.g.*, to study Type-2 computability).

4.2.1 Valuations

Let X be a topological space. A *valuation* $\nu : \mathcal{O}(X) \to [0,1]$ is a function that assigns to each open set of a topological space a probability, such that it is strict ($\nu(\emptyset) = 0$), monotone ($\nu(U) \leq \nu(V)$ for $U \subseteq V$), and modular ($\nu(U) + \nu(V) = \nu(U \cup V) + \nu(U \cap V)$ for every open U and V). A valuation shares many of the same properties as a measure, and hence, can be seen as a topological variation of distribution. Notably, unlike measures, valuations are not required to satisfy countable additivity.

However, every Borel measure μ can be restricted to the lattice of opens, written $\mu|_{\mathcal{O}(X)}$, resulting in an ω-continuous valuation. As a reminder, a valuation ν is called *ω-continuous* if

[2]We can also support distributions on distributions, etc., but restrict our attention to ground types for simplicity.

$\nu(\bigcup_{n\in\mathbb{N}} V_n) = \sup_{n\in\mathbb{N}} \nu(V_n)$ for $(V_n)_{n\in\mathbb{N}}$ an increasing sequence of opens. Hence, the countable additivity of μ encodes the ω-continuous property. Indeed, there are many connections between Borel measures, valuations, and Type-2 computable distributions.

Let $\mathcal{O}^{\subseteq}(X) = (\mathcal{O}(X), \subseteq)$ be the lattice of opens (and hence a CPO) of a topological space X ordered by subset inclusion. Let $[0,1]^{\uparrow} \triangleq ([0,1], \leq)$ be the interval $[0,1]$ ordered by $\leq$. We have the following relationships between $\mathcal{O}^{\subseteq}(X)$ and $[0,1]^{\uparrow}$ as CPOs and represented spaces (and hence topological domains):

Proposition 4.2.1.

1. $[0,1]^{\uparrow} \cong ([0,1], \delta_{<})$ *where* $\delta_{<}$ *is a representation that induces the lower topology*[3] *and*
2. $\mathcal{O}^{\subseteq}(X) \cong \mathscr{C}(X, \mathbb{S})$ *when* X *is an admissible represented space (see Schröder [88, Thm. 3.3] and its proof).*[4]

The function space between $\mathcal{O}^{\subseteq}(X)$ and $[0,1]^{\uparrow}$ is indicative of the type of a distribution.

Proposition 4.2.2. *Let* $(X, \mathcal{O}(X))$ *be a topological space.*

1. *Every Borel measure* μ *on* X *can be restricted to an* ω*-continuous valuation* $\mu|_{\mathcal{O}(X)} : [\mathcal{O}^{\subseteq}(X) \Rightarrow_{\boldsymbol{CPO}} [0,1]^{\uparrow}]$ *(see Schröder [88, Sec. 3.1]). Moreover,* μ *is uniquely determined by its restriction to the opens* $\mu|_{\mathcal{O}(X)}$.[5]
2. *When* X *is countably based,* $[\mathcal{O}^{\subseteq}(X) \Rightarrow_{\boldsymbol{CPO}} [0,1]^{\uparrow}] \cong \mathscr{C}(\mathcal{O}(X), [0,1]_{<})$ *(see Schroder [88, Sec. 3.1, Thm 3.5, Cor. 3.5]).*

---

[3] We saw this as an example previously in Section 3.2.

[4] Alternatively, recall that $\mathcal{O}(X) \cong \mathscr{C}(X, \mathbb{S})$. As X is an admissible represented space, it is sequentially generated [28] and hence also compactly generated [28]. Hence, $\mathscr{C}(X, \mathbb{S}) \cong \mathbf{Seq}(\mathscr{C}_{co}(X, \mathbb{S}))$, which can be shown to carry the Scott topology (see [Prop. 3.7][12]).

[5] Note that the ω-continuous condition encodes what it means for a function to be ω-Scott continuous, *i.e.*, an ω-CPO continuous function. The second part follows by an application of the π-λ theorem.

3. *When X is countablye based,* $[\mathcal{O}^{\subseteq}(X) \Rightarrow_{\boldsymbol{CPO}} [0,1]^{\uparrow}] \cong [\mathcal{O}^{\subseteq}(X) \Rightarrow_{\boldsymbol{TD}} [0,1]^{\uparrow}]$.[6]

In light of proposition 4.2.2, we will interchangeably use Borel distributions and valuations. As we will further restrict our attention to distributions on computable metric spaces (and hence countably based spaces) in λ_{CD}, proposition 4.2.2 gives three equivalent views of a valuation as (1) a CPO continuous function, (2) a continuous map between represented spaces, and (3) a continuous function between topological domains. (1) and (3) can be used to give semantics. (2) indicates that there is an associated theory of effectivity on valuations that we can take advantage of. Now that we have finished recalling some properties of valuations, we will continue to integration.

4.2.2 Integration

Let X be a represented space and $\mu \in \mathcal{M}_1(X)$, where $\mathcal{M}_1(X)$ is the collection of Borel measures on X that have total measure 1.

Proposition 4.2.3. *The integral of a lower semi-continuous function $f \in \mathscr{C}(X, [0,1]_<)$ with respect to a Borel measure μ*

$$\int : \mathscr{C}(X, [0,1]_<) \times \mathcal{M}_1(X) \to [0,1]_<$$

is lower semi-continuous (see [88, Prop. 3.6]). In fact, it is even lower semi-computable (see [88, Prop. 3.6] and [46, Prop. 4.3.1]).

[6] Recall that every ω-continuous pointed CPO with its Scott topology coincides with a topological domain [13]. The least element is the valuation that maps every open set to 0.

As a reminder, (Lebesgue) integration of a non-negative function f with respect to a measure μ is defined as

$$\int f \, d\mu \triangleq \sup\{\int s \, d\mu \mid s \leq f, \text{ simple}\}$$

where a simple function s is of the form

$$s(x) = \sum_{i=1}^{N} a_i \mathbb{1}_{B_i}(\cdot)$$

for B_i measurable and

$$\int s \, d\mu = \sum_{i=1}^{N} a_i \mu(B_i) \, .$$

The idea is to define the integral of a non-negative function f by approximating it from below by simple functions, *i.e.*, finite sums of step functions. In particular, we can define an integral of a simple function as a finite sum. Once we have done so, the key idea is to show that taking the limit of the integral of simple functions is equivalent to the integral of the limit of the simple functions.

$$\lim_{n\to\infty} \int f_n \, d\mu = \int \lim_{n\to\infty} f_n \, d\mu$$

In other words, we show that the limit commutes with the integration operator. This is known as Monotone Convergence in measure theory. In an order theoretic setting, it corresponds to Scott-continuity of the integration operator.

Remark (Notation). We will use the same symbol $\int$ for integrating with respect to an ω-continuous valuation. The overloading of notation is justified because every Borel measure $\mu \in \mathcal{M}_1(X)$ on a space X is uniquely determined by its restriction to an ω-continuous valuation $\mu|_{\mathcal{O}(X)}$ (recall proposition 4.2.2).

Remark (Countably based spaces). Integration on countably based spaces is nicer compared to integration on more general spaces. Let σ close a collection of sets under the σ-algebra conditions. Let X and Y be two topological spaces. It is well known that $\sigma(\mathcal{O}(X) \otimes \mathcal{O}(Y)) \neq \sigma(\mathcal{O}(X)) \times \sigma(\mathcal{O}(Y))$, where the left forms the product σ-algebra via $\otimes$ after closing each component separately while the right closes the product topology. However, the left is equivalent to the right when one of X or Y is countably based. Consequently, when restricted to countably based spaces, we can rely on standard measure-theoretic integration theorems (*e.g.*, Fubini) without having to show that an analogous result holds on valuations.

4.2.3 Semantic Constructions

Now, we combine the results about valuations and integration to define a probability monad. The constructions we define here will be used later in the semantics of λ_{CD}. We start with constructions for a sampling interpretation.

Define an (endo)functor $\mathcal{S}$ that sends a countably based toplogical predomain D to a sampler on D and a morphism to one that composes with the underlying sampler. Then, $\mathcal{S}(D)$ is a sampler producing values in D.

Proposition 4.2.4. *The functor $\mathcal{S}$ is well-defined.*

$$\mathcal{S}(D) \triangleq 2^\omega \Rightarrow D_\perp$$

$$\mathcal{S}(f : D \Rightarrow E) \triangleq s \mapsto \mathit{lift}(f) \circ s \ .$$

Proof. Let D be a countably based topological predomain. First, $2^\omega \Rightarrow D_\perp$ is clearly a topological predomain (it even has least element $\cdot \mapsto \perp$), so we check that it is countably based.

Recall that if X and Y are countably based spaces where X is additionally locally compact, then the space of continuous functions from X to Y with compact-open topology is countably based [cite]. 2^ω is compact and hence locally compact, so $2^\omega \Rightarrow D_\perp$ with compact open topology is countably based. Finally, in this case, recall the function space $\Rightarrow$ gives the sequentialization of the compact open topology (see Section 3.2) and a countably based space is sequential, so the result follows.

Second, $\mathcal{S}(f)$ is clearly well-defined. Third, we check the functor axioms. The identity clearly holds and associativity follows by the following calculation.

$$\begin{aligned}
\mathcal{S}(g \circ f)(s) &= \text{lift}(g \circ f) \circ s \\
&= \text{lift}(g) \circ \text{lift}(f) \circ s \\
&= \text{lift}(g) \circ \mathcal{S}(f) \\
&= \mathcal{S}(g) \circ \mathcal{S}(f)
\end{aligned}$$

□

Next, we define some operations on samplers.

$$\begin{aligned}
&\det : X \Rightarrow \mathcal{S}(X)_\perp \\
&\det = x \mapsto \text{lift}(\text{const}(x))
\end{aligned}$$

creates a sampler that ignores its input bit-randomness and always returns x. The function

$$\text{split} : 2^\omega \Rightarrow 2^\omega \times 2^\omega$$

$$\text{split} = u \mapsto (u_e, u_o)$$

splits an input bit-stream u into the bit-streams indexed by the even indices u_e and the odd indices u_o. The function

$$\text{samp} : \mathcal{S}(X) \Rightarrow (X \Rightarrow \mathcal{S}(Y)) \Rightarrow \mathcal{S}(Y)$$

$$\text{samp} = s, f \mapsto \text{lift}(\text{uncurry}(f)) \circ \text{lift}(\langle s \circ \pi_1, \pi_2 \rangle) \circ \text{split}$$

splits the input bit-randomness and runs the sampler s on one of the bit-streams obtained by splitting to produce a value. That value is fed to f, which in turn produces a sampler that is run on the other bit-stream obtained by splitting.

We may recognize det and samp as return and bind respectively in a sampling monad. We will not do that here, opting instead to construct a probability monad and deriving the associated laws after relating the sampling functions (see Section 4.3).

Define the (endo)functor $\mathscr{P}$ on countably based topological predomains that sends an object D to the space of valuations on D and a morphism to one that computes the pushforward.

Proposition 4.2.5. *The functor $\mathscr{P}$ is well-defined.*

$$\mathscr{P}(D) \triangleq \mathcal{O}(D) \Rightarrow [0,1]^{\uparrow}$$

$$\mathscr{P}(f : D \Rightarrow E) \triangleq \mu \mapsto \mu \circ f^{-1}\,.$$

Proof. First, we have that $\mathscr{P}(D)$ is a countably based represented space by Schröder [88, Cor. 3.5] and hence topological predomain by proposition 4.2.2. Second, we verify that the push-forward is continuous. Again as a consequence of proposition 4.2.2, we can check that the pushforward preserves directed suprema. Let $(U_n)_{n\in\mathbb{N}}$ an increasing sequence of opens.

$$\begin{aligned}
\bigcup_n \mathscr{P}(f)(\mu)(U_n) &= \bigcup_n (\mu \circ f^{-1})(U_n) \\
&= (\mu \circ f^{-1})(\bigcup_n U_n) && \text{(continuity)} \\
&= \mathscr{P}(f)(\mu)(\bigcup_n U_n)
\end{aligned}$$

Third, we need to check the functor laws. Clearly, $\mathscr{P}(id)(\mu) = \mu$. Finally, we check functor composition.

$$\begin{aligned}
\mathscr{P}(g \circ f)(\mu) &= \mu \circ (g \circ f)^{-1} \\
&= (\mu \circ f^{-1}) \circ g^{-1} \\
&= \mathscr{P}(g)(\mathscr{P}(f)(\mu)) \\
&= (\mathscr{P}(g) \circ \mathscr{P}(f))(\mu)
\end{aligned}$$

□

We can construct a probability monad using the functor $\mathscr{P}$.

Proposition 4.2.6. *The triple* $(\mathscr{P}, \eta, \succ_\flat)$ *is a monad, where*

$$\eta(x)(U) \triangleq \mathbb{1}_U(x)$$
$$(\mu \succ_\flat f)(U) \triangleq \int f_U \, d\mu \text{ for } f_U(\cdot) = f(\cdot)(U).$$

Proof. First, the step function given by $\eta(x)$ is a valuation. Second, observe that uncurry(f) is a probability kernel—uncurry$(f)(\cdot, U)$ is lower semi-continuous for any open U so it is measurable and uncurry$(f)(x, \cdot)$ for any $x \in X$ is a valuation. Hence, $U \mapsto \int f_U \; d\mu$ is the integral of a probability kernel, which produces a (Borel) measure that can be restricted to an ω-continuous valuation. Third, we check the monad laws. Clearly, the left and right identity laws hold. To show associativity, we first compute $((\mu \succ_\flat f) \succ_\flat g)$ and $\mu \succ_\flat (x \mapsto fx \succ_\flat g)$, where $\mu \in \mathscr{P}(X)$, $f : X \Rightarrow \mathscr{P}(Y)$, and $g : Y \Rightarrow \mathscr{P}(Z)$. For the former, we have

$$\begin{aligned}((\mu \succ_\flat f) \succ_\flat g)(U) &= \int_Y g_U \, d(\mu \succ_\flat f) \\ &= \int_Y g_U \, d\nu \,,\end{aligned}$$

where

$$\nu = V \mapsto \int_x f_V \, d\mu \,.$$

For the latter, we have

$$\begin{aligned}\mu \succ_\flat (x \mapsto f(x) \succ_\flat g)(U) &= \int_X (x \mapsto f(x) \succ_\flat g)_U \, d\mu \\ &= \int_X x \mapsto \int_Y g_U \, d(f(x)) \, d\mu \, .\end{aligned}$$

Thus, we need to show

$$\int_Y g_U \, d\nu = \int_X x \mapsto \int_Y g_U \, d(f(x)) \, d\mu$$

for some open U. We can use Monotone Convergence. Let $(g_U^n)_{n\in\mathbb{N}}$ be a sequence of simple functions that converges to g_U pointwise. Write the n-th simple function as $\sum_k a_k^n 1_{A_k^n}(\cdot)$. Then,

$$\int_Y g_U \, d\nu = \sum_k a_k (\int_X f_{A_k} \, d\mu)$$

and

$$\begin{aligned}\int_X x \mapsto \int g_U \, d(f(x)) d\mu &= \int_X x \mapsto \sum_k a_k f(x)(A_k) \, d\mu \\ &= \int_X x \mapsto \sum_k a_k f_{A_k}(x) \, d\mu \\ &= \sum_k a_k \int_X f_{A_k} \, d\mu\end{aligned}$$

where the last equality holds by linearity of the integral. This shows that the integrals are equivalent for each g_n and so the integrals are equal by Monotone Convergence. □

4.3 A Topological Domain Semantics

We are now ready to give a semantics to λ_{CD} based on topological domains using the constructs from Section 4.2.

4.3.1 Interpretation of Types

The interpretation of types $\mathcal{V}[\![\tau]\!]$ interprets a type τ as a topological domain and is defined by induction on types. We summarize the interpretation of types $\mathcal{V}[\![\tau]\!]$ below, which follows a call-by-name evaluation strategy.

$$\begin{aligned}
\mathcal{V}[\![\texttt{Nat}]\!] &\triangleq (\mathbb{N}, d_{\text{Discrete}}, \mathbb{N})_\perp \\
\mathcal{V}[\![\tau_1 \times \tau_2]\!] &\triangleq (\mathcal{V}[\![\tau_1]\!] \times \mathcal{V}[\![\tau_2]\!])_\perp \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!] &\triangleq (\mathcal{V}[\![\tau_1]\!] \Rightarrow \mathcal{V}[\![\tau_2]\!])_\perp \\
\mathcal{V}[\![\texttt{Real}]\!] &\triangleq (\mathbb{R}, d_{\text{Euclid}}, \mathbb{Q})_\perp \\
\mathcal{V}[\![\texttt{Samp}\ \tau]\!] &\triangleq \Sigma_{s \in \mathcal{S}(\mathcal{V}[\![\tau]\!])}\, \text{psh}_\tau(s) \text{ where } \text{psh}_\tau(s) \in \mathscr{P}(\mathcal{V}[\![\tau]\!])
\end{aligned}$$

The interpretation of standard types is similar to what one might expect from a standard CPO call-by-name interpretation. Indeed, the only difference is that we replace CPO constructs with their topological domain counterparts. In the interpretation of the types `Nat` and `Real`, we have explicitly written out the topologies (as a computable metric space). Combining the sampling functor $\mathcal{S}$ with the probability monad $\mathscr{P}$ gives the interpretation of a distribution type `Samp` τ. The function psh_τ computes the pushforward and relates the sampler with the valuation on a space denoted by τ. Note that the well-formed distribution judgement

$$
\begin{aligned}
\mathcal{E}[\![x]\!]\rho &\triangleq \rho(x) \\
\mathcal{E}[\![n]\!]\rho &\triangleq \Upsilon(n) \\
\mathcal{E}[\![\oplus e]\!]\rho &\triangleq \Upsilon(\oplus)(\mathcal{E}[\![e]\!]\rho) \\
\mathcal{E}[\![\lambda x.\ e]\!]\rho &\triangleq \lfloor \bar{x} \mapsto \mathcal{E}[\![e]\!]\rho[x \mapsto \bar{x}] \rfloor \\
\mathcal{E}[\![e_1\ e_2]\!]\rho &\triangleq \mathrm{unlift}(\mathcal{E}[\![e_1]\!]\rho)(\mathcal{E}[\![e_2]\!]\rho) \\
\mathcal{E}[\![\texttt{if0}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\rho &\triangleq \begin{cases} \mathcal{E}[\![e_2]\!]\rho & \text{if } \mathcal{E}[\![e_1]\!]\rho = \bar{0} \\ \mathcal{E}[\![e_3]\!]\rho & \text{if } \mathcal{E}[\![e_1]\!]\rho = \bar{n} \text{ for } \bar{n} \neq \bar{0} \\ \bot & \text{otherwise} \end{cases} \\
\mathcal{E}[\![\mu x.\ e]\!]\rho &\triangleq \mathrm{fix}(\bar{x} \mapsto \mathcal{E}[\![e]\!]\rho[x \mapsto \bar{x}]) \\
\mathcal{E}[\![(e_1, e_2)]\!]\rho &\triangleq \lfloor (\mathcal{E}[\![e_1]\!]\rho, \mathcal{E}[\![e_2]\!]\rho) \rfloor \\
\mathcal{E}[\![\texttt{fst}\ e]\!]\rho &\triangleq \pi_1(\mathrm{unlift}(\mathcal{E}[\![e]\!]\rho)) \\
\mathcal{E}[\![\texttt{snd}\ e]\!]\rho &\triangleq \pi_2(\mathrm{unlift}(\mathcal{E}[\![e]\!]\rho)) \\
\mathcal{E}[\![r]\!]\rho &\triangleq \Upsilon(r) \\
\mathcal{E}[\![\mathrm{op}^r(e_1, \ldots, e_n)]\!]\rho &\triangleq \Upsilon(\mathrm{op}^r)(\mathcal{E}[\![e_1]\!]\rho, \ldots, \mathcal{E}[\![e_n]\!]\rho) \\
\mathcal{E}[\![\mathit{dist}]\!]\rho &\triangleq \Upsilon(\mathit{dist}) \\
\mathcal{E}[\![\Gamma \vdash \texttt{return}\ e : \texttt{Samp}\ \tau]\!]\rho &\triangleq (\det(\mathcal{E}[\![e]\!]\rho), \eta(\mathcal{E}[\![e]\!]\rho)) \\
\mathcal{E}[\![\Gamma \vdash x \leftarrow e_1\ ;\ e_2 : \texttt{Samp}\ \tau_2]\!]\rho &= (\mathrm{samp}(\pi_1(\mathcal{E}[\![e_1]\!]\rho))(\bar{x} \mapsto \pi_1(\mathcal{E}[\![e_2]\!]\rho[x \mapsto \bar{x}]) \\
&\quad , \pi_2(\mathcal{E}[\![e_1]\!]\rho) \succ_\flat \bar{x} \mapsto \pi_2(\mathcal{E}[\![e_2]\!]\rho[x \mapsto \bar{x}]))
\end{aligned}
$$

Figure 4.2: The denotational semantics of λ_{CD}. We write $\bar{n}$ to be the semantic value corresponding to the syntax n.

$\vdash_D \tau$ ensures that the probability monad $\mathscr{P}$ is applied to only the countably based topological (pre)domains.

4.3.2 Expression Denotation

The expression denotation function $\mathcal{E}[\![\Gamma \vdash e : \tau]\!] : \mathcal{V}[\![\Gamma]\!] \Rightarrow \mathcal{V}[\![\tau]\!]$ where is defined by induction on the typing derivation and is summarized in Figure 4.2. It is parameterized by a global environment Υ that interprets constant reals r, constant primitive distributions *dist*,

and primitive functions ($\oplus$ and op^r). The global environment Υ should be well-formed. As shorthand, we will put a bar over constants to represent the semantic value obtained from a global environment lookup (*e.g.*, $\Upsilon(n) = \bar{n}$) to distinguish the semantic value from the syntax.

- For any $\oplus \in \mathrm{dom}(\Upsilon)$, the corresponding semantic function $\bar{\oplus}$ is strict and computable on its domain.
- For any $r \in \mathrm{dom}(\Upsilon)$, $r \in \mathrm{dom}(\delta_{\mathbb{R}})$ where $\delta_{\mathbb{R}}$ is the representation of reals. In other words, r should be the name of a real $\bar{r}$.
- For any $\mathit{dist} \in \mathrm{dom}(\Upsilon)$, $\mathit{dist} \in \mathrm{dom}(\delta_{(2^\omega \Rightarrow X)\times(\mathcal{O}(X)\Rightarrow[0,1]^\uparrow)})$ where $\delta_{(2^\omega \Rightarrow X)\times(\mathcal{O}(X)\Rightarrow[0,1]^\uparrow)}$ is the representation of a pair of a sampler and valuation on the space X. In other words, dist should be the name of a pair $\overline{\mathit{dist}}$ of a sampler and the corresponding distribution.
- For any $\mathrm{op}^r \in \mathrm{dom}(\Upsilon)$, the corresponding semantic function $\bar{\mathrm{op}}^r$ is strict and continuous on its domain.

Now, we will walkthrough the expression denotation function. As the denotation of expressions corresponding to the PCF-like fragment are standard, we will focus on the constructs λ_{CD} introduces. The denotation of a constant real r is a global environment lookup.

$$\mathcal{E}[\![r]\!]\rho \triangleq \Upsilon(r)$$

As a reminder, if r gives the name of the corresponding semantic element, written $\bar{r}$, then a global environment lookup corresponds to applying the representation $\delta_{\mathbb{R}}$ (*i.e.*, $\Upsilon(r) = \delta_{\mathbb{R}}(r) = \bar{r}$). The denotation of a primitive function on reals $\mathrm{op}^r(e_1, \ldots, e_n)$ applies the semantic function $\Upsilon(\mathrm{op}^r)$ to the denotation of each of its arguments.

$$\mathcal{E}[\![\mathrm{op}^r(e_1, \ldots, e_n)]\!]\rho \triangleq \Upsilon(\mathrm{op}^r)(\mathcal{E}[\![e_1]\!]\rho, \ldots, \mathcal{E}[\![e_n]\!]\rho)$$

By well-formedness of the global environment Υ, $\Upsilon(\mathrm{op}^r)$ denotes a strict function. Hence, it

produces $\bot$ whenever any of its arguments denotes $\bot$. At the level of Oracle Turing machines, this means that $\Upsilon(\mathrm{op}^r)$ diverges when any of its arguments diverges. Next, we will go over the denotation of distribution constructs in λ_{CD}.

The denotation of a constant primitive distribution *dist* is a global environment lookup. As a reminder, the interpretation of $\texttt{Samp}\ \tau$ is a pair of a sampler and valuation so the lookup should also produce a pair.

$$\mathcal{E}[\![\mathit{dist}]\!]\rho \triangleq \Upsilon(\mathit{dist})$$

The denotation of $\texttt{return}\ e$ produces a pair of a sampler that ignores the input bit-randomness and returns $\mathcal{E}[\![e]\!]\rho$, and a point mass valuation centered at $\mathcal{E}[\![e]\!]\rho$.

$$\mathcal{E}[\![\Gamma \vdash \texttt{return}\ e : \texttt{Samp}\ \tau]\!]\rho \triangleq (\det(\mathcal{E}[\![e]\!]\rho), \eta(\mathcal{E}[\![e]\!]\rho))$$

The meaning of $x \leftarrow e_1\ ;\ e_2$ also gives a sampler and a valuation.

$$\mathcal{E}[\![\Gamma \vdash x \leftarrow e_1\ ;\ e_2 : \texttt{Samp}\ \tau_2]\!]\rho =$$
$$(\mathrm{samp}(\pi_1(\mathcal{E}[\![e_1]\!]\rho))(\bar{x} \mapsto \pi_1(\mathcal{E}[\![e_2]\!]\rho[x \mapsto \bar{x}]),$$
$$\pi_2(\mathcal{E}[\![e_1]\!]\rho) \succ_\flat \bar{x} \mapsto \pi_2(\mathcal{E}[\![e_2]\!]\rho[x \mapsto \bar{x}]))$$

Under the sampling view, we use samp to compose the sampler obtained by $\pi_1(\mathcal{E}[\![e_1]\!]\rho)$ with the function $\bar{x} \mapsto \pi_1(\mathcal{E}[\![e_2]\!]\rho[x \mapsto \bar{x}])$. Under the valuation component, we reweigh $\bar{x} \mapsto \pi_2(\mathcal{E}[\![e_2]\!]\rho[x \mapsto \bar{x}])$ according to the valuation $\pi_2(\mathcal{E}[\![e_1]\!]\rho)$ using monad bind $\succ_\flat$ from $\mathscr{P}$.

Note that the sampling component and the valuation component were independent of each

other in each of the distribution expressions. Hence, we could have given two different semantics and related them. However, in the current form, we will obtain that the valuation is the pushforward along the sampler by showing that expression denotation function is well-defined.

The structure of the argument showing that the expression denotation function is well-defined is virtually identical to the argument for showing that the CPO semantics of PCF is well-defined. The interesting cases correspond to `return` e and $x \leftarrow e_1 \ ; \ e_2$ where we need to relate the sampling component with the valuation it denotes. We accomplish this with an auxiliary lemma.

Lemma 4.3.1 (Push)**.** *Let X and Y be countably based represented spaces.*

1. $\mathrm{psh}(\mathrm{det}(x)) = \eta(x)$ *for any* $x \in X$.
2. $\mathrm{psh}(\mathrm{samp}(s)(f)) = \mathrm{psh}(s) \succ_\flat v \mapsto \mathrm{psh}(f(v))$ *for any* $s : \mathcal{S}(X)$ *and* $f : X \to \mathcal{S}(Y)$.

Proof. The first claim is straightforward.

$$\begin{aligned}
\mathrm{psh}(\mathrm{det}(x)) &= \mathrm{psh}(\mathrm{lift}(\mathrm{const}(x))) \\
&= U \mapsto \mathrm{lift}(\mu_{\mathrm{iid}})(\{u \mid x \in U\}) \\
&= U \mapsto \mathbb{1}_U(x) \\
&= \eta(x)
\end{aligned}$$

The second item is slightly more involved. Let $\bar{s} = \mathrm{lift}(\langle s \circ \pi_1, \pi_2 \rangle) \circ \mathrm{split}$ and $\bar{f} = \mathrm{lift}(\mathrm{uncurry}(f))$

so that $\mathrm{samp}(s)(f) = \bar{f} \circ \bar{s}$.

$$\begin{aligned}
\mathrm{psh}(\mathrm{samp}(s)(f)) &= \mathrm{psh}(\bar{f} \circ \bar{s}) \\
&= U \mapsto \mathrm{lift}(\mu_{\mathrm{iid}})(\bar{s}^{-1}(\bar{f}^{-1}(U))) \\
&= U \mapsto \int (v,u) \mapsto \mathbb{1}_{\bar{f}^{-1}(U)}((v,u))\, d(\mathrm{psh}(\bar{s})) \\
&= U \mapsto \int (v,u) \mapsto \mathbb{1}_{\bar{f}^{-1}(U)}((v,u))\, d(\mathrm{psh}(s) \otimes \mathrm{lift}(\mu_{\mathrm{iid}})) \\
&= U \mapsto \int v \mapsto \mathrm{lift}(\mu_{\mathrm{iid}})(\bar{f}(v))^{-1}(U)\, d(\mathrm{psh}(s)) \\
&= \mathrm{psh}(s) \succ_\flat \mathrm{psh}(f)
\end{aligned}$$

Crucially, we need to use the fact that split produces independent bit-streams. □

Now, we check that the denotation is well-defined.

Lemma 4.3.2. *Let Υ be a well-formed global environment. If $\Gamma \vdash e : \tau$, then $\mathcal{E}[\![e]\!] : \mathcal{V}[\![\Gamma]\!] \Rightarrow \mathcal{V}[\![\tau]\!]$*

Proof. By induction on the derivation of $\Gamma \vdash e : \tau$. The cases involving the PCF-like fragment are identical to the CPO proof. The cases involving constant reals and distributions, $\Gamma \vdash r :$ `Real` and $\Gamma \vdash \mathit{dist} : \texttt{Samp}\ \tau$ follow from continuity of const and well-formedness of Υ.

Case 1 ($\Gamma \vdash \mathrm{op}^r(e_1, \ldots, e_n) : \texttt{Real}$). By definition, $\mathcal{E}[\![\mathrm{op}^r(e_1, \ldots, e_n)]\!] = \mathrm{apply} \circ \langle \mathrm{lift}(\Upsilon(\mathrm{op}^r)), \langle \mathcal{E}[\![e_1]\!], \ldots, \mathcal{E}[\![e_n]\!] \rangle \rangle$, which is continuous if we can show $\mathcal{E}[\![e_i]\!]$ is continuous for $1 \leq i \leq n$. This follows by the induction hypothesis.

Case 2 ($\Gamma \vdash \texttt{return}\ e : \texttt{Samp}\ \tau$). By definition, $\mathcal{E}[\![\texttt{return}\ e]\!] = \langle \mathrm{lift}(\mathrm{const}) \circ \mathcal{E}[\![e]\!], \eta \circ \mathcal{E}[\![e]\!] \rangle$, which is continuous as $\mathcal{E}[\![e]\!]$ is continuous by the induction hypothesis. The sampler is related to the valuation by the push lemma 4.3.1.

Case 3 ($\Gamma \vdash x \leftarrow e_1 \ ; \ e_2 : \texttt{Samp}\ \tau_2$). As in the other cases, we can write the denotation in point-free form. We have $\mathcal{E}[\![x \leftarrow e_1 \ ; \ e_2]\!]\rho = (s, \mu)$, where

$$s = \text{apply} \circ \langle \text{const}(\text{samp}), \langle \pi_1 \circ \mathcal{E}[\![e_1]\!], \text{uncurry}(\pi_1 \circ \mathcal{E}[\![e_2]\!]) \rangle \rangle$$

$$\mu = \text{apply} \circ \langle \text{const}(\succ_\flat), \langle \pi_2 \circ \mathcal{E}[\![e_1]\!], \text{uncurry}(\pi_2 \circ \mathcal{E}[\![e_2]\!]) \rangle \rangle \ .$$

By our induction hypothesis, we obtain that $\mathcal{E}[\![e_1]\!] : \mathcal{V}[\![\Gamma]\!] \Rightarrow \mathcal{V}[\![\texttt{Samp}\ \tau_1]\!]$ and $\mathcal{E}[\![e_2]\!] : \mathcal{V}[\![\Gamma]\!] \times \mathcal{V}[\![\tau_1]\!] \Rightarrow \mathcal{V}[\![\tau_2]\!]$. Thus, the denotation is well-defined as it is a composition of continuous functions. As before, the sampler is related to the valuation by the push lemma 4.3.1.

□

Hence, the expression denotation function is well-defined. As we just saw, topological domains provide all the necessary structure for giving semantics to a probabilistic programming language extended with continuous distributions. Indeed, they even provide an associated computability theory so we will later be able to implement the sampling semantics faithfully (Section 4.5). However, the tradeoff was that we restricted λ_{CD} to distributions on computable metric spaces. Nevertheless, as we mentioned previously, this captures essentially all spaces used in practice, including reals, (countable) products of computable metric spaces, and distributions on distributions.

Remark (Primitive functions 1). In the setting with reals and continuous distributions, we will need to be slightly more careful with the domains of primitive functions to ensure that they are continuous with respect to the appropriate topologies. For instance, consider the functions $+_1 : \mathbb{R} \to \mathbb{R}$ that adds 1 to its argument and $\log : \mathbb{R}^+ \to \mathbb{R}$. The former is a continuous function that defined on all of $\mathbb{R}$. The latter is a continuous function when restricted to the

subspace $\mathbb{R}^+$. In other words, we should model the former with the topological domain $\mathbb{R} \Rightarrow \mathbb{R}_\perp$ and the latter with the topological domain $\mathbb{R}^+ \Rightarrow \mathbb{R}_\perp$, where we equip the non-negative reals $\mathbb{R}^+$ with the subspace topology.

As our interpretation of types is given by induction on types, we will need additional types (and primitives) in λ_{CD} to distinguish between these two domains if we hope to support such primitives. More concretely, we could support a partial function such as log by adding the additional type $\texttt{Real}^+$ that denotes the appropriate topological domain, a conversion function $\texttt{nonneg\_to\_real} : \texttt{Real}^+ \to \texttt{Real}$ with the expected semantics, and another conversion function $\texttt{real\_to\_nonneg} : \texttt{Real} \to \texttt{Real}^+$ that diverges if called with a negative number.

Remark (Primitive functions 2). In addition to the previous remark on primitive functions, we will also need to be slightly careful to note when the semantics of λ_{CD} primitives are different from the "usual" semantics. For instance, consider the function $\texttt{<}$ which we might assign the type $\texttt{Real} \times \texttt{Real} \to \texttt{Bool}$. Notably, this relation is not decidable so it cannot have the usual semantics. That is, it cannot return false when $x \geq y$ for some $x, y \in \mathbb{R}$. Instead, the semantics is as follows:

$$\mathcal{E}[\![e_1 \texttt{<} e_2]\!]\rho \triangleq \begin{cases} \lfloor \bar{1} \rfloor & \text{when } \mathcal{E}[\![e_1]\!]\rho < \mathcal{E}[\![e_2]\!]\rho \\ \lfloor \bar{0} \rfloor & \text{when } \mathcal{E}[\![e_1]\!]\rho > \mathcal{E}[\![e_2]\!]\rho \\ \bot & \text{otherwise, } \textit{i.e.}\text{, when } \mathcal{E}[\![e_1]\!]\rho = \mathcal{E}[\![e_2]\!]\rho \end{cases}$$

We can actually see that this is the correct semantics by trying to implement such a primitive. Suppose instead that we have the primitives $\texttt{<}^1 : \texttt{Real} \times \texttt{Real} \to \texttt{Unit}$ and $\texttt{>}^1 : \texttt{Real} \times \texttt{Real} \to \texttt{Unit}$, where the superscripts indicate that they are 1-sided tests. In particular, these primitives correspond to the semi-decidability of $<$ and $>$ respectively—$\mathcal{V}[\![\texttt{Unit}]\!] \cong \mathbb{S}$.

Assuming that we additionally have a parallel or operation[7], then the function $<$ can be implemented by running both $<^1$ and $>^1$ *in parallel*, matching on which computation produces a value first. The case that both $<^1$ and $>^1$ diverge is when the arguments are equal.

4.4 A Complete Partial Order Semantics

Given the motto "Turing-complete probabilistic programming languages express computable distributions," one natural attempt at giving λ_{CD} semantics would be to extend a standard CPO model of PCF. In this section, we show how to give λ_{CD} a CPO semantics. Compared to the previous semantics, we will not be able to derive all the required structure categorically. Consequently, we will need to rely an auxiliary argument to recover that environment lookup is continuous—perhaps surprisingly, this relies on an Oracle Turing machine argument, which highlights the importance of considering computability.

4.4.1 Interpretation of Types

The interpretation of types $\mathcal{V}[\![\tau]\!]$ interprets a type τ as a CPO and is defined by induction on types. As a CPO in general does not carry the same topology as a computable metric space, we will use an auxiliary type interpretation function $\mathfrak{M}[\![\cdot]\!]$ to denote the types that can be endowed with a computable metric.

We write $\mathfrak{M}[\![\tau]\!]$ to associate a well-formed type ($\vdash_D \tau$) with a computable metric space,

[7]We will discuss the issues with implementing such an operation when we implement computable distributions as a library in Haskell (Section 4.5).

defined by induction on well-formed types.

$$\mathfrak{M}[\![\texttt{Nat}]\!] \triangleq (\mathbb{N}, d_{\text{Discrete}}, \mathbb{N})$$

$$\mathfrak{M}[\![\texttt{Real}]\!] \triangleq (\mathbb{R}, d_{\text{Euclid}}, \mathbb{Q})$$

$$\mathfrak{M}[\![\tau_1 \times \tau_2]\!] \triangleq \mathfrak{M}[\![\tau_1]\!] \times \mathfrak{M}[\![\tau_2]\!]$$

The interpretation of naturals and reals are standard. The interpretation of a product $\mathfrak{M}[\![\tau_1 \times \tau_2]\!]$ forms the product of computable metric spaces $\mathfrak{M}[\![\tau_1]\!]$ and $\mathfrak{M}[\![\tau_2]\!]$.

Due to the coincidence of the CPO and topological domain interpretation of ω-continuous valuations on countably based spaces (by proposition 4.2.2), we can still use the probability monad $\mathscr{P}$ from Section 4.2 to interpet the valuation component of a distribution type in the CPO semantics. However, samplers in the CPO $2^\omega \Rightarrow_{\mathbf{CPO}} D_\perp$ are not necessarily continuous topological domain maps. Nevertheless, they are measurable.[8] Moreover, valuations can be realized by continuous sampling functions. Hence, we can restrict the interpretation of samplers as elements of the CPO $2^\omega \Rightarrow_{\mathbf{CPO}} D_\perp$ to those that are continuous.

We summarize the interpretation of types below with a call-by-name evaluation strategy.

[8] It is sufficient to show that every subset of 2^ω can be written as a countable intersection of open sets (called $G - \delta$ sets). Let A be an arbitrary subset of 2^ω and define $[X] = X$ if there is an $a \in A$ such that $a \in X$ and $[X] = \emptyset$ otherwise. Next, define a sequence $(B_n)_{n\in\mathbb{N}}$ of sets as follows: $B_0 = [2^\omega]$, $B_1 = [02^\omega] \cup [12^\omega]$, $B_2 = [002^\omega] \cup [012^\omega] \cup [102^\omega] \cup [112^\omega]$, and etc. Then, $A = \cap_{n\in\mathbb{N}} B_n$.

We use $\mathcal{S}_M$ (as opposed to $\mathcal{S}$ for the topological domain) for the CPO type of samplers.

$$\mathcal{V}[\![\texttt{Nat}]\!] \triangleq \mathrm{Disc}(\mathbb{N})_\perp$$

$$\mathcal{V}[\![\tau_1 \times \tau_2]\!] \triangleq (\mathcal{V}[\![\tau_1]\!] \times \mathcal{V}[\![\tau_2]\!])_\perp$$

$$\mathcal{V}[\![\tau_1 \to \tau_2]\!] \triangleq (\mathcal{V}[\![\tau_1]\!] \Rightarrow \mathcal{V}[\![\tau_2]\!])_\perp$$

$$\mathcal{V}[\![\texttt{Real}]\!] \triangleq \mathrm{Disc}(\mathbb{R})_\perp$$

$$\mathcal{V}[\![\texttt{Samp}\ \tau]\!] \triangleq \Sigma_{s \in \mathcal{S}_M(\mathfrak{M}[\![\tau]\!])}\, \mathrm{psh}_\tau(s)$$

The interpretation of types from the PCF-like fragment are standard. Recall that S applies the specialization preorder and L lifts a topological space. Hence, $\mathrm{Disc}(\mathbb{N})_\perp \cong S(L(\mathfrak{M}[\![\texttt{Nat}]\!]))$ and $\mathrm{Disc}(\mathbb{R})_\perp \cong S(L(\mathfrak{M}[\![\texttt{Real}]\!]))$. As before, we give both a sampling interpretation and a valuation interpretation, related by the pushforward psh_τ. Note that the use of $\mathfrak{M}[\![\cdot]\!]$ only appears in the interpretation of the distribution type.

4.4.2 Expression Denotation

The expression denotation function $\mathcal{E}[\![\Gamma \vdash e : \tau]\!]\rho : \mathcal{V}[\![\Gamma]\!] \Rightarrow \mathcal{V}[\![\tau]\!]$ is defined by induction on the typing derivation. The denotation is parameterized by a global environment Υ that interprets constants and primitive functions. As before, we enforce the same well-formedness conditions on the global environment Υ. The structure of the semantics is identical to the topological domain version (see Figure 4.2), except that all the functions we use are CPO functions this time.

To relate the sampling component with the valuation component, we will again need a push lemma.

Lemma 4.4.1 (Push 2)**.** *Let X and Y be computable metric spaces.*

1. $\mathrm{psh}(\det(x)) = \eta(x)$ *for any* $x \in X$*.*
2. *Suppose* $s : \mathcal{S}_M(X)$ *and* $f : X \to \mathcal{S}_M(Y)$ *is continuous.*[9] *Then,* $\mathrm{psh}(\mathrm{samp}(s)(f)) = \mathrm{psh}(s) \succ_\flat v \mapsto \mathrm{psh}(f(v))$.

Proof. The proof is similar to that of lemma 4.3.1. However, we need the extra condition that $f : X \to \mathcal{S}_M(Y)$ is continuous to conclude that $f : X \to \mathcal{S}_M(Y) \cong X \times 2^\omega \to Y_\perp$ is measurable. To see this, observe that $f(x, \cdot) : 2^\omega \to Y_\perp$ is measurable for every $x \in X$, and hence takes the form of a jointly measurable Caratheodory function[10]. □

Next, we show that the denotation is well-defined. However, as CPO continuous functions do not coincide in general with topological domain continuous functions, our induction hypothesis carries less structure. In particular, we will need to rely on an auxiliary argument involving Oracle Turing machines that shows that environment lookup is continuous.

Proposition 4.4.2 (Environment lookup is computable)**.** *Let* $\Gamma, x : \tau_x \vdash e : \tau$*, where* $\mathcal{V}[\![\tau_x]\!]$ *and* $\mathcal{V}[\![\tau]\!]$ *both denote represented spaces. Then,*

$$\bar{x} \mapsto \mathcal{E}[\![e]\!]\rho[x \mapsto \bar{x}]$$

is computable as a map between represented spaces.

Intuitively, the proposition holds because environment lookup is computable. Hence, it is continuous. Put another way, we can treat an environment lookup of the variable x as Turing

[9] An earlier version of this work missed the continuity condition.

[10] Let Z be a measurable space and let X and Y be topological spaces. A *Caratheodory* function is a function $f : X \times Z \to Y$ such that (1) $f(x, \cdot)$ is measurable for any $x \in X$ and (2) $f(\cdot, z)$ is continuous for any $z \in Z$. Caretheodory functions are jointly measurable.

machine code for: "look at the tape named by x". For now, we will walkthrough the argument that the denotation is well-defined first and then comment on proposition 4.4.2.

Lemma 4.4.3 (Denotation well-defined)**.** *If* $\Gamma \vdash e : \tau$*, then* $\mathcal{E}[\![e]\!] : \mathcal{V}[\![\Gamma]\!] \Rightarrow \mathcal{V}[\![\tau]\!]$.

Proof. By induction on the typing derivation under the stronger induction hypothesis that the samplers are continuous (as represented maps). We check the cases corresponding to $\Gamma \vdash \texttt{return}\ e : \texttt{Samp}\ \tau$ and $\Gamma \vdash x \leftarrow e_1\ ;\ e_2 : \texttt{Samp}\ \tau_2$.

Case 4 ($\Gamma \vdash \texttt{return}\ e : \texttt{Samp}\ \tau$). The definition of the sampler and valuation are clearly well-defined. Applying lemma 4.4.1 completes this case.

Case 5 ($\Gamma \vdash x \leftarrow e_1\ ;\ e_2 : \texttt{Samp}\ \tau_2$). For this case, it is productive to address the sampler and the valuation separately, and then relate the two.

The denotation of the sampling component is $\mathrm{samp}(\pi_1(\mathcal{E}[\![e_1]\!]\rho))(\bar{x} \mapsto \pi_1(\mathcal{E}[\![e_2]\!]\rho[x \mapsto \bar{x}]))$. By the induction hypothesis, $\mathcal{E}[\![e_1]\!]\rho$ is continuous and $\mathcal{E}[\![e_2]\!]\rho[x \mapsto \bar{x}]$ is continuous for any $\bar{x}$. Moreover, by Proposition 4.4.2, the function $\mathcal{E}[\![e_2]\!]\rho[x \mapsto \bar{x}]$ is continuous. Hence, the result is continuous by continuity of all the operations involved.

The denotation of the valuation component is $\pi_2(\mathcal{E}[\![e_1]\!]\rho) \succ_\flat \bar{x} \mapsto \pi_2(\mathcal{E}[\![e_2]\!]\rho[x \mapsto \bar{x}])$. By the induction hypothesis, $\pi_2(\mathcal{E}[\![e_1]\!]\rho)$ is an ω-continuous valuation. Also by the induction hypothesis and Proposition 4.4.2, we have that $\bar{x} \mapsto \pi_2(\mathcal{E}[\![e_2]\!]\rho[x \mapsto \bar{x}]) : L(\mathfrak{M}[\![\tau_1]\!]) \Rightarrow_{\mathbf{CPO}} \mathcal{V}[\![\texttt{Samp}\ \tau_2]\!]$ is a continuous map between represented spaces (*i.e.*, is an element of $\mathscr{C}(\mathcal{V}[\![\tau_1]\!], \mathscr{C}(\mathcal{O}(\mathcal{V}[\![\tau_2]\!]), [0,1]_<))$). Hence, we can apply $\succ_\flat$ and the denotation of the valuation component is well-defined.

Finally, we need to show that the valuation is the pushforward along the sampler. Note that the interpretation of sampling type $L(\mathfrak{M}[\![\tau]\!])$ is not a computable metric space because it

is lifted. Observe that

$$\int f \, d\mu = \int f|_{\mathcal{O}(X)} \, d\mu|_{\mathcal{O}(X)} + f(\bot)\mu(\{\bot\}) \, ,$$

where $f : L(X) \to [0,1]_<$ and $\mu \in \mathscr{P}(L(X))$. Hence, the result follows by the lemma 4.4.1 and linearity of the integral.

□

The proof relies on an environment lookup argument for Oracle Turing machines. To make this precise, we refer the reader to Appendix A. For now, we will sketch out the main idea.

Sketch 3. At a high-level, we need to show that the environment lookup map has a realizer. Intuitively, if environments are represented as a collection of tapes indexed by variable names, then a variable x can be translated as Turing machine code for: "read from the tape indexed by x". Indeed, the execution of λ_{CD} can be simulated on an Oracle Turing machine—the PCF-like fragment can be simulated using a standard translation of lambda terms to their encodings on Turing machine tapes, while the constructs λ_{CD} introduces are all realizable by an Oracle Turing machine by design. However, we still need to connect the Turing machine level view to the denotational semantics. Towards this end, we can show (1) that the denotational semantics is related to an operational semantics (via soundness and adequacy) and that (2) the operational semantics can be simulated by an Oracle Turing machine. Combining the two makes the connection between the denotational semantics and Oracle Turing machines precise.

```
module ALib (CMetrizable(..), approx, anth) where

approx :: (Nat -> a) -> A a     -- fast Cauchy sequence
anth :: A a -> Nat -> a         -- project n-th approx.

class CMetrizable a where
    enum :: [a]                 -- countable, dense subset
    metric :: a -> a -> A Rat   -- computable metric

newtype A a = A { getA :: Nat -> a }



module CDistLib (RandBits, Samp(..), asampler) where

import ALib

type RandBits = Nat -> Bool
newtype Samp a = Samp { getSamp :: RandBits -> a }

asampler :: (CMetrizable a) => (RandBits -> A a) -> Samp (A a)

instance Monad Samp where
    ...
```

Figure 4.3: A Haskell library interface for λ_{CD}.

4.5 Computable Distributions as a Library

We now present a Haskell library (Figure 4.3) for expressing samplers that implements the sampling semantics. In particular, the Haskell library does not assume reals or blackbox continuous distributions.

The module `ALib` encapsulates elements of a computable metric spaces. More concretely, the type `A` τ models an element of a computable metric space can be read as an approximation by a sequence of values of type τ. For example, a computable real can be given the type `CReal` $\triangleq$ `A Rat`, meaning it is a sequence of rationals that converges to a real. We form val-

ues of type `A` τ using `approx`, which requires us to check that the function we are coercing describes a fast Cauchy sequence, and project out approximations using `anth`.

To form `A` τ, values of type τ should support the operations required of a computable metric space. We can indicate the required operations using Haskell's type-class mechanism.

```
class CMetrizable a where
    enum :: [a]
    metric :: a -> a -> A Rat
```

When we implement an instance of `CMetrizable` τ, we should check that the implementation of `enum` enumerates a dense subset and `metric` computes a metric as a computable metric space requires (see Section 2.2.2). Below, we give an instance of `A Rat` for computable reals.

```
instance CMetrizable Rat where
    enum = 0 : [ toRational m / 2^n
               | n <- [1..]
               , m <- [-2^n * n..2^n * n]
               , odd m || abs m > 2^n * (n-1) ]
    metric x y = A (\_ -> abs (x - y))
```

This instance enumerates the dyadic rationals (powers of 2), which are a dense subset of the reals. Note that there are many other choices here for the dense enumeration.[11] In this instance, we can actually compute the metric as a dyadic rational, whereas a computable metric requires the weaker condition that we can compute the metric as a computable real.

Next, we can use the module `ALib` to implement computable operations on commonly used types. This reifies the computable primitives $\mathrm{op}^r(\overrightarrow{e})$ from the core language as a library function. For example, a library for computable reals will contain the `CMetrizable` τ instance implementation above and other computable functions. However, some operations are not re-

[11] Algorithms that operate on computable metric spaces compute by enumeration so the algorithm is sensitive to the choice of enumeration.

alizable (*e.g.*, equality of reals) and so this module does not contain all operations one may want to perform on reals.

```
module RealLib (Real, pi, (+), ...) where
import ALib

type Real = A Rat
instance CMetrizable Rat where
    ...

pi :: Real
(+) :: Real -> Real -> Real
-- etc.
```

The module `CompDistLib` contains the implementation of distributions. A sampler `Samp` α is a function from a bit-stream (*i.e.*, `RandBits` represented isomorphically as `Nat -> Bool` instead of `[Bool]`) to values of type α.

```
type RandBits = Nat -> Bool
newtype Samp a = Samp { getSamp :: RandBits -> a }
```

We can implement an instance of the sampling monad by essentially translating det and samp from our semantics into Haskell.

```
instance Monad Samp where
    return x = Samp (const x)
    (>>=) s f = Samp ((uncurry (getSamp . f)) . (pair (getSamp s
        . fst) snd) . split)
        where pair f g = \x -> (f x, g x)
              split = pair even odd
              even u = (\n -> u (2 * n))
              odd u = (\n -> u (2 * n + 1))
```

As expected, `return` corresponds to a sampler that ignores its input randomness and `>>=` corresponds to a composition of samplers. The module `CompDistLib` provides the function `sampler` to coerce an arbitrary Haskell function of the appropriate type into a value of type `Samp` α.

```
asampler :: (CMetrizable a) => (RandBits -> A a) -> Samp  (A a)
asampler = Samp
```

We should call `sampler` only on sampling functions realizing Type-2 computable sampling algorithms.

This concludes the implementation of the module `CompDistLib`, and thus, a Haskell library for expressing computable distributions. We emphasize that it does not rely on any blackbox primitives.[12]

4.6 Examples

In this section, we will use the Haskell library to give examples of expressible distributions. We will use the expression denotation function $\mathcal{E}[\![\cdot]\!]\rho$ defined previously (both the topological domain and CPO version work) to check the denotations of our programs which implement samplers. For the most part, will drop the sampling component of the denotation and concern ourselves with just the valuation component.

4.6.1 Discrete Distribution

Consider the encoding of a geometric distribution with bias 1/2, which returns the number of fair Bernoulli trials until a success. The distribution `std_bernoulli` denotes a Bernoulli distribution with bias 1/2.

```
std_geometric :: Samp Nat
std_geometric = do
  b <- std_bernoulli
  if b then return 1
       else std_geometric >>= return . (\n -> n + 1)
```

[12] However, we will need a unique exception to compute the modulus of continuity (Section 4.7.2).

Let μ_B be an unbiased Bernoulli distribution and μ^n correspond to n un-foldings of `std_geometric`.

$$\mathcal{E}[\![\texttt{std\_geometric}]\!]\rho(U) = \sup_n \int v \mapsto \left(\begin{cases} \mathbb{1}_U(1) & \text{when } v = \text{true} \\ \int w \mapsto \mathbb{1}_U(w+1) d\mu^n & \text{otherwise} \end{cases} \right) d\mu_B$$

$$= \sup_{n \in \mathbb{N}} (\mathbb{1}_U(1)\frac{1}{2} + \sum_{w=0}^{\infty} \mathbb{1}_U(w+1)\mu^n(\{w\}))$$

$$= \mathbb{1}_U(1)\frac{1}{2} + \sum_{w=0}^{\infty} \mathbb{1}_U(w+1)(\sup_{n \in \mathbb{N}} \mu^n(\{w\}))$$

By induction on n, we can show that μ^n is the measure

$$\mu^n = \{0\} \mapsto 0, \{1\} \mapsto (1/2), \ldots, \{n\} \mapsto (1/2)^n \ .$$

Hence, we can conclude that $\sup_{n \in \mathbb{N}} \mu^n$ is a geometric distribution and that $\mathcal{E}[\![\texttt{std\_geometric}]\!]\rho$ is a Geometric distribution.

4.6.2 Continuous Distributions

We fill in the sketch of the standard uniform distribution we presented in the background.

As a reminder, we need to convert a random bit-stream into a sequence of (dyadic) rational approximations.

```
std_uniform :: Samp Real
std_uniform = asampler (\u -> approx (\n -> bisect (n+1) u 0 1 0)
   )
    where
      bisect n u (l :: Rat) (r :: Rat) m
          | m < n && u m       =
              bisect n u l (midpt l r) (m+1)
          | m < n && not (u m) =
              bisect n u (midpt l r) r (m+1)
```

```
    | otherwise          =
        midpt l r
midpt l r = l + (r - l) / 2
```

The function `bisect` repeatedly bisects an interval specified by $(\mathtt{l}, \mathtt{r})$. By construction, the sampler produces a sequence of dyadic rationals. We can see that this sampling function is uniformly distributed because it inverts the binary expansion specified by the uniformly distributed input bit-stream. Once we have the standard uniform distribution, we can encode other primitive distributions (*e.g.*, normal, exponential, etc.) as transformations of the uniform distribution as in standard statistics using return and bind.

For example, we give an encoding of the standard normal distribution using the Marsaglia polar transformation, which diverges with probability 0:

```
std_normal :: Samp Real
std_normal = do
  u1 <- uniform (-1) 1
  u2 <- uniform (-1) 1
  let s = u1 * u1 + u2 * u2
  if s < 1 then return (u1 * sqrt (log s / s))
           else std_normal
```

The distribution `uniform` (-1) 1 is the uniform distribution on the interval $(-1, 1)$ and can be encoded by shifting and scaling a draw from `std_uniform`. We can check that this distribution produces a sample with probability 1 by showing that both $s = 1$ (by absolute continuity) and divergence (by Borel-Cantelli) occur with probability 0. Note that the operation `<` semi-decides both $<$ and $>$, where we have that equality does not hold with probability 1.

4.6.3 Singular Distribution

Next, we give an encoding of the Cantor distribution. The Cantor distribution is singular so it is not a mixture of a discrete component and a component with a density. The distribution

can be defined recursively. It starts by trisecting the unit interval, and placing half the mass on the leftmost interval and the other half on the rightmost interval, leaving no mass for the middle, continuing in the same manner with each remaining interval that has positive probability. We can encode the Cantor distribution by directly transforming a random bit-stream into a sequence of approximations.

```
cantor :: Samp Real
cantor = asampler (\u -> approx (\n -> go u 0 1 0 n))
    where
      go u (left :: Rat) (right :: Rat) n m
        | n < m && u n       =
            go u left (left + pow) (n + 1) m
        | n < m && not (u n) =
            go u (right - pow) right (n + 1) m
        | otherwise          =
            right - (1 / 2) * pow
          where pow = 3 ^^ (-n)
```

The sampling algorithm keeps track of which interval it is currently in specified by `left` and `right`. If the current bit is 1, we trisect the left interval. Otherwise, we trisect the rightmost interval. Crucially, the number of trisections is bounded by the precision we would like to generate the sample to. We could express the Cantor distribution in a measure-theoretic language with recursion, but we would need to trisect infinitely to express the distribution exactly.

4.6.4 Distributions and Partiality

We investigate the semantics of divergence more closely now. Consider the two expressions below.

```
bot_samp :: (CMetrizable a) => Samp (A a)
bot_samp = bot_samp
```

```
bot_samp_bot :: (CMetrizable a) => Samp (A a)
bot_samp_bot = asampler (\_ -> bot)
    where bot = bot
```

In the former, we obtain the bottom valuation, which assigns 0 mass to every open set. This corresponds to the sampling function $u \in 2^\omega \mapsto \bot$ and can be interpreted as failing to provide a sampler. In the latter, we obtain the valuation that assigns 0 mass to every open set, except for the set $\{\lfloor X \rfloor \cup \bot\}$ which is assigned mass 1. This corresponds to the sampling function $u \in 2^\omega \mapsto \lfloor \bot \rfloor$ and can be interpreted as providing a sampling function that fails to produce a sample.

To further illustrate the distinction in semantics between the two expression above, consider sampling from `bot_samp` and `bot_samp_bot` respectively, but ignoring the result as below. This will highlight the laziness of the semantics.

```
always_div :: Samp Real
always_div = do
  _ <- bot_samp :: Samp Real
  std_uniform

never_div :: Samp Real
never_div = do
  _ <- bot_samp_bot :: Samp Real
  std_uniform
```

We can check that the denotation of the former is equivalent to that of `bot_samp`.

$$\begin{aligned}\mathcal{E}[\![\texttt{always\_div}]\!]\rho(U) &= \int v \mapsto \mu_{\mathcal{U}(0,1)}(U)\, d\mathcal{E}[\![\texttt{bot\_samp}]\!]\rho \\ &= 0\end{aligned}$$

Note that $\mathcal{E}[\![\texttt{bot\_samp}]\!]\rho$ maps every open set to 0 so the integral is 0 as well. However, the denotation of the latter is equivalent to that of `std_uniform`.

$$\begin{aligned}
\mathcal{E}[\![\texttt{never_div}]\!]\rho(U) &= \int v \mapsto \mu_{\mathcal{U}(0,1)}(U)\, d\mathcal{E}[\![\texttt{bot_samp_bot}]\!]\rho \\
&= \sup\{\int s\, d\mathcal{E}[\![\texttt{bot_samp_bot}]\!]\rho \mid s \leq v \mapsto \mu_{\mathcal{U}(0,1)}(U), s \text{ simple}\} \\
&= \mu_{\mathcal{U}(0,1)}(U)\, \mathcal{E}[\![\texttt{bot_samp_bot}(\{\lfloor X \rfloor \cup \bot\})]\!]\rho \\
&= \mu_{\mathcal{U}(0,1)}(U)
\end{aligned}$$

As a reminder, $\mathcal{E}[\![\texttt{bot\_samp\_bot}]\!]\rho$ has $\{\lfloor \mathbb{R} \rfloor \cup \bot\} \mapsto 1$. Hence, the integral takes its largest value on the simple function $\mu_{\mathcal{U}(0,1)}(U)\, \mathbb{1}_{\{\lfloor \mathbb{R} \rfloor \cup \bot\}}(\cdot) \leq v \mapsto \mu_{\mathcal{U}(0,1)}(U)$ for any open U.

As a final example, consider the program below that uses a coin flip to determine its diverging behavior.

```
maybe_bot :: Samp Bool
maybe_bot = do
  b <- bernoulli
  if b then return bot else bernoulli
```

Intuitively, this distribution returns a *sampler that always generates diverging samples* with probability 1/2 and returns an unbiased Bernoulli distribution with probability 1/2. If we changed `return bot` to `bot_samp` as below

```
maybe_bot' :: Samp Bool
maybe_bot' = do
  b <- bernoulli
  if b then bot else bernoulli
```

then the semantics would change to a distribution that returns a *diverging sampler* with probability 1/2 and an unbiased Bernoulli distribution with probability 1/2.

4.6.5 Non-parametric Prior

We give two different encodings of the Dirichlet process, a prior distribution used in mixture models where the number of mixtures is unknown (*e.g.*, [70]). The Dirichlet process $\mathrm{DP}(\alpha, G_0)$ is a distribution on distributions—a draw produces a discrete distribution with support determined by G_0, the *base distribution*, and mass according to α, the *concentration parameter*. The Dirichlet process can be represented in multiple ways, where each representation illuminates different properties. One representation is called the Blackwell-MacQueen urn scheme (see [70]), which describes how to sample from the distribution resulting from a draw of the Dirichlet process. Thus, we can imagine it describing the following process:

$$G \sim \mathrm{DP}(\alpha, G_0)$$

$$\theta_n \mid G \sim G \text{ for } n \in \mathbb{N} \,.$$

The conditional distribution of θ_n is

$$\theta_n \mid \theta_{1:n-1} \sim \frac{\alpha}{\alpha + n - 1} G_0 + \frac{1}{\alpha + n - 1} \sum_{j=1}^{n-1} 1_{\theta_j} \text{ for } n \in \mathbb{N} \,,$$

which shows that the base distribution G_0 determines the support and α determines how often we select a new point from G_0 to put mass on. We can encode the conditional distribution in λ_{CD}.

```
urn' :: CReal -> Samp a -> [a] -> Samp a
urn' alpha g0 prev =
    let l = length prev
        n :: Integer = (toInteger l) + 1
        w = 1 / (fromInteger n - 1 + alpha)
```

```
    ws = replicate l w ++ [alpha]
    d = disc_id ws
in do
  c <- d
  if c == n - 1
  then g0
  else return (prev !! (fromInteger c))
```

We can put our reasoning principles to work to argue that `urn'` encodes the conditional distributions. First, we can use a distributional view of the monadic block of `urn'` under an environment ρ, where $\mu = \mathcal{E}[\![\texttt{d}]\!]\rho$.

$$\mathcal{E}[\![\texttt{c} \leftarrow \texttt{d}\ ;\ \texttt{if0 c == n-1 then g0 else return (prev !! c)}]\!]\rho = \lambda U.$$

$$\int v \mapsto \pi_1 \circ \begin{cases} \mathcal{E}[\![\texttt{g0}]\!]\rho[c \mapsto v](U) & \text{if } \mathcal{E}[\![\texttt{c == n-1}]\!]\rho[c \mapsto v] \\ \mathcal{E}[\![\texttt{return (prev !! c)}]\!]\rho[c \mapsto v](U) & \text{otherwise} \end{cases} d\mu$$

Next, substituting away the `let` bindings (justified by Haskell semantics) implies that $\mathcal{E}[\![d]\!]\rho$ is the discrete distribution

$$0 \mapsto \frac{1}{\alpha + n - 1}, \ldots, n-2 \mapsto \frac{1}{\alpha + n - 1}, n-1 \mapsto \frac{\alpha}{\alpha + n - 1}\ .$$

This reduces the previous integral to the summation

$$U \mapsto \sum_{j=0}^{n-2} \frac{1}{\alpha + n - 1} \mathcal{E}[\![\texttt{return (prev !! c)}]\!]\rho[c \mapsto j](U) + \frac{\alpha}{\alpha + n - 1} \mathcal{E}[\![\texttt{g0}]\!]\rho[c \mapsto n-1](U)\ ,$$

where $\mathcal{E}[\![\texttt{g0}]\!]\rho$ is the base distribution G_0. Rewriting this in statistical notation gives the desired result

$$\mathcal{E}[\![\texttt{urn' alpha g0 prev}]\!]\rho \sim \frac{\alpha}{\alpha+n-1}G_0 + \frac{\sum_{j=1}^{n-1}}{\alpha+n-1}1_{\texttt{prev}_j} \,.$$

Next, we can describe the entire infinite sequence using lazy monadic lists

```
data MList m a = Nil | Cons a (m (MList m a))
```

and analogoulsy define common operations expected of lists such as `iterate`, `map`, and `tail`.

```
urn :: forall a. CReal -> Samp a -> Samp (MList a)
urn alpha g0 =
    let f :: ((a, [a]) -> Samp a) = return . fst)
        g (_, acc) = do
          x <- urn' alpha g0 acc
          return (x, acc ++ [x])
    in do
      x0 <- g0
      xs <- ML.map f (ML.iterate g (return (x0, [])))
      ML.tail xs
```

Expressing the resulting conditional distribution for each n gives

$$\mathcal{E}[\![\texttt{urn alpha g0}]\!]\rho_n \mid \mathcal{E}[\![\texttt{urn alpha g0}]\!]\rho_{1:n-1} \sim \frac{\alpha}{\alpha+n-1}G_0 + \frac{\sum_{j=1}^{n-1}}{\alpha+n-1}1_{\text{psh}\,\mathcal{E}[\![\texttt{urn alpha g0}]\!]\rho_j} \,.$$

Alternatively, there is a constructive representation known as the stick-breaking construction (see [70]) that gives the structure of the discrete distribution directly. We describe a process that gives $G \sim \text{DP}(\alpha, G_0)$. First, let random variables $\beta_k \sim \text{Beta}(1, \alpha)$ be distributed according to the beta distribution for $k \in \mathbb{N}$. Next, define $\pi_k = \beta_k \prod_{i=1}^{k-1}(1-\beta_i)$ for $k \in \mathbb{N}$. Let $X_k \sim G_0$ for $k \in \mathbb{N}$. The result G is $\sum_{k=1}^{\infty} \pi_k \mathbb{1}_{X_k}(\cdot)$. We can encode the stick-breaking con-

struction in λ_{CD}, where `ML.@:` is synonymously `ML.Cons` and `ML.!!!` indexes a lazy monadic list. The function `mdisc_id` samples an index according to an input lazy monad list specifying probabilities.

```
sticks :: CReal -> Samp a -> Samp a
sticks alpha g0 = do
  xs <- ML.repeat g0
  pis <- weights 1
  c <- mdisc_id pis
  xs ML.!!! fromInteger c
    where
      weights :: CReal -> Samp (MList CReal)
      weights left = do
        v <- beta 1 alpha
        return ((v * left) ML.@: (weights (left * (1 - v))))
```

We can follow a similar pattern to reason about the urn representation. For instance, we can analyze this function compositionally as before by reasoning that $c \leftarrow$ `mdisc_id pis xs` ; `ML.!!! fromInteger c` selects a sample from `xs` according to the weights `pis`. Finally, we can combine this with showing that `weights` generates the weights π_k.

The encodings show that we can use probabilistic and standard program reasoning principles at the same time. Because we checked that each program encoded their respective representation, we also obtain that sampling `sticks` an infinite number of times is equivalent to `urn` because both encode the Dirichlet process. This might seem strange because the urn encoding has more sequential dependencides than the stick-breaking representation. The equivalence relies on a probabilistic concept called *exchangeability*, which asserts the existence of a conditionally independent representation if the distribution is invariant under all finite permutations. Exchangeability has been studied in the Type-2 setting [31] and it would be interesting to see if we can lift those results into λ_{CD}.

4.7 Conditioning

Conditioning is the core operation of Bayesian inference. As we alluded to earlier, conditioning is not computable in general [5]. We could use a more abstract definition of conditioning used in measure theory, but it would be undesirable if the semantics of conditioning for a probabilistic programming language was not computable given that one of its goals is to automate inference. Instead, we take a library approach, which requires the client to provide an implementation of a conditioning algorithm and limits us to situations where conditioning is computable. Hence, the semantics of our core language remains unchanged.

4.7.1 Preliminaries

We give the computable form of conditioning (see Chapter 2 for background on conditioning).

Definition 4.7.1 (Computable probability kernel [5, Def. 4.2])**.** Let S and T be computable metric spaces, and $\mathcal{B}_T$ be the σ-algebra on T. A probability kernel $\kappa : S \times \mathcal{B}_T \to [0,1]$ is computable if $\kappa(\cdot, A)$ is a lower semi-computable function for every r.e. open $A \in \sigma(\mathcal{B}_T)$.

Definition 4.7.2 (Computable conditional distribution [5, Def. 4.7])**.** Let X and Y be random variables in computable metric spaces S and T. Let κ be a version (Section 2.3.2) of $\mathbb{P}[Y \mid X]$ (notation for $\mathbb{P}[Y \in \cdot \mid X]$). Then $\mathbb{P}[Y \mid X]$ is computable if κ is computable on a $\mathbb{P}_X$ measure-one subset.

Thus, a non-computable conditional distribution is one for which *every* version is non-computable.

```
module CondLib (BndDens, obsDens) where
import ALib
import CompDistLib
import RealLib

newtype BndDens a b =
    BndDens { getBndCondDens :: (A a -> A b -> Real, Rat) }

-- Requires comp. dist. and bounded conditional density
obsDens :: forall u v y.
    (CMetrizable u, CMetrizable v, CMetrizable y) =>
    Samp (A (u, v)) -> BndDens u y -> A y -> Samp (A (u, v))
```

Figure 4.4: An interface for conditioning.

4.7.2 Conditioning as a Library

Now, we add conditioning as a library to λ_{CD} (Fig. 4.4). λ_{CD} provides only a restricted conditioning operation `obs_dens`, which requires a conditional density. We will see that the computability of `obs_dens` corresponds to an effective version of Bayes' rule. We have given only one conditioning primitive here, but it is possible to identify other situations where conditioning is computable and add those to the conditioning library. For example, conditioning on positive probability events is computable (see [33, Prop. 3.1.2]).

The library provides the conditioning operation `obs_dens`, which enables us to condition on *continuous*-valued data when a bounded and computable conditional density is available.

Proposition 4.7.1. *[5, Cor. 8.8] Let U, V and Y be computable random variables, where Y is independent of V given U. Let $p_{Y|U}(y \mid u)$ be a conditional density of Y given U that is bounded and computable.*[13] *Then the conditional distribution $\mathbb{P}[(U,V) \mid Y]$ is computable.*

[13] Note that $p_{Y|(U,V)}(y \mid u,v) = p_{Y|U}(y \mid u)$ due to the conditional independence of Y and V given U.

The bounded and computable conditional density enables the following integral to be computed, which is in essence Bayes' rule. A version of the conditional distribution $\mathbb{P}((U,V) \mid Y)$ is

$$\kappa_{(U,V)|Y}(y, B) = \frac{\int_B p_{Y|U}(y \mid u)\, d\mathbb{P}_{(U,V)}}{\int p_{Y|U}(y \mid u)\, d\mathbb{P}_{(U,V)}}$$

where B is a Borel set in the space associated with $U \times V$.[14]

Another interpretation of the restricted situation is that our observations have been corrupted by independent smooth noise [5, Cor. 8.9]. To see this, let U be the random variable corresponding to our ideal model of how the data was generated, V be the random variable corresponding to the model parameters, and Y be the random variable corresponding to the corrupted data we observe. Notice that the model (U, V) is not required to have a density and can be an arbitrary computable distribution. Indeed, probabilistic programming systems proposed by the machine learning community impose a similar restriction (*e.g.*, see [39, 105]).

Now, we describe `obs_dens`, starting with its type signature. Let the type `BndDens` τ σ represent a bounded computable density:

```
newtype BndDens a b =
    BndDens { getBndCondDens :: (A a -> A b -> Real, Rat) }
```

Conditioning thus takes a samplable distribution, a bounded computable density describing how observations have been corrupted, and returns a samplable distribution representing the conditional. In the context of Bayesian inference, it does not make sense to condition distributions such as `maybe_bot` that diverge with positive probability. Hence, we do not give semantics to conditioning on those distributions. Again, practical probabilistic programming

[14] As a reminder, $p_{Y|(U,V)}(y \mid u, v) = p_{Y|U}(y \mid u)$ due to the conditional independence of Y and V given U. Hence, the conditional density $p_{Y|U}(y \mid u)$ in the integral written more precisely is $(u, v) \mapsto p_{Y|U}(y \mid u)$.

systems enforce a similar restriction (*e.g.*, see [39, 105]).

The implementation of `obs_dens` is in essence a λ_{CD} program that implements the proof that conditioning is computable in this restricted setting. This is possible because results in computability theory have computable realizers.[15]

```
obsDens :: forall u v y.
    (CMetrizable u, CMetrizable v, CMetrizable y) =>
    Samp (A (u, v)) -> BndDens u y -> A y -> Samp (A (u, v))
obsDens dist (BndDens (dens, bnd)) d =
    let f :: A (u, v) -> CReal = \x -> dens (afst x) d
        mu :: Prob (u, v) = stc dist
        nu :: Prob (u, v) = \bs ->
              let num  = integrate_bnd_dom mu f bnd bs
                  denom = integrate_bnd mu f bnd
              in map fst (cauchy_to_lu (num / denom))
    in
      cts nu
```

The parameter `dist` corresponds to the joint distribution of the model (both model parameters and likelihood), `dens` corresponds to a bounded conditional density describing how observation of data has been corrupted by independent noise, and `d` is the observed data. Next, we informally describe the undefined functions in the sketch. The function `afst` projects out the first component of a product of approximations. The functions `stc` and `cts` witness the computable isomorphism between samplable and computable distributions.[16] The functions `integrate_bnd_dom` and `integrate_bnd` compute an integral (see [46, Prop. 4.3.1]), and correspond to an effective Lebesgue integral. `cauchy_to_lu` converts a Cauchy description of a

[15] That is, we implement the Type-2 machine code as a Haskell program. The implementation relies on Haskell's imprecise exceptions mechanism [77] to express the modulus of continuity of a computable function. See Andrej Bauer's blog (http://math.andrej.com/2006/03/27/sometimes-all-functions-are-continuous) and also Remark 4.7.2.

[16] The computable isomorphism relies on the distributions being full-measure. The algorithm is undefined otherwise.

computable real into an enumeration of lower and upper bounds.

Because `obs_dens` works with conditional densities, we do not need to worry about the Borel paradox. The Borel paradox shows that we can obtain different conditional distributions by conditioning on equivalent probability zero events [84].

In addition, note that it is not possible to create a boolean value that distinguishes two probability zero events in λ_{CD}. For instance, the operator `==` implementing equality on reals returns false if two reals are provably not-equal and diverges otherwise because equality is not decidable.

To end, we give an encoding in λ_{CD} of an example by Ackerman *et al.* [5] that shows that conditioning is not always computable. Similar to other results in computability theory, the example demonstrates that an algorithm computing the conditional distribution would also solve the Halting problem.

```
non_comp :: Samp (Nat, Real)
non_comp = do
  n <- geometric (1/2)
  c <- bernoulli (1/3)
  u <- uniform 0 1
  v <- uniform 0 1
  x <- return (approx (\k -> dk k (tm_halts_within_k n k)))
  return (n, x)
    where dk k m | m > k = anth v k
                 | m = k = c
                 | m < k = anth u (k - m - 1)
```

The function `tm_halts_within_k` accepts a natural n specifying the n-th Turing machine and a natural k describing the number of steps to run the machine for, and returns the number of steps the n-th Turing machine halts in or k if it cannot tell. Upon inspection, we see the function `dk` produces the binary expansion (as a dyadic rational) of a real, using `tm_halts_within_k` to select different bits of the binary expansion of u or v, or the bit c. Thus, `tm_halts_within_k`

is computable because we can enumerate those Turing machines that halt within k steps.

Intuitively, computing the conditional distribution of a distribution encoded as a program corresponds to running it backwards. For example, computing the conditional distribution $P(\mathbf{N} \mid \mathbf{X})$, where the random variable $\mathbf{N}$ corresponds to the program variable n and $\mathbf{X}$ to x, would require us to compute the complement of `tm_halts_within_k`. Of course, we cannot enumerate the complement of the Halting set, so `non_comp` encodes a computable distribution whose conditional is not computable. We refer the reader to the full proof in Ackerman *et al.*'s original paper [5].

Remark (Encoding). In this section, we describe in more detail the implementation of the library interface. The implementation is a proof of concept that shows we can realize the interface by coding results and proofs about computable distributions as Haskell code, as opposed to solely relying on the written descriptions of Type-2 machine code. Currently, we have only informally argued that the code correctly encodes the corresponding proof.

One computable function we need to encode is the *modulus* of a computable function between computable metric spaces. The modulus $g : (X \to Y) \to \mathbb{N} \to \mathbb{N}$ of a computable function $f : X \to Y$ between computable metric spaces $(X, d_X, \mathcal{S}_X)$ and $(Y, d_Y, \mathcal{S}_Y)$ is a function that computes the number of input approximations consumed to produce an output approximation to a specified precision. For example, if the algorithm realizing f looks at $s^X_{i_0}, \ldots, s^X_{i_{41}}$ to compute an output s_{i_n} such that $d_Y(s^Y_{i_n}, f(x)) < 2^{-(n+1)}$ and $(s^X_{i_m})_{m \in \mathbb{N}} \to x$, then the modulus $g(f)(n)$ is 42.

We use Haskell's imprecise exceptions mechanism [77], an impure feature, in a restricted manner to express the modulus.[17] Hence, we can raise exceptions in pure code but only catch

[17] See Andrej Bauer's blog http://math.andrej.com/2006/03/27/sometimes-all-functions-are-

them in the IO monad.

```
modulus :: (A a -> A b) -> A a -> Nat -> IO Nat
modulus f x n = do
  e :: Either E.ErrorCall b <- E.try (E.evaluate $ anth (f x) n)
  case e of
    Left _ -> E.throw E.NonTermination
    Right _ -> search 0
    where
      trunc :: A a -> Nat -> A a
      trunc x n = approx (\m -> if m < n
                                then (anth x m)
                                else unique_fail)
      search :: Nat -> IO Nat
      search acc = do
            e :: Either E.ErrorCall b <- E.try $ E.evaluate $
                 anth (f (trunc x acc)) n
            case e of
              Left _ -> search (acc + 1)
              Right _ -> return acc
```

The function truncate cuts off an infinite input at some level by calling E.throw E.NonTermination to throw an unique exception if it is accessed. The function search starts at providing access to 0 elements in the input sequence and attempts to compute the function until an exception is not thrown. The first such number to not trigger an exception is the amount of input required for that particular output. Catching the exception forces the computation into the IO monad. To ensure that the unique exception from truncate is caught, we first attempt to evaluate f on the bare value x to ensure that no exceptions are thrown. Note that the modulus of two extensionally equivalent functions may not be equivalent.

Computations that use the modulus will be forced into the IO monad. To avoid polluting the type of any library function that uses the modulus we use unsafePerformIO. Informally, this is safe because computing the modulus of the same function in different contexts

---

continuous

(or world in Haskell terminology) produces equivalent results.

4.8 Realizability

Before we overview some of the related work on semantics, we informally discuss realizability and possible connections to our work.[18]

For the purposes of this dissertation, realizability is of particular interest because it provides an alternative approach to Type-2 computability.[19] Hence, it is quite likely that we can give semantics to probabilistic modeling languages based on realizability, and moreover, that such semantics are connected with the semantics presented in this chapter. We refer the interested reader to Bauer's thesis [14] and Leitz's thesis [57] (and the references within) for background on realizability. In the rest of this section, we gather the relevant definitions and ideas following Streicher's notes [91], and informally sketch out the possible connections with our semantics.

The starting structure for realizability is known as a *partial combinatory algebra* (pca), which provides an abstraction of an untyped model of computation. Hence, it is similar to the Type-2 machine that grounds Type-2 computability (Section 2.2.1). As we will see, a Type-2 machine can be seen as an instance of a particular pca. Rather than provide the standard definition of a pca, we give an equivalent characterization of a pca which makes its connection to (untyped) lambda calculus more apparent. For this, we will need some auxilliary definitions first.

[18] I would like to thank Bas Spitters and Lars Birkedal for pointing out and discussing the possible connections with the semantics presented in this chapter.

[19] In fact, realizability pre-dates Type-2 computability.

Definition 4.8.1.

- An *applicative structure* is a tuple $\mathcal{A} = (|\mathcal{A}|, \cdot)$, where $|\mathcal{A}|$ is a set and the (partial) binary operation $\cdot : |\mathcal{A}| \times |\mathcal{A}| \rightharpoonup |\mathcal{A}|$. For readability, we use infix notation for $\cdot$.
- A *polynomial* over an applicative structure $\mathcal{A}$ is a term that is build from countably many variables and constants for elements of $\mathcal{A}$ using the binary operation $\cdot : |\mathcal{A}| \times |\mathcal{A}| \rightharpoonup |\mathcal{A}|$. As notation, we write $t[x_1, \ldots, x_n]$ to be a polynomial over the variables $x_1, \ldots, x_n$, *i.e.*, with free variables $x_1, \ldots, x_n$. When the variables are clear from context, we will abbreviate $t[x_1, \ldots, x_n]$ as simply t. For $a_1, \ldots, a_n \in \mathcal{A}$, we write $t[a_1/x_1, \ldots, a_n/x_n] \downarrow$ if it is defined (recall that $\cdot$ is a partial operation). To abstract over a variable x in a term $t[x_1, \ldots, x_n, x]$, we write $\Lambda x.\, t[x_1, \ldots, x_n, x]$, in which case $(\Lambda x.\, t[x_1, \ldots, x_n, x]) \cdot a \triangleq t[x_1, \ldots, x_n, a/x]$. As usual, we have a fv defined in the typical manner that computes the free variables of a term. Finally, we write $t_1 \cong t_2$ whenever $t_1 \downarrow \vee t_2 \downarrow \implies t_1 = t_2$. In words, $t_1 \cong t_2$ whenever both are defined and are equal or whenever both are undefined. The collection of polynomials is written $\mathcal{T}(\mathcal{A})$.

Now, we can give the definition of a pca. Again, this is not the standard definition, but this equivalent characterization is suitable for our purposes.

Definition 4.8.2 ([91, Lem. 3.1])**.** An applicative structrue $\mathcal{A}$ is a *partial combinatory algebra (pca)* if for every polynomial $t \in \mathcal{T}(\mathcal{A})$ and variable x, there exists a polynomial $\Lambda x.\, t \in \mathcal{T}(\mathcal{A})$ with $\mathrm{fv}(\Lambda x.\, t) \subseteq \mathrm{fv}(t) \backslash \{x\}$ such that $\Lambda x.\, t \downarrow$ and $(\Lambda x.\, t) \cdot a \cong t[a/x]$ for any $a \in \mathcal{A}$.

In essence, this captures the idea that abstraction and substitution are related in the usual way, *i.e.*, $(\Lambda x.\, t) \cdot a \cong t[a/x]$ for any $a \in \mathcal{A}$.

We give examples of relevant pcas.

Example 2 (Kleene's first algebra [91, Ex. 3.1]). $\mathcal{K}_1 = (\mathbb{N}, \cdot)$ is a pca for $n \cdot m \triangleq \{n\}\, m$, where $\{n\}\, m$ is Kleene application. That is, apply the n-th recursive function to the argument m. Operationally, we might think of this pca as an abstraction of a Turing-machine.

Example 3 (Kleene's second algebra [91, Ex. 3.4]). $\mathcal{K}_2 = (\mathbb{N}^{\mathbb{N}}, \cdot)$ is a pca where the application

satisfies a finite prefix property (see Section 2.2.1). The space $\mathbb{N}^{\mathbb{N}}$ is known as *Baire space*. Operationally, we might think of this pca as an abstraction of a Type-2 machine.

In particular, this last example highlights the connection with Type-2 computability. Indeed, we can think of a pca as abstracting over a model of computation.

For each pca $\mathcal{A}$, we can define a category of *assemblies* $\mathbf{Asm}(\mathcal{A})$ and a category of *modest sets* $\mathbf{Mod}(\mathcal{A})$ over it. Modest sets are sufficient for our purposes—they can be considered as a category of data types that are implementable by the pca $\mathcal{A}$.

Definition 4.8.3 ([91, Defn. 4.1])**.** Let $\mathcal{A}$ be a pca. The category of *assemblies* $\mathbf{Asm}(\mathcal{A})$ is defined as follows:

- Objects are tuples $X = (|X|, \delta_X)$, where $|X|$ is a set and δ_X is a relation associating every $x \in |X|$ with a non-empty subset of $\mathcal{A}$. We sometimes write this relation as $a \vDash_X x$ instead of $a \in \delta_X(x)$ for any $a \in \mathcal{A}$, and say that a *realizes* x.
- Morphisms are maps $f : |X| \to |Y|$ such that for any $a \vDash_X x$, there exists $e \in \mathcal{A}$ such that $e \cdot a \downarrow$ and $e \cdot a \vDash_X \delta_Y(f(x))$. In this case, e is said to *track* f, written $e \vDash f$.

An assembly X over $\mathcal{A}$ is a *modest set* if $x = y$ whenever $\delta_X(x) \cap \delta_X(y)$ is non-empty. In other words, elements of the underlying set for a modest set are completely determined by their realizers, *i.e.*, different elements cannot share realizers.[20] The category of *modest sets* $\mathbf{Mod}(\mathcal{A})$ is the full subcategory of $\mathbf{Asm}(\mathcal{A})$ with modest sets as objects.

Now that we have some of the basic definitions, we can discuss possible connections with our semantics.

The chain of inclusions we would like to show, where we write $\subset$ to mean full subcategory, is the following:

$$\mathbf{TD} \subset \mathbf{TP} \subset \mathbf{Mod}(\mathcal{K}_2) \subset \mathbf{Asm}(\mathcal{K}_2) \subset \mathbf{RT}(\mathcal{K}_2)\,.$$

[20] Note that this is equivalent to the requirement that a representation in the context of Type-2 computability be surjective.

1. **TD** $\subset$ **TP**: Recall that a topological domain is a topological predomain with least element.
2. **TP** $\subset$ **Mod**$(\mathcal{K}_2)$: This is a conjecture. Battenfeld in his thesis [11, Ch. 6] shows that **TP** $\subset$ **Mod**$(\mathcal{P}(\mathbb{N}))$, where $\mathcal{P}(\mathbb{N})$ is the pca that corresponds to *Scott's graph model* [91, Ex. 3.2]. Battenfeld remarks that it is also possible to use Kleene's second algebra $\mathcal{K}_2$ instead of $\mathcal{P}(\mathbb{N})$ and obtain similar results [11, Ch. 6]. However, he does not give the details and we have not fully checked the details ourselves. Hence, we leave this as a conjecture.
3. **Mod**$(\mathcal{K}_2) \subset$ **Asm**$(\mathcal{K}_2)$: Recall that a modest set is an assembly where every element is completely determined by its realizers.
4. **Asm**$(\mathcal{K}_2) \subset$ **RT**$(\mathcal{K}_2)$: The category **RT**$(\mathcal{A})$ is known as the *realizability topos* over the pca $\mathcal{A}$ [91, Sec. 5]. It is well-known (among those that study realizability) that **Asm**$(\mathcal{A})$ forms a full subcategory of **RT**$(\mathcal{A})$ for any pca $\mathcal{A}$ [91, Sec. 5]. In particular, the subcategory of assemblies corresponds exactly to the $\neg\neg$-separated object of **RT**$(\mathcal{A})$ [91, Sec. 5].[21]

Remark (Hints in the literature). Faissole *et al.* [29] recently give a formalization of valuations using synthetic topology and Homotopy Type Theory (HoTT) in the Coq proof assistant that hints at the connection of realizability with our semantics for probabilistic modeling language based on topological domains. They add (1) axioms for synthetic topology and (2) axioms for HoTT (*i.e.*, univalence) to Coq's logic (*i.e.*, as a shallow embedding). Then, they formalize valuations and their associated operations (*e.g.*, integration). Interestingly, they also discuss possible semantic models of this extended logic, which is where a possible connection to realizability can be found. First, we should review their semantics and how it relates to ours.

[21] Let E be a binary relation. We say that an object X of a topos $\mathcal{E}$ is *$\neg\neg$-separated* if the formula $\forall x, y : X., \neg\neg E(x, y) \to E(x, y)$ holds in $\mathcal{E}$. Intuitively, we should think of E as an equality relation, in which case we can think of an $\neg\neg$-separated object as an object with a stable equality. If we can show that the base topological domains and constructions we use on them yield stable equalities, then we will have found an embedding of topological domains into **RT**$(\mathcal{K}_2)$.

They suggest using standard models of synthetic topology (*i.e.*, subcategories of presheaf categories of topological spaces) and then construct valuations by defining a monad on the appropriate category. This is similar to our semantics as well, where we also define a probability monad on an appropriate category to interpret valuations. Interestingly, we could use other models to interpret the synthetic topology axioms such as represented spaces. Note that the standard models of synthetic topology are supercategories, whereas the ones listed above are all subcategories. Another difference is that they treat recursion using enriched categories, whereas topological domains by construction contain a least element.

Remark (Why investigate this connection?). It may be fruitful to investigate the connections between a semantics for probabilistic programs based on Type-2 computable distributions and realizability. In particular, full-abstraction and universality have been studied in the context of using realizability toposes to give semantics to PCF-like languages [59]. Consequently, if we can embed such a semantics into a realizability topos, then we can leverage these results. In particular, this would rigorously formalize our motto: "Turing-complete probabilistic modeling languages express computable distributions"!

4.9 Related Work

In this section, we review related work on the semantics of probabilistic programming languages and comment on its relation to ours. Earlier work focuses on semantics that can be used to model random algorithms (*e.g.*, [107, 54]). More recently, the focus has shifted towards machine learning applications such as Bayesian inference, which requires an account of continuous distributions and observation of zero probability events (*e.g.*, [17]).

Saheb-Djahromi developed a probabilistic version of LCF by considering distributions on

	distributions supported	language features	sampling interpretation	distribution interpretation
Jones *et al.* [50]	discrete	higher-order w/ recursion	yes and computable	yes
Ramsey *et al.* [83]	discrete	higher-order w/o recursion	yes and computable	yes
Kozen *et al.* [54]	measures	first-order w recursion	yes	yes
Park *et al.* [75]	measures	higher-order w recursion	yes	no
Borgström *et al.* [17]	measures	first-order w/o recursion	no	yes
Toronto *et al.* [97]	measures	first-order w recursion	no	yes
Our work	computable	higher-order w/ recursion	yes and computable	yes

Figure 4.5: A comparison of the different approaches to semantics of probabilistic programs.

base types, *i.e.*, booleans and naturals [85].[22] He constructs two additional CPOs, one CPO for the distributions on booleans and the other CPO for distributions on naturals, and uses this to give denotational semantics to Probabilistic LCF. He also gives operational semantics as a Markov chain (described by an infinite transition matrix as all spaces he considers distributions on are discrete) and shows that the operational semantics is equivalent to the denotational semantics. Hence, the importance of relating a sampling semantics with a distributional semantics has been recognized since the advent of the study of the semantics of probabilistic programs.

Saheb-Djahromi extended his previous work by showing how to construct a CPO of Borel measures on an arbitrary CPO D using D's Scott topology [86]. However, the resulting CPO

[22] As a historical note, this work followed soon after Plotkin's landmark paper "LCF considered as a programming language" [81] so the study of the semantics of high-level probabilistic languages is almost as old as the study of the semantics of high-level languages itself.

of Borel measures did not possess domain-like properties (*e.g.*, it was not ω-algebraic).

Jones, in her seminal work, developed the theory of valuations on CPOs to further the study of distributions on CPOs [50, 49].[23] She considers *valuations* on the Scott open sets of a CPO, instead of *measures*. The functor P that sends a CPO D to the CPO of valuations $P(D)$ on D is the probabilistic powerdomain (*i.e.*, distributions on powersets). However, this construction is not closed under the function space.[24] Consequently, she interprets the function space $D \Rightarrow E$ probabilistically as $D \Rightarrow P(E)$ and not $P(D) \Rightarrow P(E)$. Our semantics has the same restrictions as hers as well.

Instead of taking order-theoretic structure as primary and extending it with probabilistic concepts, another line of research takes probabilistic structure as primary and derives order-theoretic structure to support recursion. As a reminder, the probabilistic powerdomain is not well-behaved on continumm-sized spaces so this alternative approach can better support standard continuous distributions.

Kozen takes a structure amenable for modeling probability as primary, *i.e.*, Banach spaces, and derives order-theoretic structure from it [54].[25] Consequently, his semantics supports standard continuous distributions. However, his semantics does not support higher-order functions. In addition to the distributional semantics, Kozen also gives a sampling semantics and shows it equivalent to the distributional semantics. Hence, we can view our work as a modern take on Kozen's semantics that extends to support higher-order functions as well as gives an account of computability.

---

[23] As terminology, she refers to CPOs as IPOs.

[24] To our knowledge, it is still an open problem [51].

[25] This shift in viewpoint is similar in spirit to topological domains, which take the topology as primary and the order as derived.

Panangaden identifies a category of stochastic relations and shows how to use it to give denotational semantics to Kozen's first-order while language [74]. The category has measurable spaces as objects and probability kernels as morphisms (as opposed to measurable functions). He then identifies (partially) additive structure in this category and uses it to interpret fixpoints for Kozen's while language.

Continuing in the direction of starting with probabilistic structure and deriving order-theoretic structure, Danos *et al.* identify Probabilistic Coherence Spaces (PCSs) and use it to give denotational semantics to a probabilistic variant of PCF extended with (countable) choice [21]. Hence, their approach supports discrete distributions. They also give an operational semantics and provide an adequacy result: the denotation of a term evaluated at some natural is the probability that a term reduces to it.

The semantics of probabilistic programs has also been studied in the context of Bayesian machine learning.[26] Ramsey *et al.* introduced a stochastic lambda calculus with discrete distributions via a probability monad, but without recursion, and translated it into a language of measure terms [83]. Ramsey *et al.* use Jones' powerdomain of valuations to give denotational semantics to the stochastic lambda calculus, but interpret the language of measure terms using discrete measures. Their intention was to show how the language of measure terms could efficiently support variable elimination, a technique for computing expectations. They also hint at Jones' powerdomain as a starting point for extending the stochastic lambda calculus with recursion.

Park *et al.* support continuous distributions in a PCF-like programming language, also

[26] As a historical note, Gibbs sampling was rediscovered by the statistics community in 1984 by Geman and Geman [36], which led to a resurgence of Bayesian statistics. Indeed, the BUGs (Bayesian inference Using Gibbs sampling) language was developed soon after to take advantage of Gibbs sampling.

with a probability monad, and provide a small-step sampling semantics [75]. They show how to encode Bayesian models in their language and how to answer probabilistic queries in their language (*e.g.*, by programming a rejection sampler). Their sampling semantics deterministically maps input randomness into outputs. Hence, the sampling semantics is in essence identical to Kozen's and ours. However, we use random bit-streams to make the connection to Type-2 computability whereas Park *et al.*assume a blackbox supply of random real numbers and do not address computability. Moreover, Park *et al.* do not give denotational semantics.

Related to the study of operational semantics of probabilistic programs but without an emphasis on machine learning, Dal Lago *et al.* investigate a probabilistic lambda calculus more directly (*i.e.*, without a monad) by extending the lambda calculus with a choice operator and studying its operational semantics [56]. Thus, they study a non-deterministic untyped lambda calculus. They give both small-step and big-step semantics in call-by-value and call-by-name variations. In addition, they show that the language is sound and complete with respect to computable probability distributions on the natural numbers. Hence, their work focuses on discrete distributions.

Borgström *et al.* give denotational semantics to a first-order probabilistic programming language without recursion based on *measure transformers* [17]. They propose the idea of *types as measurable spaces*, *i.e.*, the interpretation of a type is the space of measures (which itself is a measurable space) on the base measurable space associated with the type. Thus, they work in the category of measurable spaces, identify a functor $\mathscr{P}$ that sends a measurable space D to the space of measures on D, and lift the non-probabilistic semantics on morphisms $f : D \to E$ as $\mathscr{P}(f) : \mathscr{P}(D) \to \mathscr{P}(E)$ (hence, measure transformer) to give probabilistic

semantics.[27] They show how to compile this language into a factor graph for inference.

Another critical aspect of their work is to give semantics to observation of zero probability events, in essence by taking the limit of a sequence of sets that converges to 0. Here, we point out that showing that a regular conditional distribution exists is highly non-trivial and also that unless certain *nice* conditions (typically topological in nature) are imposed, the sequence of sets can change the answer [Kec-Slepian Tjur point]. Moreover, by the non-computability of conditioning, it is unclear that their semantics is faithfully implementable.

Toronto *et al.* give denotational semantics to a first-order probabilistic language with recursion [97]. They translate the source language into a variant of lambda calculus extended with ZFC set theory called λ_{ZFC} [96]. The idea is to interpret a probabilistic program as a random variable, *i.e.*, a measurable function and treat conditioning as a preimage computation, which is entirely set-theoretic. Compared to other denotational semantics of probabilistic programs, theirs is somewhat nonstandard. For instance, it is not clear (at least to me) what a model of λ_{ZFC}, the target of denotation, is. Moreover, they support recursion with an operational semantics instead of with the usual order-theoretic treatment. They support conditioning in a similar manner to Borgström *et al.*, and hence, will encounter the same issues.

Staton *et al.* give both operational and denotational semantics to a higher-order probabilistic language without recursion [90], and relate the two. They first give denotational semantics to the first-order subset of their language by interpreting types using the category of measurable spaces (as Borgström [17]). To extend this to the higher-order setting (because they category of measurable spaces is not cartesian closed), they consider the functor category obtained by the Yoneda embedding. It is well that this *presheaf category* is cartesian closed, and

[27] I have taken the liberty to summarize their work using the language of category theory, although their original work is not presented in this manner.

hence, has the structure to interpret higher-order functions.

5

Towards Compilation of Probabilistic Modeling Languages

In this chapter, we review MCMC methods in preparation for the second part of the dissertation, where we describe a PPL compiler that generates MCMC inference algorithms. MCMC is a fairly general and blackbox method [18] that can be used to sample from analytically intractable posterior distributions. Consequently, it is the inference method of choice for many PPLs [19, 39, 106, 94, 99**?**] as well as the approach used in this dissertation. After we review

MCMC basics, we will walk through at a high-level how to apply it to a GMM to highlight some of the design choices available to a compiler.

5.1 Markov Chain Monte Carlo (MCMC)

We begin our review of MCMC in a general setting (Section 5.1.1) to introduce the basic concepts behind MCMC, including Markov chains and the Monte Carlo principle. Then, we will review MCMC in the context of Bayesian inference (Section 5.1.2), including standard MCMC algorithms such as Metropolis-Hastings (MH). For more background on MCMC, we refer the reader to the MCMC Handbook [18].

5.1.1 General MCMC

A MCMC (Markov Chain Monte Carlo) algorithm, as its name suggests, combines (1) Markov chains and (2) the Monte Carlo principle. The former is used to construct a sampler targeting a distribution of interest. The latter is used to convert a collection of samples into an estimate of an expectation (with respect to that distribution).

Markov chains Let $(X, \mathcal{F})$ be a measurable space. A *Markov chain* taking values in X is a sequence of X-valued random variables $(\mathbf{X}_n)_{n\in\mathbb{N}}$ such that

$$\mathbb{P}(\mathbf{X}_{n+1} \mid \mathbf{X}_1, \ldots, \mathbf{X}_n) = \mathbb{P}(\mathbf{X}_{n+1} \mid \mathbf{X}_n)\,.$$

In words, the random variable $\mathbf{X}_{n+1}$ depends on only the previous random variable $\mathbf{X}_n$ instead of the entire sequence of random variables $(\mathbf{X}_0, \ldots, \mathbf{X}_n)$ preceding it. Let $\kappa : X \times \mathcal{F} \to$

$[0, 1]$ be a probability kernel such that

$$\mathbb{P}(\mathbf{X}_{n+1} \in B) = \int_X \kappa(\cdot, B)\, d\mu$$

for any measurable B, where $\mu = \mathbb{P}(\mathbf{X}_n \in \cdot)$. As a reminder, we can think of a probability kernel as the generalization of a conditional density (see kernel in Section 2.3.2). Hence, we can think of κ as describing the transition probabilities of the Markov chain. The *limiting distribution* of the Markov chain is any distribution π that satisfies

$$\pi(B) = \int_X \kappa(\cdot, B)\, d\pi\,,$$

where B is measurable. The limiting distribution is also called the *equillibrium distribution* or *invariant distribution* because the probability kernel κ leaves π invariant.

Monte Carlo Given N i.i.d. samples $\{x_1, \ldots, x_N\}$ drawn from a distribution π, we can estimate the expectation of a (measurable) function $f : X \to \mathbb{R}$ via the *Monte Carlo principle*:

$$\int_X f\, d\pi \approx \frac{1}{N} \sum_{n=1}^{N} f(x_n)\,.$$

MCMC Let π be the *target distribution*, *i.e.*, the distribution we wish to draw samples from. The idea behind a MCMC algorithm is to construct a Markov chain $(\mathbf{X}_n)_{n\in\mathbb{N}}$, by specifying its transition kernel κ, whose limiting distribution is π. Then, for large enough values of n, we have that $\mathbb{P}(\mathbf{X}_n \in \cdot) \approx \pi$. Note that different transition kernels, and hence difference Markov chains, may have the same limiting distribution. However, even though different Markov chains may share the same limiting distribution, the number of steps n we need to run each

Markov chain before its n-th marginal distribution is approximately the limiting distribution can vary. Thus, in practice, one would ideally design a MCMC algorithm's transition kernel so that it converges quickly to the limiting distribution.

It is common to apply the Monte Carlo principle to the samples generated by a single MCMC algorithm, *i.e.*, from a single Markov chain. This introduces bias into the Monte Carlo estimate because the samples are correlated. Nevertheless, this saves computation time. Thus, in practice, one would ideally design a MCMC algorithm to minimize its within-chain correlation.

5.1.2 MCMC for Bayesian Inference

MCMC in an applied setting can be introduced using the MH (Metropolis-Hastings) algorithm. As a reminder, posterior inference is analytically intractable in general because we cannot compute the posterior distribution's normalizing constant. As we will see, the MH algorithm overcomes this limitation by constructing a Markov chain whose limiting distribution is the posterior distribution of interest using only its unnormalized density.

Figure 5.1 summarizes the MH algorithm and transition kernel. The MH sampling algorithm uses the Markov chain $(\mathbf{X}_n)_{n\in\mathbb{N}}$ defined recursively as $\mathbf{X}_{n+1} = \kappa_{MH}(\mathbf{X}_n)$, where $\mathbf{X}_0$ is some starting point. Hence, the MH algorithm iteratively applies the MH transition kernel. The transition kernel depends on the (potentially unnormalized) density π of the target distribution and is parameterized by a proposal distribution q. Given a current state x, the MH transition kernel first draws a sample y according to the proposal distribution $q(\cdot \mid x)$. Then,

Input : A (potentially unnormalized) density π, a proposal distribution q, an initial point x_0, and the number of iterations N to run the Markov chain.
Output: A Markov chain with limiting distribution π.
Function $MH(\pi, q, x_0, N)$
  For $n \leftarrow 1 \dots N$
    $x_n \leftarrow \kappa_{MH}(\pi, q, x_{n-1})$;
  return $(x_1, \dots, x_N)$;
Function $\kappa_{MH}(\pi, q, x)$
  $y \sim q(\cdot \mid x)$;
  $\alpha \leftarrow \min\left(1, \frac{\pi(y)\, q(y|x)}{\pi(x)\, q(x|y)}\right)$;
  return $y \oplus_\alpha x$;

Figure 5.1: Pseudocode for the MH algorithm and MH transition kernel. The Markov chain used by the MH sampling algorithm is defined recursively as $\mathbf{X}_{n+1} = \kappa_{MH}(\mathbf{X}_n)$, where $\mathbf{X}_0$ is some starting point.

it computes the *acceptance ratio*

$$\alpha(x, y) = \min\left(1, \frac{\pi(y)\, q(y \mid x)}{\pi(x)\, q(x \mid y)}\right),$$

which is a function of the current state x and proposal state y. Finally, it returns y with probability α and x otherwise. One can see by calculation that this transition kernel leaves π invariant.

As we mentioned previously, the specifics of a MCMC algorithm's transition kernel impact its convergence rate and within-chain correlation. In the MH algorithm we just saw, this corresponds to the choice of proposal distribution q. To guide the construction of a proposal distribution, MCMC researchers have designed variations of the MH algorithm with proposal distributions of a certain form. These algorithms include Gibbs sampling, Hamiltonian Monte Carlo (HMC) sampling, and Slice sampling. The requirements of each algorithm are summarized in Figure 5.2. For each algorithm, we list the form of the proposal, the number of likeli-

Algorithm	Form of Proposal	Number of Likelihood Evaluations	Sensitivity to Tuning
Metropolis-Hastings (MH)	arbitrary	1	very high*
Hamiltonian Monte Carlo (HMC)	gradient-based	variable	high
Gibbs	full-conditional	0	none
Slice	step-out procedure	variable	low-high*

Figure 5.2: A table summarizing the requirements of standard MCMC algorithms. The superscript * indicates that the value is dependent on the form of the proposal.

hood evaluations required, and the sensitivity of the algorithm to parameter tuning. For more detail on each of these algorithms, we refer the reader to the MCMC Handbook [18].

GIBBS SAMPLING A Gibbs sampling algorithm uses full-conditional distributions as proposals. Suppose we want to construct a sampler for the target distribution $p(x_1, \ldots, x_N)$. A *full-conditional* of $p(x_1, \ldots, x_N)$ is any conditional distribution

$$p(x_n \mid x_{-n}) \,,$$

where the notation $x_{-n} \triangleq \{x_1, \ldots, x_N\} \backslash \{x_n\}$ indicates all the variables except the n-th one for any $1 \leq n \leq N$. A Gibbs sampling algorithm samples from the target distribution by sequentially sampling from all the full-conditionals. One can check that the acceptance ratio is always 1. Thus, the proposal is always accepted and the acceptance ratio does not need to be computed.

HMC SAMPLING A HMC algorithm constructs a proposal that uses the gradient of the log-likelihood of the model. Thus, a HMC algorithm can only be applied to continuous spaces.

We illustrate one sense in which a MCMC algorithm is *compositional*. Suppose we would like to construct a MCMC sampler for a distribution $p(x, y, z)$ defined on $\mathbb{R}^3$. We can construct this sampler, for instance, by combining a MCMC sampler that samples the xy-plane with a MCMC sampler that samples the z-axis. Note that we can interpret this as a Gibbs sampler that targets $p(x, y, z)$ by sampling from the conditionals $p(x, y \mid z)$ and $p(z \mid x, y)$.

5.2 A Running Example

In this section, we will walkthrough how to hand-construct a MCMC inference algorithm for a GMM to provide a high-level idea of what goes into the implementation of such an algorithm. For convenience, Figure 5.3 summarizes the generative process for a GMM again (see Figure 1.2a in the introduction).

For the purposes of deriving and implementing an inference algorithm, practioners often turn to the representation of a probabilitistic model in terms of its density. Recall that a density can be used to compare the relative likelihoods of different points in the parameter space. The density for the GMM is given below.

$$p(\mu, z, y) = \prod_{k=1}^{K} p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu_k) \prod_{n=1}^{N} p_{\mathcal{D}(\pi)}(z_n)\, p_{\mathcal{N}(\mu_{z_n}, \Sigma)}(y_n)$$

In practice, it is more convenient to work with the model's log-likelihood, which log-transforms the model's density.

$$\mathcal{L}(\mu, z, y) = \sum_{k=1}^{K} \log(p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu_k)) + \sum_{n=1}^{N} \log(p_{\mathcal{D}(\pi)}(z_n)) + \log(p_{\mathcal{N}(\mu_{z_n}, \Sigma)}(y_n))$$

Now that we have a representation of a GMM in a form ammenable for inference, we can

$$\begin{aligned}
\mu_k &\sim \mathcal{N}(\mu_0, \Sigma_0) \;\text{ for } 1 \leq k \leq K \\
z_n &\sim \mathcal{D}(\pi_1, \ldots, \pi_K) \;\text{ for } 1 \leq n \leq N \\
y_n \mid z_n, \mu &\sim \mathcal{N}(\mu_{z_n}, \Sigma) \;\text{ for } 1 \leq n \leq N
\end{aligned}$$

Figure 5.3: The generative process for a GMM, a model for clustering a collection of N points in $\mathbb{R}^D$ into K clusters centered at $\mu_k \in \mathbb{R}^D$ (for $k \in 1, \ldots, K$), written with random variable notation. The model is parameterized by the *hyper-parameters* μ_0, Σ_0, π, and Σ.

construct a MCMC algorithm for it. As a first step, we might consider decomposing the density according to its full-conditionals so we can apply a different MCMC algorithm to the cluster means μ and cluster assignments z.

$$\begin{aligned}
p(\mu \mid z, y) &= \prod_{k=1}^{K} p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu_k) \prod_{n=1}^{N} p_{\mathcal{N}(\mu_{z_n}, \Sigma)}(x_n) \\
p(z \mid \mu, y) &= \prod_{n=1}^{N} p_{\mathcal{D}(\pi)}(z_n)\, p_{\mathcal{N}(\mu_{z_n}, \Sigma)}(y_n)
\end{aligned}$$

One reason for doing this is so that we can apply different methods for the continuous random variables (*i.e.*, the cluster means) separately from the discrete random variables (*i.e.*, the cluster assignments).

For example, we can apply HMC updates to each μ_k, which requires the derivation of partial derivatives. The notation $[\cdot]_P$ indicates an Iverson bracket, *i.e.*, it takes on the value inside the brackets if the predicate P holds and 1 otherwise.

$$\begin{aligned}
\frac{\partial}{\partial \mu_k} \mathcal{L}(\mu \mid z, y) = \left(\frac{\partial}{\partial \mu_k} p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu_k) \right) \frac{1}{p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu_k)} + \\
\sum_{n=1}^{N} \left(\frac{\partial}{\partial \mu_k} [p_{\mathcal{N}(\mu_{z_n}, \Sigma)}(x_n)]_{k=z_n} \right) \frac{1}{[p_{\mathcal{N}(\mu_{z_n}, \Sigma)}(x_n)]_{k=z_n}}
\end{aligned}$$

Notably, the GMM's log-likelihood has a partial derivative with respect to each μ_k, even though it also contains discrete random variables z_n. For the discrete variables, we can directly compute its full-conditional to perform a Gibbs update.

$$p(z_n = k) = \frac{p_{\mathcal{D}(\pi)}(k)\, p_{\mathcal{N}(\mu_k,\Sigma)}(x_n)}{\sum_{k'=1}^{K} p_{\mathcal{D}(\pi)}(k')\, p_{\mathcal{N}(\mu_{k'},\Sigma)}(x_n)}$$

A complete MCMC algorithm can be constructed by alternating HMC updates for μ with Gibbs updates for z.

This finishes the description of a MCMC algorithm at a high-level, but there are other details of this algorithm that have not been specified that will impact its computational performance. For example, consider the available parallelism in this high-level algorithm. First, the log-likelihood evaluations required by HMC can be computed as a map-reduce. In addition, the computation of the gradient itself can be parallelized. Indeed, there are many parallelization choices. For instance, we can compute each $\partial/\partial\mu_k$ in parallel. However, it may be more efficient to first pre-compute the sum

$$\sum_{n=1}^{N} \left(\frac{\partial}{\partial \mu_k} [p_{\mathcal{N}(\mu_{z_n},\Sigma)}(x_n)]_{k=z_n} \right) \frac{1}{[p_{\mathcal{N}(\mu_{z_n},\Sigma)}(x_n)]_{k=z_n}}$$

in parallel and then proceed as before. Second, the Gibbs update for each z_n can be done in parallel.

It is also possible to construct other high-level MCMC algorithms for this GMM. For example, we can exploit the conjugacy relation between the cluster means and data to obtain a closed-form full-conditional required for Gibbs sampling. Recall that a *conjugacy relation* exists between the prior distribution $p(\theta)$ and sampling distribution $p(y \mid \theta)$ when the form

of the posterior distribution $p(\theta \mid x)$ takes on the same functional form as $p(\theta)$. We can also apply Elliptical Slice sampling, a sampling method designed specifically for multivariate Gaussian distributions, to update the cluster means. In short, we can construct many inference algorithms for this simple GMM, as well as choose many possible implementation strategies.

5.3 Towards Languages with Automated Inference

As we saw in the previous section, the design space for MCMC sampling algorithms is quite large. Importantly, in the context of implementing a PPL, how should we architect a compiler to manage its complexity? Our solution is to come up with a sequence of ILs that enable a compiler to gradually and successively refine a *declarative* specification of a model and a query for posterior samples into an *executable* MCMC sampling algorithm. Hence, we will focus on automating a standard technique for probabilistic inference that also forms the basis of many PPL implementations.

6

Compiling MCMC Algorithms for Probabilistic Modeling

In this chapter, we describe a compiler for the AugurV2 language, a language that is similar in expressive power to Bugs [94] and Stan [19]. Although these languages are not as expressive as PPLs embedded in general-purpose languages, they can still express practical probabilistic models of interest. Indeed, Bugs has been one of the most widely used PPLs to date (*e.g.*, see Bayesian Modeling Using WinBUGS [72]). Even in this restricted setting, building a

system that can automate inference efficiently is still a challenging task due to the analytical intractibility of inference.

The AugurV2 compiler is architected like a traditional compiler. We summarize the phases of compilation below.

- The frontend (Section 6.2) transforms a model in the surface language into the model intermediate representation, which can be analyzed to derive model-specific facts useful for generating MCMC algorithms. Hence, the frontend deals exclusively with models.
- The middle-end (Section 6.3) transforms a model into a high-level, executable inference algorithm. For instance, it is tasked with reifying code that computes mathematical quantities such as conjugacy relations and gradients. Hence, the middle-end touches aspects of the model and inference.
- The backend (Section 6.4) transforms an executable inference algorithm into a native inference algorithm. Hence, the backend deals exclusively with inference. The backend is the most standard of the phases and is tasked with generating Cuda/C code that makes all sources of memory usage and parallelism explicit. The backend also statically bounds the memory requirements of a MCMC sampling algorithm so the runtime does not need dynamic memory management.

To support these phases of compilation, the AugurV2 compiler uses several intermediate languages (ILs), summarized below.

- The compiler uses a *Density IL* to express the density factorization of a model (Section 6.2.1). This IL can be analyzed to symbolically compute the conditionals of a model so that the compiler can generate *composable* MCMC algorithms.
- The compiler uses a *Kernel IL* to express the high-level structure of a MCMC algorithm as a composition of MCMC algorithms (Section 6.3.1). In particular, the AugurV2 compiler supports multiple kinds of basic MCMC algorithms, including ones that leverage conjugacy relations (*e.g.*, Gibbs) and gradient information (*e.g.*, HMC). The Kernel IL is similiar to the subset of the Blaise language [16] that corresponds to inference. We use the IL for the purposes of compilation, whereas Blaise focuses on the expressivity of the language.
- The compiler uses *Low++* and *Low-- ILs* to express the details of MCMC inference (Section 6.3.3). The first exposes parallelism and the second exposes memory management. The AugurV2 compiler leverages these ILs to support both CPU and GPU (Section 6.4) compilation of composable MCMC algorithms.

Our preliminary experiments show that such a compilation process enables us to both leverage the flexibility of composable MCMC inference algorithms on the CPU and GPU as well as improve upon the scalability of automated inference (Section 6.6). We have also found it relatively easy to add new base MCMC updates to the compiler without restructuring its design. Hence, the compiler is relatively extensible. We do not address the compilation of more expressive PPLs, but hope that the ideas used in the compilation of AugurV2 can provide insight into the compilation of more expressive PPLs.

6.1 AugurV2 Overview

In this section, we will introduce the AugurV2 system concretely by showing (1) how to encode a GMM in the modeling language (Section 6.1.1) and (2) how to use the system to fit a GMM to observed data (Section 6.1.2). Throughout the chapter, we will use the GMM as a running example to explain how the AugurV2 compiler automates the construction of MCMC sampling algorithms. After we have introduced the entire, end-to-end system, we will the provide an overview of the AugurV2 compiler (Section 6.1.3).

6.1.1 Modeling Language

At a high-level, the AugurV2 modeling language expresses *Bayesian networks* whose graph structure is fixed. This class contains many practical models of interest, including regression models, mixture models, topic models, and deep generative models such as sigmoid belief networks. Models that cannot be expressed include ones that use non-parametric distributions (which can be encoded in a general-purpose language) and models with undirected dependency structure (*e.g.*, Markov Random Fields whose dependency structure is hard to express

```
(K, N, mu_0, Sigma_0, pis, Sigma) => {
  param mu[k] ~ MvNormal(mu_0, Sigma_0)
      for k <- 0 until K ;
  param z[n] ~ Categorical(pis)
      for n <- 0 until K ;
  data y[n] ~ MvNormal(mu[z[n]], Sigma)
      for n <- 0 until N ;
}
```

Figure 6.1: The GMM (*i.e.*, the running example) encoded as an AugurV2 program. As the example illustrates, the AugurV2 modeling language mirrors random variable notation.

in a functional setting). Note that even in this setting, inference can still be difficult due to the presence of high-dimensional distributions. For example, the dimensionality of a mixture model such as the GMM scales with the number of observations–each observed point introduces a corresponding latent (*i.e.*, unobserved) cluster assignment.

Figure 6.1 contains an AugurV2 program encoding our running example. The *model body* is a sequence of declarations, each consisting of a random variable and its distribution. Each declaration is annotated as either a model parameter (`param`) or observed data (`data`). Model parameters are inferred (*i.e.*, output) whereas model data is supplied by the user (*i.e.*, input). In the case of a GMM, the means `mu` and cluster assignments `z`[1] are model parameters, while the `y` values are model data. At the top-level, the model closes over any free variables mentioned in the model body. These include the model hyper-parameters (`mu_0`, `Sigma_0`, `pis`, `Sigma`) and any other variables the model is parameterized by (`K`, `N`).[2] It is also possible to define a random variable as a deterministic transformation of existing variables, although this

[1] To be pedantic, we should call these *latent variables*.

[2] We reserve the term *model parameter* to refer to a random variable in the model whose distribution we are trying to infer. Hence, to avoid confusion, we do not refer to the variables that a model is parameterized by as model parameter.

feature is not needed to express the GMM in this example.

Random vectors (*e.g.*, `mu[k]`) are specified using comprehensions with the `for` construct. The semantics of AugurV2 comprehensions are *parallel*, meaning that they do not depend on the order of evaluation. The idea is to provide a syntactic construct that corresponds to the mathematical phrase "let $\mu_k \sim \mathcal{N}(\vec{\mu_0}, \Sigma_0)$ for $0 \leq k < K$." Because such a mathematical statement is implicitly parallel and occurs frequently in the definition of probabilistic models, we opt for syntactic constructs that capture standard statistical practice, instead of more standard programming language looping constructs (*e.g.*, an imperative for loop).

As AugurV2 provides only parallel comprehensions, we discourage users from expressing models with sequential dependencies. For example, we would need to write a Hidden Markov Model (HMM), where each hidden state depends on the previous state, by manually unfolding the entire model. This is *doable*, but does not take advantage of the design of AugurV2.

We impose two further restrictions on AugurV2 comprehensions. First, comprehension bounds cannot mention model parameters. This forces the comprehension bounds to be constant (although they can still be ragged). For this reason, we say that AugurV2 expresses *fixed-structure* models. Second, AugurV2 provides only primitive distributions whose pdf or pmf has known functional form. Hence, the models AugurV2 expresses are *parametric*. Currently, AugurV2 does not support non-parametric prior distributions (*i.e.*, distributions with an infinite number of parameters so that the number of parameters used scales with the number of observations).

```
import AugurV2Lib
import numpy as np

# Part 1: Load data
y = load_gmm_data('/path/to/data')
N, D = y.shape; K = 3
mu_0 = np.zeros(D)
Sigma_0 = np.eye(D);
Sigma = np.eye(D); pis = np.full(K, 1.0/K)

# Part 2: Invoke AugurV2
with AugurV2Lib.Infer('path/to/model') as av2:
  opt = AugurV2Lib.Opt(target='cpu')
  av2.setCompileOpt(opt)
  sched = 'ESlice mu (*) Gibbs z'
  av2.setUserSched(sched)
  av2.compile(K, N, mu_0, Sigma_0, pis, Sigma)(y)
  mapEst = av2.mapest(numIter=1000)
```

Figure 6.2: Fitting a GMM with AugurV2 using a Python interface.

6.1.2 Using AugurV2

We use the AugurV2 library interface, contained in the Python package `AugurV2`, to perform inference on models. Figure 6.2 contains an example of how to invoke AugurV2's inference capabilities. The first part of the code loads the data and hyper-parameters and the second part invokes AugurV2.

Instances of the `AugurV2Infer` class provide methods for obtaining posterior samples (*e.g.*, `mapest`). The class takes a path to a file containing the model. Once we have created an instance of an AugurV2 inference object (`av2`), we need to indicate to the compiler what kind of inference it should generate (via `setCompileOpt` and `setUserSched`). For example, we can set the target for compilation as either the CPU or GPU (`target`). We can also customize the inference algorithm by choosing our own inference schedule (via `setUserSched`).

In this example, we decide to apply Elliptical Slice sampling [68] to the cluster means (`ESlice mu`) and Gibbs sampling to the cluster assignments (`Gibbs z`). Hence, this schedule indicates a *compositional* MCMC algorithm, where we apply different MCMC updates to different portions of the model. This feature is inspired by the *programmable inference* proposed by other probabilistic programming systems (*e.g.*, Venture [61]). Indeed, we can also use AugurV2 as a way to explore the performance of inference algorithms on different models. For example, we can apply Gibbs sampling to update the cluster means as well and compare its performance to Elliptical Slice sampling. If a user schedule is not specified, the compiler uses a heuristic to select which combination of MCMC methods to use.

To compile the model, we supply the model arguments, hyper-parameters, and data (as Python variables) in the order that they are specified in the model. Thus, the AugurV2 compiler is invoked at *runtime*. Consequently, given different data sizes and hyper-parameter settings, the AugurV2 compiler may choose to generate a different inference algorithm. The compiler generates Cuda/C code depending on whether the target is the GPU or the CPU. The native inference code is then further compiled using Nvcc (the Cuda compiler) or Clang into a shared library which contains inference code for a specific instantiation of a model. After compilation, the object `av2` contains a collection of inference methods that wrap the native inference code. The frontend handles the conversion of Python values to and from Cuda/C via the Python CTypes interface. Hence, the user can work exclusively in Python. In this example, we ask for a MAP estimate (*i.e.*, a point estimate of the mode of the posterior distribution).

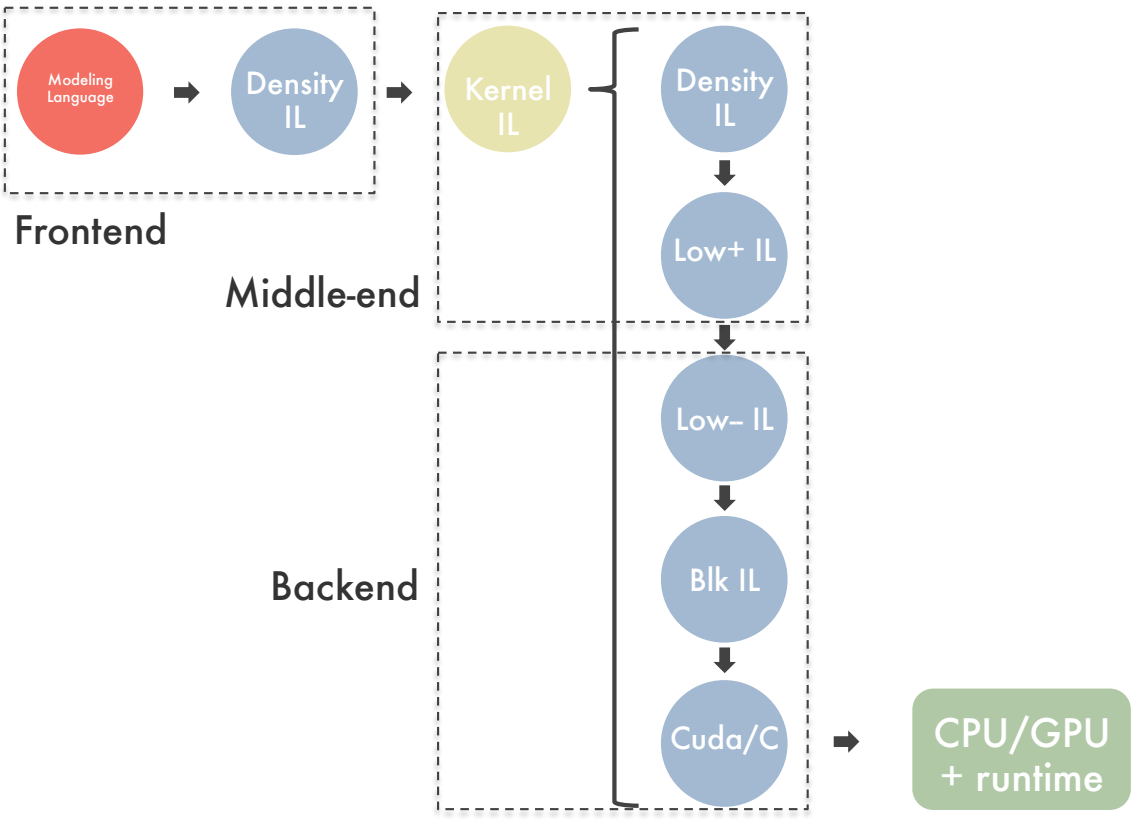


Figure 6.3: An overview of the AugurV2 compilation process. The phases of compilation include a frontend (model to declarative inference), middle-end (declarative inference to executable inference), and backend (executable inference to native inference).

6.1.3 Compilation Overview

Figure 6.3 gives an overview of the AugurV2 compilation process, illustrating the phases of compilation and the ILs involved. Like a traditional compiler, the AugurV2 compiler uses multiple ILs to guide the transformation of a high-level description of a probabilistic model into a low-level description of an inference algorithm. Moreover, the structure of the AugurV2 compiler also mirrors that of a compiler for a traditional language, although the task of each phase of the compiler is somewhat different.

Of course, we will need a semantics to justify the compilation process. As AugurV2 is a simple language that intuitively expresses Bayesian networks, its semantics is not an issue, in contrast to more expressive languages that provide higher-order functions and recursion. Nevertheless, in Chapter 7, we will give AugurV2 semantics in terms of (Type-2) computable distributions. This enables us to think of a compiler as a (computable) function that witnesses the result that (Type-2) computable distributions are realizable by (Type-2) computable sampling algorithms. In our case, the compiler realizes a MCMC sampling algorithm.

6.2 Frontend

In the first step of compilation, the compiler converts a program expressed in the modeling language into a program in the *Density IL*, which encodes the density factorization of a model. This follows standard statistical practice, where models expressed using random variables (AugurV2's modeling language) are converted into its description in terms of densities (Density IL). The compiler analyzes this representation to decompose the model according to its conditional independence relationships, or equivalently, its (unnormalized) *full-conditionals*

$$
\begin{aligned}
obj &::= \lambda(\overrightarrow{x}).\ fn \\
fn &::= p_{dist}(\overrightarrow{e}) \mid fn\ fn \mid \prod_{x \leftarrow gen} fn \mid \texttt{let}\ x = e\ \texttt{in}\ fn \mid [fn]_{x=e} \\
e &::= x \mid i \mid r \mid dist(\overrightarrow{e}) \mid op^n(\overrightarrow{e}) \mid e[e] \\
gen &::= e\ \texttt{until}\ e \\
\sigma &::= \texttt{Int} \mid \texttt{Real} \\
\tau &::= \sigma \mid \texttt{Vec}\ \tau \mid \texttt{Mat}\ \sigma
\end{aligned}
$$

Figure 6.4: The Density IL encodes the density factorization of a probabilistic model.

(see Section 6.2.2). The purpose of supporting model decomposition is so that we can generate composable MCMC inference algorithms.

The challenge with supporting the analysis comes from handling structured products. Conceptually, these products can be unfolded because we compile at runtime and AugurV2 expresses fixed-structure Bayesian networks. However, the resulting network loses regularity and can also be quite large. (Moreover, avoiding static analysis would seem to miss examining how inference on probabilistic programs compares to inference on other representations of distributions such as PGMs.) Instead, the compiler statically over-approximates the conditional independence relations.

6.2.1 Representing Models: the Density IL

The syntax for the Density IL is summarized in Figure 6.4. In essence, it encodes standard statistical notation for writing densities. As an example, the running example (Figure 6.1) is

encoded in the Density IL as

$$\lambda(K, N, \mu_0, \Sigma_0, \pi, \Sigma, \mu, z, y).\ \prod_{k \leftarrow 0 \texttt{ until } K} p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu[k])$$
$$\prod_{n \leftarrow 0 \texttt{ until } N} p_{\mathcal{D}(\pi)}(z[n]) \prod_{n \leftarrow 0 \texttt{ until } N} p_{\mathcal{N}(\mu[z[n]], \Sigma)}(y[n] \mid \mu[z[n]])\,.$$

At the top-level, a model is represented as a single function $\lambda(\overrightarrow{x}).\ fn$ with bindings $\overrightarrow{x}$ for model hyper-parameters, other model constants and data, and a density function body *fn*. A density function is either (1) the density of a primitive, parameterized distribution $p_{dist}(\overrightarrow{e})$, (2) the composition of two density functions $fn_1\ fn_2$ (*i.e.*, multiplication of two density functions), (3) a structured product $\prod_{x \leftarrow gen} fn$ that specifies a product of density functions according to the comprehension $x \leftarrow gen$, (4) a standard let-binding $\texttt{let}\ x = e\ \texttt{in}\ fn$, or (5) an Iverson bracket $[fn]_{x=e}$ that takes on the value inside the brackets when $x = e$ is satisfied and 1 otherwise.

The Density IL is simply-typed. Base types include integers `Int` and reals `Real`. Compound types include vectors `Vec` τ and matrices `Mat` σ. Thus, compound types such as vectors of matrices are allowed, whereas matrices of vectors are rejected. The type system is used to check simple properties. For example, the type system checks that densities are defined on the appropriate spaces and that comprehension bounds are specified with integers.

6.2.2 Model Decomposition: Approximating Full-conditionals

The Density IL supports an analysis that statically approximates the full-conditionals (up to a normalizing constant) of a probabilistic model. Throughout the rest of this section, the term full-conditional will refer to an unnormalized full-conditional unless stated otherwise. Before

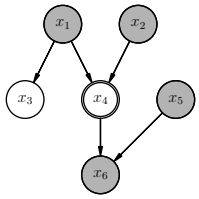


Figure 6.5: Example where the Markov blanket of x_4 is shaded.

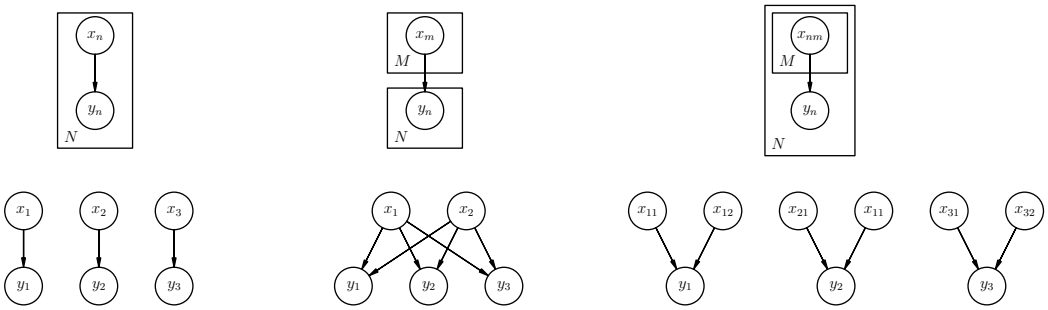
we describe the analysis, we will first make the connection between symbolically computing full-conditionals and conditional independence relationships in Bayesian networks.

FULL-CONDITIONALS IN BAYESIAN NETWORKS Consider the Bayesian network in Figure 6.5, which has the following density factorization:

$$p(x_1, x_2, x_3, x_4, x_5, x_6) = p(x_1)\, p(x_2)\, p(x_3 \mid x_1)\, p(x_4 \mid x_1, x_2)\, p(x_5)\, p(x_6 \mid x_4)\,.$$

We can read off the conditional independence relationship for the variable x_4 from the Bayesian network by computing its *Markov blanket*, which includes its parents (x_1 and x_2), its children (x_6), and its children's parents (x_5). Hence, a Bayesian network representation of a probability distribution supports conditional independence relationship queries via a graph query. It is also possible to compute the Markov blanket of a node symbolically by computing its full-conditional. For example, the full-conditional of x_4 is

$$\begin{aligned} p(x_4 \mid x_1, x_2, x_3, x_5, x_6) &= \frac{1}{Z}\, p(x_4 \mid x_1, x_2)\, p(x_6 \mid x_4) \\ Z &= \int_{x_4} p(x_4 \mid x_1, x_2)\, p(x_6 \mid x_4, x_5)\, \mathrm{d}x_4\,, \end{aligned}$$



(a) $\prod_{n=1}^{N} p(x_n)\, p(y_n \mid x_n)$ **(b)** $\prod_{n=1}^{N} p(x_n) \prod_{m=1}^{M} p(y_m \mid x)$ **(c)** $\prod_{n=1}^{N} p(y_n \mid x_n) \prod_{m=1}^{M} p(x_{nm})$

Figure 6.6: Plated graphical models (top), their unfoldings (bottom), and the corresponding structured products.

which is the result of an application of Bayes rule and canceling the densities not bound by the integral. The variables mentioned in the simplified full-conditional expression comprise x_4's Markov blanket. Hence, a density factorization representation of a probability distribution supports conditional independence relationship queries via a bound variable analysis. Note that the conditional independence relationships indicate which variables are involved but do not give the functional form of the full-conditional. As the AugurV2 compiler internally represents a model as its density factorization, it can conveniently compute on the symbolic representation already.

The above correspondence can be extended to handle density factorizations with structured products. In a Bayesian network, structured products can be represented using *plate notation*, which repeats the graph structure within the plate. Figure 6.6 illustrates plated Bayesian networks (top), their unfoldings (bottom), and the corresponding structured products. Observe that the variables inside the plate all share the same conditional independence relationships, and hence, offers a more compact representation of these relationships compared to the un-

folded Bayesian network. On the density factorization representation, variables within the same plate corresponds to densities sharing the same structured product.

The density factorization can encode the same conditional independence relationships in multiple ways. For example, the structured product

$$\prod_{n=1}^{N} p(x_n)\, p(y_n \mid x_n)$$

given in Figure 6.6a can also be written as

$$\prod_{n=1}^{N} p(x_n) \prod_{n=1}^{N} p(y_n \mid x_n)\,.$$

However, note that the unfactored encoding of the density cannot be expressed using plate notation. Hence, we can think of the analysis that the AugurV2 compiler performs on the Density IL as converting ordinary Bayesian networks into plated Bayesian networks at compile-time.

Analysis To compute full-conditionals, the compiler normalizes the model IL expression by factoring it so that all structured products with the same comprehensions bounds use the same structured product. This corresponds to the rule below.

$$\prod_{i \leftarrow gen_1} fn_1 \prod_{i \leftarrow gen_2} fn_2 \rightarrow \prod_{i \leftarrow gen_1} fn_1\, fn_2 \text{ when } gen_1 = gen_2$$

As a reminder, comprehension bound expressions in AugurV2 cannot refer to random variables and so are constant. If the compiler cannot determine the equality of comprehension bounds, it will not be factored, and precision in the approximation of the full-conditional is

lost. For example, we can factor the GMM from our running example using the above rule as

$$\left(\prod_{k \leftarrow 0 \text{ until } K} p_{\mathcal{N}(\mu_0,\Sigma_0)}(\mu[k])\right)\left(\prod_{n \leftarrow 0 \text{ until } N} p_{\mathcal{D}(\pi)}(z[n])\; p_{\mathcal{N}(\mu[z[n]],\Sigma)}(y[n] \mid \mu[z[n]])\right)$$

where we have omitted the top-level bindings.

The compiler also implements the special factoring rule

$$\prod_{i \leftarrow gen_i} fn \rightarrow \prod_{k \leftarrow gen_k} \prod_{i \leftarrow gen_i} [fn]_{k=z}\,,$$

where z is a Categorical variable with range gen_k. For example, factoring the GMM from our running example according to this rule produces

$$\left(\prod_{k \leftarrow 0 \text{ until } K} p_{\mathcal{N}(\mu_0,\Sigma_0)}(\mu[k])\right)\left(\prod_{k \leftarrow 0 \text{ until } K} \prod_{n \leftarrow 0 \text{ until } N} [fn]_{k=z[n]}\right)$$

where $fn = p_{\mathcal{D}(\pi)}(z[n])\; p_{\mathcal{N}(\mu[z[n]],\Sigma)}(y[n] \mid \mu[z[n]])$. Notably, after we apply the categorical indexing rule, it is possible to apply the factoring rule again. Currently, the compiler attempts to apply the categorical indexing rule first and then attempts to factor. The compiler tries every possible factoring. Note that probabilistic programs are typically small so this strategy is feasible in practice.

For the restricted language that we consider, we have found the two factoring rules to be precise enough to be useful for practical models. The first rule reflects the essence of plate notation—it can only capture fine-grained conditional relationships common to the entire plate. Edges that cross between two different plates are fully-connected. The second rule captures a mixture-modeling pattern. However, there are conditional independence structures

found in practice that currently cannot be represented compactly by the AugurV2 compiler.

For instance, one notable conditional independence structure that cannot be represented compactly is a sequential dependence as found in state-space models such as a HMM. As future work, we would like to add support for this kind of dependency structure. For now, however, the lack of support for this dependency structure in the analysis is intentional. First, note that the plate notation as presented cannot express such a dependence. Importantly, this is reflected in AugurV2's modeling language, which does not provide sequential comprehensions. In other words, we cannot express this kind of dependency structure in the first place. Second, one would typically use a different inference algorithm for a sequential dependence structure such as particle MCMC methods [cite], which the AugurV2 compiler currently does not support. Hence, even if the compiler could compactly represent this structure, it currently cannot exploit it. The takeaway from this discussion is that through careful PPL language design and implementation (like the design and implementation of any other language), we can encourage users to write only those models that the system can efficiently handle.

After the factoring rules have been applied, the AugurV2 compiler applies a bound variable analysis to simplify the resulting full-conditionals. As a reminder, this mathematically corresponds to an application of Bayes rule followed by canceling the densities in the denominator that are not bound by the integral. For reference, the full-conditionals for the GMM are given below, where we have omitted the comprehension bounds.

$$fn_{\mu} = \prod_{k} \left(p_{\mathcal{N}(\mu_0,\Sigma_0)}(\mu[k]) \prod_{n} [p_{\mathcal{D}(\pi)}(y \mid \mu[z[n]])]_{k=z[n]} \right)$$

$$fn_{z} = \prod_{n} p_{\mathcal{D}(\pi)}(z[n]) p_{\mathcal{N}(\mu[z[n]],\Sigma)}(y[n] \mid \mu[z[n]])$$

$$
\begin{aligned}
\mathit{sched}\,\alpha &::= \lambda(\overrightarrow{x}).\ k\ \alpha \\
k\ \alpha &::= (\kappa\ \alpha)\ ku\ \alpha \ \mid\ k\ \alpha \otimes k\ \alpha \\
ku &::= \texttt{Single}(x) \ \mid\ \texttt{Block}(\overrightarrow{x}) \\
\kappa\ \alpha &::= \texttt{Prop}\ (\texttt{Maybe}\ \alpha) \ \mid\ \texttt{FC} \ \mid\ \texttt{Grad}\ (\texttt{Maybe}\ \alpha) \ \mid\ \texttt{Slice}
\end{aligned}
$$

Figure 6.7: The Kernel IL encodes the structure of an MCMC algorithm. It is parametric in α, which is instantiated with successively lower level ILs that encode how the MCMC algorithm is implemented.

6.3 Middle-end

The AugurV2 middle-end converts a model encoded as its density factorization into a high-level, executable inference algorithm. This phase uses two additional ILs. First, the *Kernel IL* represents the high-level structure of a MCMC algorithm as a composition of basic MCMC updates applied to the model's full-conditionals. Second, the *Low++ IL* serves as the first pass target for executable MCMC inference code. It is an imperative language that makes sources of parallelism explicit, but abstracts away memory management. Importantly, we can leverage domain-specific knowledge of each base MCMC update to directly generate parallel code. Thus, we do not need to (re)discover parallelism at a lower-level of abstraction.

6.3.1 Representing Inference I: the Kernel IL

The AugurV2 compiler represents a MCMC algorithm as a composition of base updates in the *Kernel IL*, whose syntax is summarized in Figure 6.7. As an example, the user schedule presented in Figure 6.2 is encoded in the Kernel IL as

$$
\lambda(K, N, \mu_0, \Sigma_0, \pi, \Sigma, \mu, z, x).\ \ \texttt{Slice}\ \texttt{Single}(\mu)\ \mathit{fn}_{\mu} \otimes \texttt{FC}\ \texttt{Single}(z)\ \mathit{fn}_{z}\,,
$$

where fn_μ and fn_z correspond to the full-conditionals up to a normalizing constant.

The syntax

$$\texttt{Slice Single}(\mu)\ fn_\mu$$

encodes a base update indicating that we apply Slice sampling to the variable μ with proportional full-conditional fn_μ. The kernel unit ku specifies whether we should sample a variable x by itself (*i.e.*, `Single`(x)), or whether to sample a list of variables $\overrightarrow{x}$ jointly (*i.e.*, `Block`(x)). Sampling variables jointly, also known as blocking, is useful when those variables are heavily correlated. The syntax

$$\texttt{FC Single}(z)\ fn_z$$

encodes a base update indicating that we apply Gibbs sampling (using a closed-form full-conditional) to the variable z with proportional full-conditional fn_z. In general, a base update $\kappa\ ku\ \alpha$ is parametric in the representation of the proportional full-conditional α. This enables the compiler to successively instantiate α with lower-level ILs that expose more computational details of inference (*e.g.*, parallelism or memory usage). iven two MCMC updates $k_1\ \alpha$ and $k_2\ \alpha$, we can sequence (*i.e.*, compose) the two updates with the syntax $k_1\ \alpha \otimes k_2\ \alpha$. Sequencing is not commutative, *i.e.*, the MCMC update $k_2\ \alpha \otimes k_1\ \alpha$ is different from $k_1\ \alpha \otimes k_2\ \alpha$. Currently, the compiler does not check that the composition of MCMC updates is *correct*, *e.g.*, check properties such as the irreducibility or aperiodicity of the MCMC algorithm. Static checking of such properties would be an interesting direction for future work.

In addition to Slice (`Slice`) and closed-form full-conditional updates (`FC`), the Kernel IL also supports proposal-based updates (`Prop`) and gradient-based updates (`Grad`). These updates each contain an optional piece of code of type α that specifies the proposal and the gra-

dient respectively for the corresponding full-conditional that the base update is applied to. In contrast, Slice and closed-form full conditional updates can be derived purely from the corresponding model portion's full-conditional.

6.3.2 Specifying a High-level MCMC Algorithm

Once the compiler has decomposed the model according to its unnormalized full-conditionals, it will determine which base MCMC updates can be applied to each full-conditional. In this step, the compiler simply chooses *which* updates to apply and not *how* to implement them. Thus, the output is a program in the Kernel IL with full-conditionals specified in the Density IL. As a reminder, the user can supply the MCMC schedule, in which case the compiler will check that it can indeed generate the desired schedule and fail otherwise. When a user does not supply a MCMC schedule, the compiler uses a heuristic to accomplish this task. First, it determines which variables it can perform Gibbs sampling with conjugacy relations. For the remaining discrete variables, it will also apply Gibbs sampling by manually approximating the closed-form full-conditional. For the remaining continuous variables, it will apply HMC sampling to take advantage of gradient information.

6.3.3 Representing Inference II: the Low++ IL

The Low++ IL expresses parallelism available in a MCMC algorithm. After the compiler generates code in this IL, the compiler only needs to reason at the level of an inference algorithm—all aspects of the model are eliminated. Figure 6.8 summarizes the syntax for the Low++ IL. The language is largely standard, so we highlight the aspects useful for encoding MCMC.

First, the IL provides additional distributions operations *dop*, including the log-likelihood

$$
\begin{aligned}
\mathit{decl} &::= \mathit{name}(\overrightarrow{x})\{\texttt{global} : \overrightarrow{g},\ \texttt{body} : e,\ \texttt{return} : e\} \\
s &::= e \mid x\ \mathit{sk}\ e \mid e[\overrightarrow{e}]\ \mathit{sk}\ e \mid s\ s \\
&\quad \mid \texttt{if}(e)\{s\}\{s\} \mid \texttt{loop}(i \leftarrow_{\mathit{lk}} \mathit{gen})\{s\} \\
\mathit{sk} &::= \ \texttt{=} \mid \texttt{+=} \\
\mathit{lk} &::= \texttt{Seq} \mid \texttt{Par} \mid \texttt{AtmPar} \\
e &::= x \mid i \mid r \mid \mathit{dist}(\overrightarrow{e}).\mathit{dop} \mid \mathit{op}^n(\overrightarrow{e}) \mid e[e] \\
\mathit{dop} &::= \texttt{ll} \mid \texttt{samp} \mid \texttt{grad}i
\end{aligned}
$$

Figure 6.8: The Low++ IL is an imperative language that exposes parallelism in the computation of a MCMC update, but abstracts away from details of memory management.

`ll`, sampling `samp`, and gradients `grad`i, where the integer i refers to the position of the argument to take the gradient with respect to. At the level of the Density IL, the distribution operation was implicitly its density. For expressing inference algorithms, we may need other operations on distributions such as sampling from them.

Second, the IL contains a dedicated increment and assign statement x `+=` e. (We can also increment and store to locations $e[\overrightarrow{e}]$ `+=` e.) We added this additional syntactic category because many MCMC updates require incrementing some quantity. For example, we may need to count the number of occurrences satisfying some predicate to compute a conjugacy relation or accumulate derivative computations after an application of the chain-rule. Because we hope to generate parallel inference code, this separate syntactic category indicates to the compiler that the increment and assign must be done atomically.

Third, The ILs annotate loops with whether they can be executed sequentially (`Seq`), in parallel (`Par`), or in parallel given that all increment and assign operations are done atomically (`AtmPar`). For instance, we can annotate a loop that samples a collection of conditionally independent variables in parallel (*e.g.*, when implementing a conjugacy relation) with `Par`.

MCMC update	likelihood	full-conditional	gradient
MH	✓	x	x
Gibbs	x	✓	x
HMC	✓	x	✓
Reflective Slice	✓	x	✓
Elliptical Slice	✓	x	x

Figure 6.9: A summary of base MCMC updates and the primitives required to implement them.

As we will see later when we describe AugurV2's implementation of gradients (Section 6.4.4), these computations will use the loop annotation `AtmPar`.

6.3.4 Primitive Support for Base MCMC Updates

The AugurV2 compiler currently supports user-supplied MH proposals, Gibbs updates, HMC updates[3], and (reflective and Elliptical) Slice updates. Fortunately, each base update can be decomposed into yet further primitives, summarized in Figure 6.9. The rest of the functionality can be supported as library code—the primitives encapsulate the parts of the MCMC algorithm that are specific to the full-conditional. This helps us manage the complexity of the compiler. These primitives include (1) likelihood evaluation, (2) closed-form full-conditional derivation, and (3) gradient evaluation. The compiler will implement these primitives in the Low++ IL.

Likelihood evaluation It is straightforward to generate Low++ code that reifies a full-conditional as it encodes a likelihood. Full-conditionals with structured products can be reified as `AtmPar` loops. For example, if we apply Elliptical Slice sampling to sample the cluster means μ_k, then the code for the log-likelihood of the (unnormalized) full-conditional is:

[3]There is also a prototype of No-U-Turn sampling.

```
log_likelihood_mu_k(K, N, mu_0, sigma_0, pi, sigma, y, mu, z) {
  ll = 0;
  ll += MvNormal(mu[k], mu0, sigma0).ll;
  loop atmpar (n <- 0 until N) {
    ll += indicator(k == z[n], MvNormal(y[n], mu[z[n]], sigma)).ll;
  }
  return ll;
}
```

At the level of the GPU, we can implement these loops as a map-reduce. Nevertheless, at the level of the Low++ IL, we simply annotate the parallelism available and defer the implementation details to a later phase of the compiler.

Closed-form full-conditional derivation When the compiler partitions the model, it only computes a full-conditional that is known up to a normalizing constant. To obtain a closed-form solution for the full-conditional, the compiler needs to solve for the normalizing constant, which requires solving an integral. The AugurV2 compiler supports closed-form full-conditionals in two cases.

First, like Bugs and Augur, AugurV2 exploits conjugacy relations. Recall that a *conjugacy relation* exists when the form of the conditional distribution $p(\theta \mid x)$ takes on the same functional form as $p(\theta)$. This bypasses the need to compute the normalizing constant. There is a well-known list of conjugacy relations. Consequently, the AugurV2 compiler supports conjugacy relations via table lookup. For example, there is a conjugacy relation between a multivariate Normal prior and multivariate Normal likelihood in the GMM. In general, computing a conjugacy relations requires traversing the involved variables and computing some statistic. We provide the conjugacy relation for the GMM cluster means below for our running exam-

ple.

$$\mu'_k = \Sigma'(\Sigma_0^{-1}\mu_0 + n_k\Sigma^{-1}\bar{x})$$
$$n_k = \sum_{n=1}^{N}[z_n = k]$$
$$\Sigma' = (\Sigma_0^{-1} + n_k\Sigma^{-1})^{-1}$$

The compiler may fail to detect a conjugacy relation if (1) the approximation of the full-conditional is imprecise or (2) the compiler needs to perform mathematical rearrangements beyond structural pattern matching. It would be interesting to see if we can improve upon the latter situation by combining AugurV2 with a computer algebra system (CAS) as other systems have done (*e.g.*, [69]) and leveraging the CAS to solve the integral, but we leave this for future work.

Second, AugurV2 can also approximate the closed-form full-conditional for a discrete variable as a finite sum, even if a conjugacy relation does not exist. That is, it can generate code that directly sums over the support of the discrete variable up to some predetermined bound or when we have encountered most of the probability mass (*e.g.*, using Markov's inequality).

$$p(z = k \mid \vec{x}) = \frac{p(z = k)p(y \mid z = k)}{\sum_{k'} p(z = k')\, p(y \mid z = k')}$$

Below, we give Low++ code that implements this marginalization for the z variables in our GMM running example. In the example, we assume a that there is a function `normalize_and_sample` that normalizes a vector to obtain a pmf and then samples from it.

```
fc_z(K, N, mu_0, sigma_0, pi, sigma, y, mu, z) {
  loop Par (n <- 0 until N){pmf} {
```

```
    norm = 0;
    loop AtmPar (k <- 0 until N) {
      prob = Categorical(z[n], pi).ll * MvNormal(y[n], mu[k], sigma).ll;
      pmf[k] = prob;
      norm += prob;
    }
    }
    z[n] = normalize_and_sample(pmf, norm);
  }
}
```

Gradient evaluation The compiler implements source-to-source, reverse-mode automatic differentiation (AD) to support gradient evaluation of the model likelihood. For more background, we refer the reader to the literature on AD (*e.g.*, [9]). We summarize some design decisions that one make when implementing gradients below. We will write f to refer to a mathematical function and $\mathcal{A}_f$ to refer to the program code that implements f to distinguish the two.

- (Symbolic vs. AD). One reason for implementing AD over symbolic differentiation is so that the complexity of the gradient code $\mathcal{A}_{\nabla f}$ is proportional to the complexity of the original code $\mathcal{A}_f$. In contrast, symbolic differentiation can lead to code that is exponential in the complexity of the original code $\mathcal{A}_f$. However, symbolic derivatives can interoperate with computer algebra systems, whereas AD cannot.
- (Forward vs. Reverse). Suppose $f : \mathbb{R}^n \to \mathbb{R}^m$. Forward-mode AD requires n passes over the original function $\mathcal{A}_f$ to compute the gradient ∇f. Reverse-mode AD requires m passes over the original function $\mathcal{A}_f$ and a stack known as the Wengert tape [9] to keep track of the history of computations to computes the gradient ∇f. For a function such as a model log-likelihood $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$, reverse-mode AD requires one pass over the original function at the cost of increased memory usage compared to forward-mode AD.
- (Instrumentation vs. source-to-source). AD can be implemented by instrumenting the original program or as a source-to-source transformation. The former can be easier to implement in languages that support operator overloading, *i.e.*, overload all primitive operations to compute both the original function and its derivative. The latter requires a compiler pass. Nevertheless, there are opportunities for optimization. For example, a compiler can leverage that the semantics of parallel comprehensions are independent of the order of evaluation and optimize away the stack for reverse-mode AD when it is

$$
\begin{aligned}
\overline{(x, y)} &= \bar{y} \texttt{ += } \bar{x} \\
\overline{(x, i)} &= 0 \\
\overline{(x, r)} &= 0 \\
\overline{(x, dist(y_1, \ldots, y_n))} &= \bar{y_1} \texttt{ += } \bar{x} * dist(y_1, \ldots, y_n).\texttt{grad1} \\
&\ldots \\
&\bar{y_n} \texttt{ += } \bar{x} * dist(y_1, \ldots, y_n).\texttt{grad}n \\
\overline{(x, op^n(y_1, \ldots, y_m))} &= \bar{y_1} \texttt{ += } \bar{x} * op^n(y_1, \ldots, y_m).\texttt{grad1} \\
&\ldots \\
&\bar{y_n} \texttt{ += } \bar{x} * op^n(y_1, \ldots, y_m).\texttt{grad}m \\
\overline{(x, y_1[y_2])} &= \bar{y_1}[y_2] \texttt{ += } \bar{x}
\end{aligned}
$$

(a) The adjoint expression translation. A variable notated $\bar{x}$ is the adjoint of variable x. It contains the result of the partial derivative with respect to x.

$$
\begin{aligned}
\overline{p_{dist(\overrightarrow{e})}(x)} &= \overline{(x, dist(\overrightarrow{e}))} \\
\overline{fn_1\, fn_2} &= \overline{fn_2}; \overline{fn_1} \\
\overline{\texttt{let } x = e \texttt{ in } fn} &= x \texttt{ = } e; \overline{fn}; \overline{(x, e)} \\
\overline{\prod_{x \leftarrow gen} fn} &= \texttt{loop}(x \leftarrow_{\texttt{atmpar}} gen)\{[]\}\overline{fn} \\
\overline{[fn]_C} &= \texttt{if}(C)\{\overline{fn}\}\{0\}
\end{aligned}
$$

(b) The adjoint density function translation. The absence of complex control-flow in the Density IL simplifies the implementation.

Figure 6.10: The construction of an adjoint program for source-to-source, reverse-mode AD in AugurV2 from the Density IL to the Low++ IL.

translating this construct. This is an instance of where providing parallel comprehensions as a language construct can potentially increase the efficiency of inference code.

Taking the design decisions into account, the AugurV2 compiler implements reverse-mode AD as a source-to-source transformation from the Density IL to the Low++ IL. We highlight two differences as compared to more conventional choices.

First, the AugurV2 compiler implements source-to-source AD, in contrast with many other systems [19] that implement AD by instrumentation. This choice was largely motivated by the lack of complex control flow in the AugurV2 modeling language, which is the primary difficulty of implementing source-to-source AD. Here, the AugurV2 compiler leverages the semantics of parallel comprehensions to optimize the gradient code. Moreover, we found that it was easier to support the composition of different MCMC algorithms by directly generating code that implements gradients when needed, instead of redesigning the entire system runtime to support the instrumentation required for AD. We also found that this choice makes it easier to support compilation to the GPU, which cannot (efficiently) support a sophisticated runtime.

Second, the AugurV2 compiler implements source-to-source AD as a translation from one language to another (Density IL to Low++ IL), as opposed to a transformation on the same language. In a typical setting, one applies AD to code written in a general-purpose programming language. In our setting, we only need to apply AD to models written in a restricted modeling language. Hence, for simplicity, we can choose the ILs that make the translation as simple as possible (see Figure 6.10).

Figure 6.10 summarizes the source-to-source, reverse-mode AD translation implemented by the AugurV2 compiler. The input program should be transformed to contain only simple expressions. The output is an *adjoint program*, *i.e.*, a program that computes the derivative.

The translation is quite simple—it fits on a single page. Below, we provide an example adjoint program for taking the partial derivative of the log-likelihood of the model with respect to the cluster means.

```
grad_log_likelihood_mu(K, N, mu_0, sigma_0, pi, sigma, y, mu, z) {
  adj_ll = 1;
  Loop AtmPar (n <- 0 until N) {
    t1 = y[n];
    t2 = z[n];
    t3 = mu[t2];
    adj_t1 += adj_ll * MvNormal(t1, t3, sigma).grad1;
    adj_t3 += adj_ll * MvNormal(t1, t3, sigma).grad2;
    adj_mu[t2] += adj_t3;
  }
  Loop AtmPar (k <- 0 until K) {
    t0 = mu[k]
    adj_t0 += adj_ll * MvNormal(t0, mu0, sigma0).grad1;
    adj_mu[k] += adj_t0;
  }
  return adj_mu;
}
```

6.4 Backend

The AugurV2 backend performs the last step of compilation and generates *native* inference code. As we mentioned before, the target of compilation for the AugurV2 compiler is a C/Cuda runtime library that supports operations on vectors/matrices and distributions. Hence, the target of compilation for AugurV2 is minimal compared to other PPS's, which implement almost all the functionality of inference in this layer.

Except for the final step where the compiler synthesizes a complete MCMC algorithm and eliminates the Kernel IL, the backend phase is largely similar to a traditional compiler pass in terms of the kinds of transformations it performs. The AugurV2 backend (1) performs size-inference to statically bound the amount of memory usage an AugurV2 inference algorithm

consumes and (2) reifies parallelism. Towards this end, the compiler introduces the *Low-- IL* to handle memory management and the *Blk IL* to handle parallelization. The Low-- IL conceptually follows a Partioned Global Address Space (PGAS) programming model. We have not investigated the possibility of using existing languages based on the PGAS model such as Chapel [cite] or X10 [cite], but hope to do so in future work to extend the capabilities of AugurV2 to distributed settings. The current backend serves as a prototype to investigate CPU and GPU compilation of MCMC inference algorithms for fixed-structure Bayesian networks.

6.4.1 Representing Inference III: the Low-- IL

The Low-- IL is structurally the same as the Low++ IL (Figure 6.8), except that programs must manage memory explicitly. However, details of how to reify parallelism are still left abstract.

6.4.2 Size Inference

As a reminder, AugurV2 programs express fixed-structure models. Consequently, for efficiency reasons, we can bound the amount of memory an inference algorithm will use and allocate it up front. Moreover, for GPU inference, it is necessary to determine how much memory an inference algorithm will consume up front because we cannot dynamically allocate memory while executing GPU code. To accomplish this, the compiler first makes all sources of memory usage explicit. For example, primitives such as vector addition that produce a result that requires allocation will be converted into a side-effecting primitive that updates an explicit destination. These functional primitives made the initial lowering step from model to infer-

$$b ::= \texttt{seqBlk}\ \{s\} \mid \texttt{parBlk}\ lk\ x \leftarrow gen\ \{s\} \mid \texttt{loopBlk}\ x \leftarrow gen\ \{b\}$$
$$\mid e_{acc} = \texttt{sumBlk}\ e_0\ x \leftarrow gen\ \{s\ ;\ \texttt{return}\ e\}$$

Figure 6.11: The Blk IL exposes the different kinds of parallelism, including data-parallel (`parblk`), reduction (`sumblk`), and the absence of parallelism (`seqblk`).

ence tractable and can be removed at this step. Second, the compiler performs size inference to determine how much memory to allocate. Currently, the compiler does not support standard optimizations such as destructive updates. We hope to add this in the future.

6.4.3 Representing Parallelism: the Blk IL

When AugurV2 is set to target the GPU, the compiler produces Cuda/C code from Low-- IL code. Here, the compiler can finally leverage the loop annotations present in the inference code. As a reminder, the loops were annotated when the compiler first generated a base MCMC update. At this point, the compiler chooses *how* to use utilize the GPU. As with size-inference and memory management, this step is also similar to a more traditional code-generation step. Towards this end, the compiler introduces an additional IL called the *Blk IL*. The syntax is summarized in Figure 6.11.

The design of the IL is informed by the SIMD parallelism provided by a GPU. For example, the construct $\texttt{parBlk}\ lk\ x \leftarrow gen\ \{s\}$ indicates a parallel block of code, where *gen* copies of the statement s (in the Low-- IL) can be run in parallel according to the loop annotation *lk*. This corresponds to launching *gen* threads on the GPU. The syntax $e_{res} = \texttt{sumBlk}\ e_0\ x \leftarrow gen\ \{s\ ;\ \texttt{return}\ e\}$ indicates a summation block, where the body s maps some computation across *gen* and returns the expression e. The result is summed with e_0 as the initial value and is assigned to the expression in e_{res}. Hence, this corresponds to a GPU map-reduce. The con-

struct `loopBlk` $x \leftarrow$ *gen* $\{b\}$ loops a block b. Thus, this corresponds to launching *gen* parallelized computations encoded in b in sequence. The construct `seqBlk` $\{s\}$ encodes a sequential block of code consisting of a statement s. This corresponds to the absence of parallelism.

6.4.4 Parallelizing Code

To parallelize the body of a declaration in the Low-- IL, the compiler first translates it into the Blk IL. Every top-level loop we encounter in the body is converted to a parallel block with the same loop annotation. The remaning top-level statements that are not nested within a loop are generated as a sequential block. Loop blocks and summation blocks are not generated during the initial translation step, but are generated as the result of additional transformations. The current parallelization strategy is a proof-of-concept that illustrates how to reify the parallelism specific to the base MCMC updates the compiler generates and we expect that there are many opportunities for improvement here. We summarize some optimizations that we have found useful in the context of the MCMC algorithms generated by AugurV2.

Commuting loops As a reminder, AugurV2 compiles at runtime so it has access to the sizes of the data and parameters. It can use this information to commute IL blocks of the form

```
parblk Par (k <- 0 until K) {
  loop Par (n <- 0 until N) {
    ...
  }
}
```

when $K \ll N$ so that the code utilizes more GPU threads.

INLINING The compiler inlines primitive functions that are implemented with loops. For example, the compiler can inline the sampling of a Dirichlet distribution, which samples a vector of Gamma distributed variables and then normalizes the vector (with `normalize`).

```
sample_dirichlet(alpha) {
  loop Par (v <- 0 until V) {
    x[v] = Gamma(alpha).samp;
  }
  normalize(x);
  ret x;
}
```

Inlining expose additional sources of parallelism. For example, if we inline the sampling of the Dirichlet distribution `sample_dirichlet` above in a `ParBlk`, then the loops may be commuted.

CONVERSION TO SUMMATION BLOCKS Lastly, the compiler analyzes parallel blocks marked atomic parallel to see which should be converted to summation blocks. To illustrate this more concretely, suppose we have the following code produced by AD:[4]

```
parBlk AtmPar (n <- 0 until N) {
  adj_var += adj_ll * Normal(y[n], 0, var).grad3;
}
```

Parallelizing this code by launching N threads leads to high contention in updating the variable `adj_var`. Instead, the compiler estimates the contention rate as the ratio of the number of threads we are parallelizing with (*i.e.*, N in this example) compared to the number of loca-

[4]This code could arise from a model with density factorization $p_{\text{Exp}}(\sigma^2)\, p_{\mathcal{N}}(x_n^* \mid 0, \sigma^2)$ where $p_{\text{Exp}}(\sigma^2)$ is an Exponential distribution and $p_{\mathcal{N}}(x_n^* \mid 0, \sigma^2)$ is a Normal distribution with variance σ^2. Importantly, any model where there is a higher-level parameter (*e.g.*, the variance parameter) that controls the shape of lower-level distributions (*e.g.*, the likelihood), will result in gradient code of this form.

tions the atomic additions are accessing (*i.e.*, 1 in this example). If the ratio is high as it is in this example ($N/1$), then the compiler converts it to a summation block.

```
adj_var = sumBlk adj_var (n <- 0 until N) {
  t = adj_ll * Normal(y[n], 0, var).grad3;
  return t;
}
```

Importantly, the compiler is invoked at runtime so the symbolic values can be resolved.

Once the compiler has translated the body into the Blk IL and performed the optimizations above, it will generate Cuda/C code. The Blk IL maps in a straightforward manner onto Cuda/C code. In general, such a compilation strategy will generate multiple GPU kernels for a single Low-- declaration.

6.4.5 Synthesizing a Complete MCMC Algorithm

In this step, the compiler eliminates the Kernel IL and synthesize a complete MCMC algorithm. More concretely, the compiler generates code to compute the *acceptance ratio* (AR) associated with each base MCMC update.

Recall that a base MCMC update targeting the distribution $p(x)$ can be thought of as a MH algorithm with a proposal $q(x \to x')$ of a specific form. To ensure that the distribution $p(x)$ is sampled from appropriately, the proposals are taken (or *accepted*) with probability

$$\alpha(x, x') = \min\left(1, \frac{p(x')q(x \to x')}{p(x)q(x' \to x)}\right),$$

where $\alpha(x, x')$ is known as the AR. If the proposal is not taken, it is said to be *rejected*. Some base MCMC updates such as Gibbs updates are always accepted, *i.e.*, have AR $\alpha(x, x') = 1$ for any x and x'. Thus, the acceptance ratio does not need to be computed for such updates.

Other MCMC updates such as HMC updates require the computation of the AR. The compiler generates code that computes the AR after every base update that requires it. Because such base updates can be rejected, the compiler maintains two copies of the MCMC state space, one for the current state and one for the proposal state, and enforces the invariant that the two are equivalent after the execution of a base MCMC update. The invariant ensures that the execution of a base MCMC update always uses the most current state.

6.5 System Implementation

In this section, we summarize AugurV2's system implementation, which includes the compiler, the user interface, and the runtime library.

6.5.1 Compiler implementation

The AugurV2 compiler is written in Haskell. The frontend is roughly 950 lines of Haskell code, the middle-end is roughly 1860 lines, and the backend is roughly 3420 lines. The entire compiler is roughly 9000 lines, which includes syntax, pretty printing, and type checking. We found the backend to be the most tedious to write, particularly the details of memory transfer between the Python interface and Cuda/C.

6.5.2 Runtime library

The AugurV2 runtime library is written in Cuda/C. It provides functionality for primitive functions, primitive distributions, additional MCMC library code, and vector operations. The libraries are written for both CPU and GPU inference. We believe there is room for improvement, particularly in the GPU inference libraries. For example, in the GMM example, we

need to perform the same matrix operation on many small matrices in parallel. In contrast, the typical GPU use case is to perform one matrix operation on one large matrix.

The runtime representation of AugurV2 vectors are flattened. That is, AugurV2 supports vectors of vectors (*i.e.*, ragged arrays) in its surface syntax, but the eventual representation of the data will be contained in a flattened, contiguous region of memory. This enables us to use a GPU efficiently when we want to map an operation across all the data in a vector of vectors, without following a pointer-directed structure. For this reason, the runtime representation of vectors of vectors pairs a separate pointer-directed structure with a flattened contiguous array holding the actual data. The former provides random access capabilities, while the latter enables an efficient mapping operation across the data structure. The flattened representation is also beneficial for CPU inference algorithms because of the increased locality.

6.6 Evaluation

In this section, we evaluate the design of AugurV2's compiler in terms of its (1) extensibility and (2) inference capabilities. We also compare against Jags (a variant of Bugs) and Stan, systems with similar modeling languages.

6.6.1 Extensibility

As MCMC methods are improved, it is important to design systems that can be extended to incorporate the latest advances. In this part of the evaluation, we comment on the extensibility of AugurV2's design in supporting base MCMC updates.

To support a new base update, we need to (1) add a node to the Kernel IL AST and modify the parser, (2) extend common Kernel IL operations to support the new node, and (3)

implement the Cuda/C code for the base update against the AugurV2 runtime. From experience, we have found the first two items to be simple, provided that the base update uses the MCMC primitives already implemented. For instance, once we have an implementation of AD, we can easily support both reflective Slice sampling and HMC using the same AD transformation. In contrast, supporting Gibbs updates were difficult because we need to implement a separate code-generator for each conjugacy relation. Fortunately, there is a well-known list of conjugacy relations so this can be done once. The third item is between 0 lines of C code (for a Gibbs update) to 30 lines of C code (*e.g.*, an implementation of Leapfrog integration for the HMC update) depending on the complexity of the base update. We have found that it often takes on the order of days to add a new base update, where most of the time is spent understanding the statistics and implementing the C code.

6.6.2 Performance

In this part of the evaluation, we compare AugurV2 against Jags and Stan, concentrating on how design choices made in the AugurV2 system–(1) compositional MCMC inference, (2) compilation, and (3) parallelism–compare against those made in Jags and Stan. We consider three probabilistic models commonly used to assess the performance of PPLs (*e.g.*, see the Stan user manual [92]): (1) Hierarchical Logistic Regression (HLR), (2) Hierarchical Gaussian Mixture Model (HGMM), and (3) Latent Dirichlet Allocation (LDA). We ran all the experiments on an Ubuntu 14.04 desktop with a Core-i7 CPU and Nvidia Titan Black GPU.

MODELS The generative model for a HLR is summarized below. It is a model that can be used to construct classifiers.

$$\sigma^2 \sim \text{Exponential}(\lambda)$$

$$b \sim \text{Normal}(0, \sigma^2)$$

$$\theta_k \sim \text{Normal}(0, \sigma^2)$$

$$y_n \sim \text{Bernoulli}(\text{sigmoid}(x_n \cdot \theta_k + b))$$

The generative model for a HGMM is summarized below. It is a model that clusters points on D-dimensional Euclidean space.

$$\pi \sim \text{Dirichlet}(\alpha)$$

$$\mu_k \sim \text{Normal}(\mu_0, \Sigma_0)$$

$$\Sigma_k \sim \text{InvWishart}(\nu, \Psi)$$

$$z_n \sim \text{Categorical}(\pi)$$

$$y_n \sim \text{Normal}(\mu_{z_n}, \Sigma_{z_n})$$

The generative model for LDA is summarized below. It is a model that can be used to infer

topics from a corpus of documents.

$$\theta_d \sim \text{Dirichlet}(\alpha)$$

$$\phi_k \sim \text{Dirichlet}(\beta)$$

$$z_{dj} \sim \text{Categorical}(\theta_d)$$

$$w_{dj} \sim \text{Categorical}(\phi_{z_{dj}})$$

Compositional MCMC To assess the impact of generating compositional MCMC algorithms, we use the HLR model and the HGMM model. The HLR model contains only continuous parameters. Hence, a system such as Stan, which is specifically designed for gradient-based MCMC algorithms, should peform well. The HGMM model is fully-conjugate. Hence, a system such as Jags, which is specifically designed for Gibbs sampling, should perform well.

For the HLR model, we visually verified the trace plots (*i.e.*, a plot of the values of the parameter from one sample to the next) of each system. On the German Credit dataset [58], we found that AugurV2 configured to generate a CPU HMC sampler with manually picked parameters to be roughly 25 percent slower than Stan set to use the same HMC sampling algorithm in generating 1000 samples (with no thinning). It takes roughly 35 seconds for Stan to compile the model (due to the extensive use of C++ templates in its implementation of AD), whereas AugurV2 compiles almost instantaneously. It would be interesting to see how if there are additional optimizations that can be applied during compilation to improve AugurV2's implementation of AD. Although fast compilation times are not an issue in our setting where we just need to compile the model (and query) once, this may become more of an issue in structure learning [**?**]. Jags had the poorest performance as it defaults to adaptive rejection

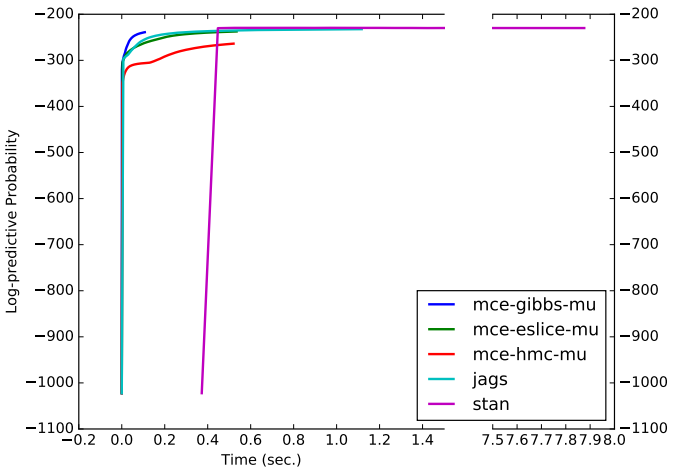


Figure 6.12: The log-predictive probability of a HGMM when using the samplers generated by AugurV2, Jags, and Stan. We use AugurV2 to generate 3 different MCMC inference algorithms. We draw 150 samples using each system with no thinning. Stan requires an initial tuning period of 50 samples.

sampling.

FLEXIBLE INFERENCE Compared to both Jags and Stan, AugurV2 has more flexible inference. For example, in a HLR, we can update the variance (σ^2) and offset (b) parameters with a Gaussian proposal centered at the current location, while using the more computationally intensive HMC to sample the coefficients (θ). Indeed, this corresponds to HMC practice, where we use simpler MCMC methods to update higher-level parameters [18]. In contrast, Jags will perform adaptive rejection sampling on all variables and Stan will apply a variant of HMC called No-U-Turn to the entire model.

Figure 6.12 contains plots of the log-predictive probability versus training time for a $2D$-HGMM model with 1000 synthetically-generated datapoints and 3 clusters. A log-predictive probability plot can be seen as a proxy for learning–as training time increases, the algorithm should be able to make better predictions. We configured AugurV2 to generate 3 different

(K, D, N)	AugurV2	Jags	Speedup
$(3, 2, 1000)$	0.2	1.1	$\sim 5.5x$
$(3, 2, 10000)$	1.4	17.4	$\sim 12.4x$
$(10, 2, 10000)$	3.7	51.5	$\sim 13.9x$
$(3, 10, 10000)$	15.6	93.0	$\sim 5.9x$
$(10, 10, 10000)$	17.8	301.9	$\sim 16.9x$

Figure 6.13: Approximate timing results comparing the performance of AugurV2's compiled Gibbs sampler versus Jag's Gibbs sampler on a HGMM with varying clusters (K), dimensions (D), and datapoints (N).

MCMC samplers corresponding to sampling the cluster locations with Elliptical Slice updates, Gibbs updates, and HMC updates. The plot shows that every system converges to roughly the same log-predictive probability, the difference being the amount of time it takes. For instance, Jags and AugurV2's Gibbs sampler have better computational performance because they can leverage the conjugay relation, whereas Stan uses gradient-based MCMC.

Compilation For the HLR model, which contains only continuous variables, we set AugurV2 to generate a HMC sampler for the entire model and compare to Stan's implementation. Jags cannot generate a HMC sampler and so we do not compare against it. On the German Credit dataset [58], we found AugurV2 to be roughly 25 percent slower than Stan in generating 1000 samples. Note that Stan is specifically tailored to generate gradient-based MCMC and it takes roughly 35 seconds to compile the model, whereas AugurV2 generates more flexible inference algorithms and compiles almost instantaneously. As the cost of HMC is dominated by the cost of computing the gradient, we hope that the performance of AugurV2's AD implementation as well as of the runtime library can be improved by leveraging ideas from the implementation of Stan, but we leave that for future work.

Figure **??** summarizes the timing results to generate 150 samples for a HGMM on a synthetically generated dataset for a varying number of clusters, dimensions, and datapoints.

We set AugurV2 to generate a Gibbs update for all the variables and compare against Jags' Gibbs sampler so that both are running the same high-level inference algorithm. The difference is that Jags reifies the Bayesian network structure and performs Gibbs sampling on the graph structure, whereas AugurV2 directly generates code that performs Gibbs sampling using symbolically computed conditionals. The experiments show that AugurV2's approach outperforms Jags' approach. We also compared to the performance of Stan as well as checked the log-predictive probabilities for all three systems, but didn't include timing results for Stan because they aren't particularly meaningful. For instance, we applied Stan's No-U-Turn sampler to the largest model with 10 clusters in 10 dimensions, for which it takes approximately 19 hours to draw 150 samples. Notably, Stan does not natively support discrete distributions so the user must write the model to marginalize out all discrete variables, which increases the complexity of computing gradients. This highlights the importance of being able to support compositional MCMC algorithms.

Parallelism Jags and Stan support parallel MCMC by running multiple copies of a chain in parallel. In contrast, AugurV2 supports parallel MCMC by parallelizing the computations *within* a single chain. As these methods are not comparable, we will focus on AugurV2's GPU inference capabilities. For these experiments, we will use all three models. We found that GPU inference can improve scalability of inference, but it is highly model-dependent.

For example, when we fit the HLR to the German Credit dataset using AugurV2's GPU HMC sampler, the computational performance was roughly an order of magnitude worse compared to AugurV2's CPU HMC sampler. This can be attributed to the small dataset size (roughly 1000 points) and the low dimensionality of the parameter space (26 parameters). When we apply the GPU HMC sampler to the Adult Income dataset [58], which has roughly

Dataset-Topics	CPU (sec.)	GPU (sec.)	Speedup
Kos-50	159	60	$\sim 2.7x$
Kos-100	265	73	$\sim 3.6x$
Kos-150	373	82	$\sim 4.6x$
Nips-50	504	161	$\sim 3.1x$
Nips-100	880	168	$\sim 5.2x$
Nips-150	1354	235	$\sim 5.8x$

Figure 6.14: Approximate timing results comparing the performance of AugurV2's CPU Gibbs inference against GPU Gibbs inference for LDA. The Kos dataset [58] has a vocabulary size of 6906 and contains roughly $460k$ words. The Nips dataset [58] has a vocabulary size of 12419 and roughly 1.9 million words.

50000 observations and 14 parameters, the gradients were parallelized differently due to the summation block optimization–it is more efficient to run 14 map-reduces over 50000 elements as opposed to launching 50000 threads all contending to increment 14 locations.

In contrast to a model such as a HLR, models such as HGMM and LDA have much higher dimensional spaces. Indeed, the number of latent variables scales with the number of data-points. In these cases, GPU inference was much more effective. Figure 6.14 summarizes the (approximate) timing results for AugurV2's performance on LDA across a variety of datasets, comparing its CPU Gibbs performance against its GPU Gibbs performance. We also checked the log-predictive probabilities [cite] to make sure that they were roughly the same for the CPU and GPU samplers. The general trend is that the GPU provides more benefit on larger datasets, with larger vocabulary sizes, and with more topics. We have also tried this with the HGMM and found the same general trend. We could not get Jags or Stan to scale to LDA, even for the smallest dataset.

6.7 Related Work

The design of AugurV2 builds upon a rich body of prior work on PPLs. In this section, we compare AugurV2's design with related PPLs, using the (*model*, *query*, *inference*) tuple (*i.e.*, the probabilistic modeling and inference tuple) as a guide. At a high-level, PPL modeling languages can be classified according to whether they (1) express well-known probabilistic modeling abstractions such as Bayesian networks [19, 94] as in AugurV2 or (2) are embedded into general-purpose programming languages [**?** 73, 39**?** , 71]. Most PPLs provide fixed queries and inference strategies, although languages such as Edward [98] and Venture [61] explore more expressive query and inference strategies. We discuss related systems in more depth now.

6.7.1 Related Systems

Language: Bugs To our knowledge, Bugs was one of the first probabilistic programming systems designed for Bayesian probabilistic modeling [94, 60]. The original BUGS language provides an imperative modeling language for expressing Bayesian networks and uses Gibbs sampling to explore posterior distributions. Hence, it supports fixed queries with a fixed inference strategy.

The AugurV2 modeling language is inspired by the BUGS modeling language. Both languages are first-order and are designed to be closer to mathematical notation. However, AugurV2 provides parallel comprehensions instead of an imperative `for` loop that Bugs provides. Inference in the BUGS systems is performed by reifying the Bayesian network corresponding to a model described in the language, and then querying the network structure at runtime to figure out which variables are conditionally independent of others to perform Gibbs updates.

In contrast, AugurV2 approximates the conditional independence at compile-time and uses this information to directly generate code that samples conditionally independent variables in parallel. Hence, the runtime representation of an AugurV2 inference algorithm is much more compact and regular.

Language: Stan Stan [19] is another PPL similar in design to both BUGS and AugurV2. Again, the major difference between the Stan language and AugurV2 language is that AugurV2 uses parallel comprehensions. A minor difference is that Stan does not support vectors of vectors so users must flatten their models. Moreover, Stan does not natively support discrete random variables as its primary method of inference uses No-U-Turn sampling, a variant of HMC, which works only on continuous parameter spaces. The latest version of Stan supports Automatic Differentiable Variational Inference. Hence, Stan supports several fixed inference strategies. In contrast, AugurV2 provides a family of inference algorithms based on MCMC sampling.

Factor graph approachs Several other probabilistic programming systems use factor graphs as the basis of inference, including Infer.net [65], Factorie [62], and Dimple [43]. Both Factorie and Dimple provide a library API for constructing factor graphs. Infer.net provides a surface level language that can be compiled into a factor graph for the purposes of inference. We consider this approach somewhat less-declarative as the user needs to manually construct the factor graph. These systems implement inference strategies that can take advantage of the factor graph structure such as message-passing algorithms. Again, the difference with AugurV2 is mainly that these systems are limited to uniformly applying a single inference method to the entire model. Nevertheless, they support several fixed inference strategies.

Symbolic approaches The Hakaru language embeds probabilistic programming in Haskell, and hence, provides a much richer modeling language [2]. Nevertheless, users should not use Haskell features such as recursion or IO when inside the Hakaru subset. Hence, the expressivity of Hakaru is similar to that of modeling languages such as AugurV2. Hakaru implements posterior inference symbolically, *i.e.*, it performs exact inference. Like AugurV2, Hakaru has a language for representing inference algorithms that can be analyzed and optimized. Unlike AugurV2, the language Hakaru uses is much richer. In fact, the inference language is the modeling language itself. Consequently, Hakaru can express more complicated inference algorithms instead of AugurV2's strategy of composing base MCMC kernels. Indeed, the Hakaru system interacts with a computer algebra system (CAS) to derive facts useful for posterior inference. In contrast, the Hakaru representation does not have access to the lower-level details of memory management and order-of-evaluation. Hakaru is also not compiled.

The Psi system [34] is similar to Hakaru in that it performs symbolic inference. Unlike Hakaru, Psi identifies a core subset of mathematical transformations that useful for solving integrals arising in Bayesian inference practice, instead of appealing to a more general CAS. By leveraging the extra structure, the Psi system has been demonstrated to handle larger models than Hakaru. The Psi system handles loops by unfolding, which leads to scalability issues. In summary, these languages explore the feasibility of supporting exact inference queries on a restricted modeling language. It would be interesting to see how these techniques can be integrated into the implementation of AugurV2. Notably, the Psi system has uses an intermediate representation similar to AugurV2's Density IL.

General-purpose embeddings Another family of PPLs adds to a general, Turing-complete language (1) sampling and (2) probabilistic queries. The idea is that standard programming

languages are already themselves a *good* abstraction for expressing probabilistic models. Hence, languages in this family embrace traditional programming language constructs, such as higher-order functions, recursion, state, and etc. The challenge once programs are proposed as an alternative to traditional abstractions (*e.g.*, probabilistic graphical models) is to develop efficient inference techniques.

The earliest of these languages such as Stochastic Lisp [53] and IBAL [78, 79] add discrete distributions. As only discrete distributions are involved, inference can be performed by summations. This includes techniques such as variable elimination. However, many practically useful models require *continuous* distributions. Once *continuous* distributions are added to a programming language, new inference techniques need to be developed.

The Church [39] probabilistic programming language is embedded in Scheme and proposes MCMC inference on *program traces*. Church restricts primitive distributions to only those that have densities, called *exchangeable random primitives*, but otherwise allows the full power of the Scheme language to build models. Thus, each execution trace of a Church program can be associated with the likelihood of the random choices made along that trace. Consequently, we can perform MCMC sampling on these traces. The challenge is to develop efficient inference in the space of program traces. The first version of Church implemented MH sampling with default proposals. This approach does not scale to more complex models in higher-dimensional parameter spaces.

Other trace-based MCMC systems have been developed, including Anglican [105] and R2 [71]. The former proposes particle MCMC approaches and show that this approach can leverage OS-level concurrency. As a reminder, a particle MCMC estimates the posterior distribution via a weighted set of samples called *particles* (on the order of 10 particles). Anglican

leverages OS-level concurrency to track each particle. The latter improves the efficiency of MH sampling by using static analysis (*i.e.*, weakest precondition calculation) to rule out impossible states, and thus, craft tailored proposals.

LANGUAGE: BLAISE The Blaise [16] system provides a domain-specific graphical language for expressing both models and MCMC algorithms. Hence, such a system provides a more expressive language for encoding inference algorithms beyond choosing from a preselected and fixed number of strategies. As we mentioned in the introduction, the Kernel IL used by AugurV2 is closely related to the Blaise language. In particular, it should be possible to translate the Kernel IL instantiated with the Density IL to a Blaise graph. We use the Kernel IL for the purposes of compilation, whereas in Blaise the language is interpreted. It would be an interesting direction of future work to see if the subset of the Blaise language related to inference can also be used for the purposes of compilation.

FLEXIBLE INFERENCE Edward [98] and Venture [61] also explore increasing the expressivity of different query and inference languages. Edward is built on top of Tensor-flow [3], and hence, provides gradient-based inference strategies such as HMC and AVDI. Notably, the Tensor-flow system efficiently supports the computation of gradients for an input computational graph (*e.g.*, the computational graph of the log-likelihood calculation for a probabilistic model). Consequently, Edward leverages the benefits of Tensor-flow. It would be interesting direction for future work to see how much of Tensor-flow's infrastructure for AD can be used in AugurV2 as done in Edward. However, note that AugurV2 provides higher-level functionality than Tensor-flow. For example, AugurV2 can generate non-gradient-based MCMC algorithms (*e.g.*, Gibbs samplers) as well as compose gradient-based MCMC with non-gradient-

based MCMC. Venture [61] provides an inference language that is closer to a general-purpose programming language, and hence, explores the more expressive regime of the design space for inference algorithms.

Other modeling languages The probabilistic programming languages we have focused on so far have largely been functional. Hence, they are most suited for expressing models with directed dependencies as opposed to undirected depedencies (*e.g.*, with cycles). In this section, we review other language designs that are not as closely related to AugurV2 as the ones we presented previously, but nevertheless, are included here to more broadly survey the landscape of probabilistic modeling languages.

For instance, probabilistic logic programming adds probabilistic operations to a logic programming language. Intuitively, the idea is to replace hard, logical entailments with soft, probabilistic entailments. Hence, this approach attempts to combine logic with probability theory.

Markov Logic Networks (MLNs) provide another logic-programming approach that combines first-order logic with Markov Random Fields [23]. The user provides a collection of first-order formula, their weights, and a set of ground terms (*i.e.*, atoms). This results in a possible-world semantics, where intuitively, the probability of a possible world is proportional to the weighted number of first-order formula it violates. Thus, MLNs do not describe generative processes, which exhibit casual dependency structures, but relational dependency structures. A MLN can answer questions of the form: "what is the probability that a formula holds, given another formula holds?" Hence, MLNs subsume basic probabilistic inference where we obtain a posterior distribution on numerical parameters given numerical data. As a result, MLNs are typically solved with optimization techniques to obtain point esti-

mates, rather than Bayesian techniques that return distributions. Tractable Markov Logic (TML) is a variation of MLN that restricts the structure of first-order formulae in order to make inference more tractable [24]. This is reminiscent of the restricted language design we saw in AugurV2 to make inference more tractable in practice as well. There are other languages that attempt to combine logic with probability such as Stochastic Logic Programs [66], BLOG [63, 64] and Problog [22], but we refer the reader to work on MLN for a more comprehensive discussion.

Tabular [40], as its name suggests, explores how to use schemas (*e.g.*, databases or spreadsheets at a first approximation) to express probabilistic models queries. It falls at an interesting point in the design space because most modeling languages use traditional programming language syntax. A schema naturally expresses missing values in an observed dataset. Indeed, a common problem encountered in probabilistic modeling in practice is how to treat incomplete datasets.

Implementation techniques In addition to exploring various points of the (*model*, *query*, *inference*) tuple space, many PPLs have also proposed interesting implementation decisions that are related to AugurV2's. For example, the Hierarchical Bayes Compiler (HBC) [48] explores the compilation of Gibbs samplers to C code. In this work and our previous work on Augur, we support compilation of Gibbs samplers to the GPU. Swift [106] uses a compiler pass to optimize how conditional independence relationships are tracked at runtime for a modeling language that can express dynamic relationships. In contrast, AugurV2 provides a simpler modeling language that expresses static relationships, and hence, statically approximates these relationships. Bhat *et al.* [**?**] give a proof of correctness for a compiler that transforms a term in a first-order modeling language into a density. This language subsumes

AugurV2's (*e.g.*, AugurV2 does not provide branching constructs) and can be seen as justifying the implementation of the AugurV2 frontend, which translates a term in the modeling language into its density factorization.

Related language designs We would like to briefly mention some other work outside of the probabilistic programming literature that the design of AugurV2 draws from. For example, Halide is a DSL for expressing image processing algorithms that separates the specification of the intrinsic image processing algorithm from the schedule, *i.e.*, the storage and order of execution [82]. Similarly, AugurV2 is a DSL for expressing parametric Bayesian networks that separates the specification of the model from the MCMC schedule.

7

The AugurV2 Language

In this chapter, we formalize AugurV2's modeling language and show how the semantic foundation based on computable distributions developed in the first part of the dissertation can guide the design and implementation of a PPL. For the purposes of giving semantics, we first define an expression-based language called Core AugurV2 that we can desugar programs in AugurV2's surface-level modeling language into (Sections 7.1 and 7.2). Then, we show that every Core AugurV2 program of full-measure denotes a distribution that admits a density factorization (Section 7.3). Importantly, this justifies the implementation of inference in

Core AugurV2 via MCMC (a method which requires densities). Finally, we show that conditioning on a well-formed distribution (*i.e.*, a full measure distribution) in Core AugurV2 is (Type-2) computable (Section 7.4). In particular, an error such as an out-of-bounds access can produce a sub-probability distribution. Hence, the semantics of conditioning on these distributions is undefined.

7.1 A Core Language

In this section, we define a language called Core AugurV2 that AugurV2's surface-level modeling language can be desugared into. The syntax for Core AugurV2 is summarized in Figure 7.1. Unlike the random variable notation used in AugurV2's modeling language, Core AugurV2 is an expression-based language that incorporates distributions using a probability monad. It will be easier to see how the semantics of this language can be given by compilation to λ_{CD} (Chapter 4) because the languages are structurally similar. The desugaring translation is straightforward, and hence, omitted. Figure 7.2 provides an example of how a GMM (see Figure 5.3 for the GMM expressed in random variable notation) is desugared.

Syntax We use an expression-based language with a probability monad to describe the generative model. In particular, we can use three syntactic constructs to build complex distributions from primitive distributions *dist*. First, the syntax $x \leftarrow e_1 \;;;\; e_2$ is like an ordinary bind from the probability monad that additionally returns the value bound to x. Hence, we call it a *product bind expression*. Second, the syntax $x = e_1 \;;;\; e_2$ corresponds to a standard let expression. Third, the syntax $\{e \mid x \leftarrow gen\}_D$ is a n-fold product bind expression, which can be used to express models with repetitive structure. The semantics of a product bind expression are independent of the order of evaluation. Nevertheless, note that this does not imply that

$$
\begin{aligned}
\mathit{model} &::= \lambda \overrightarrow{x}.\, e_h \leftarrow e_o \;;;\; \texttt{obs} \\
\tau &::= \texttt{Int} \mid \texttt{Real} \mid \texttt{Real}|_{\texttt{pt}} \mid \tau \times \tau \mid \texttt{Vec}\,\tau \mid \overrightarrow{\tau} \to \tau \mid \texttt{Dist}\,\tau \\
e &::= x \mid i \mid r \mid f(\overrightarrow{e}) \mid e[e] \mid \mathit{dist}(\overrightarrow{e}) \\
&\quad \mid x \leftarrow e \;;;\; e \mid x = e \;;;\; e \mid \{e \mid x \leftarrow \mathit{gen}\}_D \\
\mathit{gen} &::= g\ \texttt{until}\ g \\
g &::= x \mid i \mid g[g]
\end{aligned}
$$

Figure 7.1: The syntax for Core AugurV2, an expression-based language with a probability monad `Dist` τ.

```
\(D, K, N, mu0, Sigma0, pis, Sigma).
  obs ( mu <- { MvNorm(mu0, Sigma0) | k <- 0 until K } ;;
        z <- { Disc(pis) | n <- 0 until N }
      ) (\(mu, z). { MvNorm(mu[z[n]], sigma) | n <- 0 until N })
```

Figure 7.2: A Core AugurV2 model encoding a GMM (see Figure 5.3 for the the GMM expressed in random variable notation).

the distributions specified with this construct are necessarily independent. For example, the random variables corresponding to the datapoints for a GMM in Figure 7.2 are expressed with a product bind expression, but are only conditionally independent of each other given the appropriate random variables (and not independent of each other). Comprehension bounds *gen* are specified with a restricted subset of expressions g, which includes (1) variables x, (2) constant integers i, and (3) vector projections $g[g]$ (which can be used to encode ragged arrays). A complete Core AugurV2 model is a function $\lambda \overrightarrow{x}.\, y \leftarrow e_p \;;;\; \texttt{obs}\, e_s$ from model hyperparameters and constants $\overrightarrow{x}$ to an expression $y \leftarrow e_h \;;;\; \texttt{obs}\, e_o$ describing the hidden (e_h) and observed (e_o) parts of a generative model.

TYPES Base types include integers `Int`, reals `Real`, and reals restricted to a discrete point $\texttt{Real}|_{\texttt{pt}}$. The reason we have different types for reals `Real` and restricted reals $\texttt{Real}|_{\texttt{pt}}$ is to

specify the base distribution that a Core AugurV2 model has density with respect to. Informally, we have the subtyping relation $\texttt{Int} \leq \texttt{Real}$ and $\texttt{Real}|_{\texttt{pt}} \leq \texttt{Real}$. Compound types include products $\tau_1 \times \tau_2$ and vectors $\texttt{Vec}\ \tau$. Primitive functions are assigned function types $\overrightarrow{\tau} \to \tau$. Finally, distributions are assigned the type $\texttt{Dist}\ \tau$.

STATICS Figure 7.3 summarizes Core AugurV2's type system. The judgement $\vdash_D\ \tau$ checks that distribution types are well-formed. The expression typing judgement $\Gamma \vdash e : \tau$ checks expressions. Finally, the judgement $\Psi \vdash \mathit{model}$ checks that a whole model is well-formed, under a context Ψ that contains the types of primitive distributions and functions. We walk through each of the judgements in turn.

The judgement $\vdash_D\ \tau$ is used to restrict the types that can be supplied to a distribution type. For instance, AugurV2 allows distributions on base types $\texttt{Int}$, $\texttt{Real}$, and $\texttt{Real}|_{\texttt{pt}}$, as well as products $\tau_1 \times \tau_2$ and $\texttt{Vec}\ \tau$ of well-formed distribution types. These correspond to familiar distributions on integers and reals, and (n-fold) products of these distributions.

The judgement $\Gamma \vdash e : \tau$ types expressions and is mostly standard. We review the typing rules for expressions that work with distributions. To type the expression $x \leftarrow e_1\ ;;\ e_2$, or a product bind expression, we check that e_1 is a distribution $\texttt{Dist}\ \tau_1$. Then, we check that e_2 is a distribution $\texttt{Dist}\ \tau_2$ under the additional assumption that x has type τ_1. Additionally, we check that τ_1 and τ_2 are well-formed distribution types. The result is the product distribution type $\texttt{Dist}\ \tau_1 \times \tau_2$. To type the distribution comprehension expression $\{e \mid x \leftarrow \mathit{gen}\}_D$, we check that the body e under the assumption that x is an integer has the distribution type $\texttt{Dist}\ \tau$ and that gen is a well-formed generator. This produces a distribution on the vector space $\texttt{Dist}\ (\texttt{Vec}\ \tau)$.

The typing rule for the let sampling expression $x = e_1\ ;;\ e_2$ is similar to that of the rule

$\boxed{\vdash_D \tau}$ Well-formed distribution type

$$\frac{}{\vdash_D \texttt{Int}} \qquad \frac{}{\vdash_D \texttt{Real}} \qquad \frac{}{\vdash_D \texttt{Real}|_{\texttt{pt}}} \qquad \frac{\vdash_D \tau_1 \qquad \vdash_D \tau_2}{\vdash_D \tau_1 \times \tau_2} \qquad \frac{\vdash_D \tau}{\vdash_D \texttt{Vec}\ \tau}$$

$\boxed{\Gamma \vdash g : \tau}$ Generator expression judgement

$$\frac{\Gamma(x) : \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash i : \texttt{Int}} \qquad \frac{\Gamma \vdash g_1 : \texttt{Vec}\ \tau \qquad \Gamma \vdash g_2 : \texttt{Int}}{\Gamma \vdash g_1[g_2] : \tau}$$

$\boxed{\Gamma \vdash gen : \tau}$ Generator judgement

$$\frac{\Gamma \vdash g_1 : \texttt{Int} \qquad \Gamma \vdash g_2 : \texttt{Int}}{\Gamma \vdash g_1\ \texttt{until}\ g_2 : \texttt{Vec Int}}$$

$\boxed{\Gamma \vdash e : \tau}$ Expression judgement

$$\frac{\Gamma(x) : \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash i : \texttt{Int}} \qquad \frac{}{\Gamma \vdash r : \texttt{Real}}$$

$$\frac{\Gamma(f) = \tau_1 \times \cdots \times \tau_n \to \tau \qquad \Gamma \vdash e_i : \tau_i \text{ for } 1 \leq i \leq n}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau} \qquad \frac{\Gamma \vdash e_1 : \texttt{Vec}\ \tau \qquad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1[e_2] : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Dist}\ \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \texttt{Dist}\ \tau_2 \qquad \vdash_D \tau_1 \qquad \vdash_D \tau_2}{\Gamma \vdash x \leftarrow e_1\ ;;\ e_2 : \texttt{Dist}\ (\tau_1 \times \tau_2)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \texttt{Dist}\ \tau_2 \qquad \vdash_D \tau_1 \qquad \vdash_D \tau_2}{\Gamma \vdash x = e_1\ ;;\ e_2 : \texttt{Dist}\ (r(\tau_1) \times \tau_2)}$$

$$\frac{\Gamma, x : \texttt{Int} \vdash e : \texttt{Dist}\ \tau \qquad \Gamma \vdash gen : \texttt{Vec Int} \qquad \vdash_D \tau}{\Gamma \vdash \{e \mid x \leftarrow gen\}_D : \texttt{Dist}\ (\texttt{Vec}\ \tau)}$$

$\boxed{\Psi \vdash model}$ Model judgement

$$\frac{\Psi, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_h : \texttt{Dist}\ \tau_h \qquad \Psi, x_1 : \tau_1, \ldots, x_n : \tau_n, y : \tau_h \vdash e_o : \texttt{Dist}\ \tau_o}{\Psi \vdash \lambda(x_1, \ldots, x_n).\, y \leftarrow e_h\ \texttt{;;}\ \texttt{obs}\, e_o : \tau_o \to \texttt{Dist}\ \tau_h}$$

Figure 7.3: Core AugurV2's type system.

$$
\begin{aligned}
r(\texttt{Int}) &\triangleq \texttt{Int} \\
r(\texttt{Real}|_{\texttt{pt}}) &\triangleq \texttt{Real}|_{\texttt{pt}} \\
r(\texttt{Real}) &\triangleq \texttt{Real}|_{\texttt{pt}} \\
r(\tau_1 \times \tau_2) &\triangleq r(\tau_1) \times r(\tau_2) \\
r(\texttt{Vec}\ \tau) &\triangleq \texttt{Vec}\ (r(\tau))
\end{aligned}
$$

Figure 7.4: The definition of the restriction function r, a function on Core AugurV2 types, which is used in the Core AugurV2 type system to indicate the base measure a distribution has density with respect to.

for the product sample, with two differences. First, we check that e_1 has type τ_1 (instead of $\texttt{Dist}\ \tau_1$). Second, the resulting type is $\texttt{Dist}\ r(\tau_1) \times \tau_2$ and not $\texttt{Dist}\ \tau_1 \times \tau_2$. This uses the restriction function r, which operates on types and is defined by induction on well-formed distribution types. The restriction function is summarized in Figure 7.4. The idea is that when we let bind the expression e_1 to a variable x, we should consider the degenerate product space where the first component is restricted to the one-point space, instead of the entire space of values of type τ_1. This distinction will be used later when we give the density factorization of AugurV2 models.

The judgement for a well-formed model $\Psi \vdash \mathit{model}$ checks that the entire model *model* is well-formed under the context Ψ. The context Ψ contains types for primitive functions, distributions, and densities. A Core AugurV2 model is a generative model $e_h \leftarrow e_o\ \texttt{;;}\ \texttt{obs}$ whose hidden portion is given by e_h and observed portion is given by e_o. The expression is typed similarly to a product bind expression. However, the resulting type is a function from observed data τ_o to a posterior distribution $\texttt{Dist}\ \tau_h$.

```
module Vec (Vec, idx, unt, sequenceA) where
import CompDistLib

type Vec a = [a]

idx :: Vec a -> Int -> a
idx vec n | n < length vec = v !! n
          | otherwise = bot

unt :: Int -> Int -> Vec Int
unt n m | 0 <= n && n < m = { i | i <- n..m }
        | otherwise = bot

sequenceA :: Vec (Samp a) -> Samp (Vec a)
sequenceA (hd : tl) = hd >>= \hd' >>= sequenceA tl >>=
    \tl' >>= return (hd' : tl')
sequenceA [] = return []
```

Figure 7.5: Vector module for translation.

$$
\begin{aligned}
\mathcal{T}[\![e_1 \texttt{ until } e_2]\!] &\triangleq \texttt{unt}\ (\mathcal{T}[\![e_1]\!])\ (\mathcal{T}[\![e_2]\!]) \\
\mathcal{T}[\![x]\!] &\triangleq x \\
\mathcal{T}[\![i]\!] &\triangleq i \\
\mathcal{T}[\![r]\!] &\triangleq r \\
\mathcal{T}[\![f(e_1,\ldots,e_n)]\!] &\triangleq f(\mathcal{T}[\![e_1]\!],\ldots,\mathcal{T}[\![e_n]\!]) \\
\mathcal{T}[\![e_1[e_2]]\!] &\triangleq \texttt{idx}\ (\mathcal{T}[\![e_1]\!])\ (\mathcal{T}[\![e_2]\!]) \\
\mathcal{T}[\![dist(e_1,\ldots,e_n)]\!] &\triangleq dist(\mathcal{T}[\![e_1]\!],\ldots,\mathcal{T}[\![e_n]\!]) \\
\mathcal{T}[\![x \leftarrow e_1 \;;;\; e_2]\!] &\triangleq x \leftarrow \mathcal{T}[\![e_1]\!]\ ; \\
&\qquad\quad y \leftarrow \mathcal{T}[\![e_2]\!]\ ;\ \text{where } y \text{ not free in } \mathcal{T}[\![e_2]\!] \\
&\qquad\quad \texttt{return}\ (x,y) \\
\mathcal{T}[\![x = e_1 \;;;\; e_2]\!] &\triangleq x \leftarrow \texttt{return}\ \mathcal{T}[\![e_1]\!]\ ; \\
&\qquad\quad y \leftarrow \mathcal{T}[\![e_2]\!]\ ;\ \text{where } y \text{ not free in } \mathcal{T}[\![e_2]\!] \\
&\qquad\quad \texttt{return}\ (x,y) \\
\mathcal{T}[\![\{e \mid x \leftarrow gen\}_D]\!] &\triangleq \texttt{sequenceA}\ \{\mathcal{T}[\![e]\!] \mid x \leftarrow \mathcal{T}[\![gen]\!]\}
\end{aligned}
$$

Figure 7.6: Translation of AugurV2 into λ_{CD}

7.2 Semantics

In this section, we translate a Core AugurV2 expression into a λ_{CD} expression and then derive the semantics. This supports the idea that a Turing-complete probabilistic modeling language expresses computable distributions—in particular, we show now how to use one (*i.e.*, λ_{CD}) to implement an interpreter for AugurV2.

7.2.1 Translation

Figure 7.6 summarizes the translation $\mathcal{T}[\![\cdot]\!]$ of a well-typed Core AugurV2 expression into a λ_{CD} expression. In order to translate the vector type, we assume that λ_{CD} has lists and their associated operations with the usual semantics (see Figure 7.5). We focus on the translation for distribution constructs as the translations for the other constructs are largely standard.

The translation of a product sampling expression $x \leftarrow e_1 \;;;\; e_2$ uses bind from λ_{CD}. Its translation is

$$\mathcal{T}[\![x \leftarrow e_1 \;;;\; e_2]\!] \triangleq x \leftarrow \mathcal{T}[\![e_1]\!] \;;\; (y \leftarrow \mathcal{T}[\![e_2]\!] \;;\; \texttt{return}\; (x, y))\,,$$

where y is not free in $\mathcal{T}[\![e_2]\!]$. The translation of the assignment expression $x = e_1 \;;;\; e_2$ also uses bind from λ_{CD}.

$$\mathcal{T}[\![x = e_1 \;;;\; e_2]\!] \triangleq x \leftarrow \texttt{return}\; (\mathcal{T}[\![e_1]\!]) \;;\; (y \leftarrow \mathcal{T}[\![e_2]\!] \;;\; \texttt{return}\; (x, y))\,,$$

where y is not free in $\mathcal{T}[\![e_2]\!]$. To translate an expression producing a distribution on an n-fold product (*i.e.*, a vector), we use lists (see Figure 7.5). We will treat lists isomorphically with

n-fold products.

$$\mathcal{T}[\![\{e \mid x \leftarrow gen\}_D]\!] \triangleq \texttt{sequenceA}\,\{\mathcal{T}[\![e]\!] \mid x \leftarrow \mathcal{T}[\![gen]\!]\}$$

The λ_{CD} function `sequenceA` is a monadic sequence operator, *i.e.*, it binds a list of monadic values, and returns the list (see Figure 7.5). We have also taken the liberty of using comprehension syntax.

Finally, the translation of an observation construct simply calls the implementation of computable conditioning with a bounded, computable density `obsDens` from λ_{CD}. As a reminder, the function `getBndCondDens` obtains the conditional density of its argument which is a distribution.

$$\mathcal{T}[\![y \leftarrow e_h \texttt{ ;; obs } e_o]\!] \triangleq \texttt{obsDens}\,\mathcal{T}[\![e_h]\!]\,(\lambda y.\,\texttt{getBndCondDens}(\mathcal{T}[\![e_o]\!])(y))$$

7.2.2 Derived semantics

Together, the translation $\mathcal{T}[\![\cdot]\!]$ and the semantics of λ_{CD} gives a derived semantics for AugurV2 models. Below, we compute the derived semantics for Core AugurV2's distribution constructs and check that they are what we expect.

First, we calculate the semantics of a product sampling expression $x \leftarrow e_1 \;;;\; e_2$.

$$\begin{aligned}
&\mathcal{E}[\![\mathcal{T}[\![x \leftarrow e_1 \;;;\; e_2]\!]]\!]\rho(U) \\
&= \mathcal{E}[\![x \leftarrow \mathcal{T}[\![e_1]\!] \;;\; (y \leftarrow \mathcal{T}[\![e_2]\!] \;;\; \texttt{return}\;(x,y))]\!]\rho(U) \\
&= \int \bar{x} \mapsto \mathcal{E}[\![y \leftarrow \mathcal{T}[\![e_2]\!] \;;\; \texttt{return}\;(x,y)]\!]\rho[x \mapsto \bar{x}]\, d\mathcal{E}[\![e_1]\!]\rho \\
&= \int \bar{x} \mapsto \int \bar{y} \mapsto \mathbb{1}_U((\bar{x},\bar{y}))\, d\mathcal{E}[\![e_2]\!]\rho[x \mapsto \bar{x}]\, d\mathcal{E}[\![e_1]\!]\rho
\end{aligned}$$

Hence, we obtain a product distribution.

Second, we calculate the semantics of a let sampling expression $x = e_1 \;;;\; e_2$.

$$
\begin{aligned}
&\mathcal{E}[\![\mathcal{T}[\![x = e_1 \;;;\; e_2]\!]]\!]\rho(U) \\
&= \mathcal{E}[\![x \leftarrow \texttt{return}\ \mathcal{T}[\![e_1]\!]\ ;\ (y \leftarrow \mathcal{T}[\![e_2]\!]\ ;\ \texttt{return}\ (x, y))]\!]\rho(U) \\
&= \int \bar{x} \mapsto \mathcal{E}[\![y \leftarrow e_2\ ;\ \texttt{return}\ (x, y)]\!]\rho[x \mapsto \bar{x}]\, d\mathcal{E}[\![\texttt{return}\ e_1]\!]\rho \\
&= \int \bar{x} \mapsto \int v_y \mapsto \mathbb{1}_U((\bar{x}, \bar{y}))\, d\mathcal{E}[\![e_2]\!]\rho[x \mapsto \bar{x}]\, d\mathcal{E}[\![\texttt{return}\ e_1]\!]\rho \\
&= \int \bar{y} \mapsto \mathbb{1}_U((\bar{x}, \bar{y}))\, d\mathcal{E}[\![e_2]\!]\rho[x \mapsto \bar{x}] \text{ where } \bar{x} = \mathcal{E}[\![e_1]\!]\rho
\end{aligned}
$$

Again, we obtain a product distribution. The last equality follows from the property of an indicator function under an integral and justifies the intuitive substitution semantics.

Third, we can check that the distribution comprehension $\{e \mid x \leftarrow gen\}_D$ results in an n-fold product distribution.

$$
\begin{aligned}
&\mathcal{E}[\![\mathcal{T}[\![\{e \mid x \leftarrow gen\}_D]\!]]\!]\rho(U) \\
&= \mathcal{E}[\![\texttt{sequenceA}\ \{\mathcal{T}[\![e]\!] \mid x \leftarrow \mathcal{T}[\![gen]\!]\}]\!]\rho(U) \\
&= \int \bar{x}_1 \mapsto \ldots \int \bar{x}_n \mapsto \mathbb{1}_U((\bar{x}_1, \ldots, \bar{x}_n))\, d\mathcal{E}[\![e]\!]\rho[x \mapsto \bar{g}_n] \ldots d\mathcal{E}[\![e]\!]\rho[x \mapsto \bar{g}_1]\ ,
\end{aligned}
$$

where $(\bar{g}_1, \ldots, \bar{g}_n) = \mathcal{E}[\![\mathcal{T}[\![gen]\!]]\!]\rho$. Note that `sequenceA` forces evaluation of the list comprehension, so a diverging generator expression results in the bottom valuation. Moreover, note that each $\mathcal{E}[\![e]\!]\rho[x \mapsto \bar{g}_i]$ is independent of each other for each $1 \leq i \leq n$, and thus, we refer to the n-fold product distribution as also a *parallel* comprehension.

Finally, we can check the semantics of observation.

$$\begin{aligned}\mathcal{E}[\![\mathcal{T}[\![y \leftarrow e_h \,;;\, \texttt{obs}\, e_o]\!]]\!]\rho &\triangleq \texttt{obsDens}\ \mathcal{T}[\![e_h]\!]\ (\lambda y.\, \texttt{getBndCondDens}(\mathcal{T}[\![e_o]\!])(y)) \\ &= \bar{y} \mapsto B \mapsto \frac{\int_B f_{\bar{y}}\, d\mu}{\int_{\mathcal{V}[\![\tau_o]\!]} f_{\bar{y}}\, d\mu}\,,\end{aligned}$$

where $\mu \triangleq \mathcal{E}[\![\Gamma \vdash e_h : \texttt{Dist}\ \tau_h]\!]\rho$ and $f_{\bar{y}} \triangleq \texttt{getBndCondDens}(\mathcal{T}[\![e_0]\!])$. That is, $f_{\bar{y}}$ satisfies

$$\int_B f_{\bar{y}}\, d\nu = \nu(B)$$

for any measurable B and $\bar{y} \in \mathcal{V}[\![\tau_o]\!]$, where $\nu \triangleq \mathcal{E}[\![\Gamma, y : \tau_h \vdash e_o : \texttt{Dist}\ \tau_o]\!]\rho[y \mapsto \bar{y}]$. As expected, the semantics is a function from observed data to the posterior distribution.

7.3 Density Factorization

The derived, distributional semantics enables us to reason about Core AugurV2 models. Nevertheless, for the purposes of implementing MCMC algorithms to sample from the model's posterior distribution, we will need the model's (potentially unnormalized) density. Hence, in this section, we characterize the distributions that Core AugurV2 models denote as those admitting a density factorization.

To begin, we need to indicate what base measure a Core AugurV2 model has density with

respect to. Towards this end, we define a collection of base measures by induction on types.

$$\mathfrak{B}[\![\texttt{Int}]\!] \triangleq \{\nu_{\text{counting}}\}$$

$$\mathfrak{B}[\![\texttt{Real}]\!] \triangleq \{\nu_{\text{Lebesgue}}\}$$

$$\mathfrak{B}[\![\texttt{Real}|_{\texttt{pt}}]\!] \triangleq \bigcup_{r \in \mathbb{R}} \mathbb{1}_{\{r\}}$$

$$\mathfrak{B}[\![\tau_1 \times \tau_2]\!] \triangleq \{\overline{\nu_1 \otimes \nu_2} \mid \nu_1 \in \mathfrak{B}[\![\tau_1]\!] \wedge \nu_2 \in \mathfrak{B}[\![\tau_2]\!]\}$$

The base measures for types `Int` and `Real` are the counting measure and Lebesgue measure respectively. The base measure for a real restricted to a point $\texttt{Real}|_{\texttt{pt}}$ is a family of point measures $(\mathbb{1}_{\{r\}})_{r \in \mathbb{R}}$ indexed by reals. The base measure for a product type $\tau_1 \times \tau_2$ is the completion of the product measure of the base measures for τ_1 and τ_2. We include the completion so that the base measure for $\texttt{Real} \times \texttt{Real}$ is the Lebesgue measure on $\mathbb{R} \times \mathbb{R}$.[1]

Now that we have the base measure for a Core AugurV2 program of type $\texttt{Dist}\ \tau$, we can show that the denotation has a density with respect to the base measure. Towards this end, we will need to strengthen the interpretation of distribution types of Core AugurV2 programs to those that admit a density with respect to the appropriate base measure.

$$\mathcal{V}[\![\texttt{Dist}\ \tau]\!] \triangleq \{\mu \mid \mu \in \mathscr{C}(\mathcal{V}[\![\tau]\!], [0,1]_<) \wedge \exists f, \nu \in \mathfrak{B}[\![\tau]\!].\mu = f d\nu\}\ .$$

Then, the density factorization result is a consequence of type-soundness for AugurV2 programs.

Lemma 7.3.1 (Type-soundness)**.** *If* $\Gamma \vdash e : \tau$*, then* $\mathcal{E}[\![e]\!] : \mathscr{C}(\mathcal{V}[\![\Gamma]\!], \mathcal{V}[\![\tau]\!])$

[1] This is a (minor) technicality that affects the version of Fubini-Tonelli we can use.

Proof. The proof follows by induction on the typing derivation, where in each case that works with distributions, we can explicitly witness what the density is. It is instructive to work through the distribution cases to see what the density is.

Case 6 ($\Gamma \vdash x \leftarrow e_1 \;;;\; e_2 : \texttt{Dist}\ \tau_1 \times \tau_2$). We need to witness a density for $\mathcal{E}[\![x \leftarrow e_1 \;;;\; e_2]\!]\rho$ for $\rho \in \mathcal{V}[\![\Gamma]\!]$. By the induction hypothesis, $\mathcal{E}[\![e_1]\!] : \mathscr{C}(\mathcal{V}[\![\Gamma]\!], \mathcal{V}[\![\texttt{Dist}\ \tau_1]\!])$ and $\mathcal{E}[\![e_2]\!] : \mathscr{C}(\mathcal{V}[\![\Gamma, x : \tau_1]\!], \mathcal{V}[\![\texttt{Dist}\ \tau_2]\!])$. Let f be the density and ν_1 be the base measure corresponding to $\mathcal{E}[\![e_1]\!]\rho$. Moreover, let $\kappa \triangleq \operatorname{curry}(\mathcal{E}[\![e_2]\!])(\rho) \in \mathscr{C}(\mathcal{V}[\![\tau_1]\!], \mathcal{V}[\![\texttt{Dist}\ \tau_2]\!])$. κ is a probability kernel, and moreover, there is $g_{\bar{x}}$ and $\nu_2 \in \mathfrak{B}[\![\tau_2]\!]$ such that $\kappa(\bar{x}) = g_{\bar{x}}$. We claim

$$\bar{x}, \bar{y} \mapsto f(\bar{x}) g_{\bar{x}}(\bar{y})$$

is a density with respect to $\nu_1 \otimes \nu_2 \in \mathfrak{B}[\![\tau_1 \times \tau_2]\!]$, which follows by calculation.

$$\begin{aligned}
&\int (\bar{x}, \bar{y} \mapsto \mathbb{1}_U(\bar{x}, \bar{y}) f(\bar{x}) g_{\bar{x}}(\bar{y}))\ d(\nu_1 \otimes \nu_2) \\
&= \int \left(\bar{x} \mapsto \int (\bar{y} \mapsto \mathbb{1}_U(\bar{x}, \bar{y}))\, g_{\bar{x}}(\bar{y})\, d\nu_2 \right) f(\bar{x}) d\nu_1 \\
&= \int \left(\int (\mathbb{1}_U(\bar{x}, \bar{y}))\ d\mathcal{E}[\![e_2]\!]\rho[x \mapsto \bar{x}] \right) d\mathcal{E}[\![e_1]\!]\rho
\end{aligned}$$

Case 7 ($\Gamma \vdash x = e_1 \;;;\; e_2 : \texttt{Dist}\ r(\tau_1) \times \tau_2$). We need to witness a density for $\mathcal{E}[\![x = e_1 \;;;\; e_2]\!]\rho$ for $\rho \in \mathcal{V}[\![\Gamma]\!]$. By the induction hypothesis, $\mathcal{E}[\![e_1]\!] : \mathscr{C}(\mathcal{V}[\![\Gamma]\!], \mathcal{V}[\![\tau_1]\!])$ and $\mathcal{E}[\![e_2]\!] : \mathscr{C}(\mathcal{V}[\![\Gamma, x : \tau_1]\!], \mathcal{V}[\![\texttt{Dist}\ \tau_2]\!])$. Let $\tilde{x} \triangleq \mathcal{E}[\![e_1]\!]\rho$ and $\mu \triangleq \operatorname{curry}(\mathcal{E}[\![e_2]\!])(\rho)(\tilde{x}) \in \mathcal{V}[\![\texttt{Dist}\ \tau_2]\!]$. Thus, there is density $g_{\tilde{x}}$ and base measure $\nu_2 \in \mathfrak{B}[\![\tau_2]\!]$ such that $g_{\tilde{x}}\, d\nu_2 = \mu$. We claim

$$\bar{x}, \bar{y} \mapsto g_{\tilde{x}}$$

is a density with respect to $\mathbb{1}_{\{\tilde{x}\}} \otimes \nu_2$, which follows by calculation.

$$\begin{aligned}
&\int (\bar{x}, \bar{y} \mapsto \mathbb{1}_U(\bar{x}, \bar{y}) g_{\tilde{x}}(\bar{y}))\ d(\mathbb{1}_{\{\tilde{x}\}} \otimes \nu_2) \\
&= \int \left(\bar{x} \mapsto \int (\bar{y} \mapsto \mathbb{1}_U(\bar{x}, \bar{y}))\, g_{\tilde{x}}(\bar{y})\, d\nu_2 \right) d\mathbb{1}_{\{\tilde{x}\}} \\
&= \int (\bar{y} \mapsto \mathbb{1}_U((\tilde{x}, \bar{y})))\ d\mathcal{E}[\![e_2]\!]\rho[x \mapsto \tilde{x}]
\end{aligned}$$

□

7.4 On Modeling Language Design

We just saw an application of the semantics from the first part of the dissertation by giving semantics to Core AugurV2 by translation and characterizing the models as those that admit a density factorization. In this section, we offer a few additional remarks on how a semantics based on computable distribution can be used to aid the design of a probabilistic modeling language.

Error values Note that a well-typed AugurV2 program can denote an *ill-formed* distribution, *e.g.*, a distribution for which it is nonsensical to condition on. For instance, we can define a model where the variance argument to a Normal distribution is negative. AugurV2 handles this *partiality* by using the diverging value $\bot$ to represent an error value and considering distributions on spaces that take $\bot$ into account. For example, supplying a negative variance to a Normal distribution produces $\bot \in \mathcal{V}[\![\texttt{Dist Real}]\!]$.

The constructs that can potentially introduce an error value $\bot$ in AugurV2 include application of primitive functions $\mathrm{op}^r(e_1, \ldots, e_n)$, instantiating a parameterized distribution

dist$(e_1, \ldots, e_n)$, and vector indexing $e[e]$. We could make all these operations total, in which case every well-typed AugurV2 program would denote a distribution that is meaningful to condition on. For example, we could give vector indexing $e_1[e_2]$ the semantics where we use the index e_2 modulo the length of e_1 (see [17]). Nevertheless, modifying the underlying semantics to get rid of error values is somewhat unsatisfying. Instead, by compiling to a suitably expressive language such as λ_{CD} with the appropriate semantics, we can get the desired semantics.

CONDITIONING The computable semantics from the first part of the dissertation also enables us to show that when an AugurV2 program does not denote an error value, then conditioning is computable.

Proposition 7.4.1. *When AugurV2 denotes a full-measure distribution, then conditioning is Type-2 computable.*

This easily follows as we have restricted our conditioning primitive to a Type-2 computable setting when we have a bounded conditional density. Indeed, a conditioning expression is simply translated to a call to the appropriate function implemented in the module `CondLib` (see Section 4.7.2 where we implement conditioning as a library).

IF EXPRESSIONS AugurV2 does not provide `if` expressions. In order to branch on a probabilistic expression, users must instead encode it with a Bernoulli random variable. For instance, one possible encoding on real-valued expressions e_2 and e_3 and condition x (drawn from a distribution e_1) is

$$\texttt{if0}\ x\ \texttt{then}\ e_2\ \texttt{else}\ e_3 = x * e_2 + (1 - x) * e_3\ .$$

This kind of encoding is done in practice to encode branching expressions in graphical models. The marginal distributions induced will be the same for both encodings. To see this, we can calculate out the semantics as we have done before. On the left, we have

$$U \mapsto \int \bar{x} \mapsto \begin{cases} \mathcal{E}[\![\Gamma \vdash e_2 : \texttt{Dist Real}]\!]\rho[x \mapsto \bar{x}](U) & \text{when } \bar{x} = 1 \\ \mathcal{E}[\![\Gamma \vdash e_3 : \texttt{Dist Real}]\!]\rho[x \mapsto \bar{x}](U) & \text{when } \bar{x} = 0 \end{cases} d\mathcal{E}[\![\Gamma \vdash e_1 : \texttt{Dist Nat}]\!]\rho \,.$$

On the right, we have

$$\int \bar{x} \mapsto \bar{x}\, \mathcal{E}[\![\Gamma \vdash e_2 : \texttt{Dist Real}]\!]\rho[x \mapsto \bar{x}](U) + (1-\bar{x})\, \mathcal{E}[\![\Gamma \vdash e_3 : \texttt{Dist Real}]\!]\rho[x \mapsto \bar{x}](U)\, d\mathcal{E}[\![\Gamma \vdash e_1 : \texttt{Dist Nat}]\!]\rho \,.$$

These are equivalent.

However, there are conceptually two different methods we can use to implement a MCMC sampler for these distributions. To see this, it is helpful to look at what is going on from the perspective of density factorization. For the distribution denoted by e_2, we will write $p_2(y_1, \ldots, y_k, y_{k+1}, \ldots y_n)$ for its density. Similarly for the distribution denoted by e_3, we will write $p_3(y_1, \ldots, y_k, y_{k+1}, \ldots y_m)$ for its density. Note that the prefix $y_1, \ldots, y_k$ is shared between p_2 and p_3 so that $y_{k+1}, \ldots y_n$ has been marginalized out for p_2 and $y_{k+1}, \ldots y_m$ has been marginalized out for p_3. Then, on the left, we have

$$p_1(\bar{x}) \begin{cases} p_2(y_1, \ldots, y_k, y_{k+1}, \ldots y_n) & \text{when } \bar{x} = 1 \\ p_3(y_1, \ldots, y_k, y_{k+1}, \ldots y_m) & \text{when } \bar{x} = 0 \end{cases} .$$

One the right, we have

$$p_1(\bar{x})\bar{x}\, p_2(y_1, \ldots, y_k, y_{k+1}, \ldots y_n) + (1 - \bar{x})\, p_1(\bar{x})\, p_3(y_1, \ldots, y_k, y_{k+1}, \ldots y_m)\,.$$

The density on the left is reminiscent of model selection. Depending on the condition, we either have model with density p_2 or a model with density p_3. These models can be handled with reversible-jump MCMC [41] which is designed for the case when each model has a different number of dimensions. In our example, we have explicitly written out the marginalized variables for p_2 and p_3 to indicate that each model may make a different number of choices. A reversible-jump MCMC has a more involved acceptance ratio calculation as well as the need to compute Jacobians (to ensure reversibility). In contrast, the density on the right considers both branches simultaneously, and hence, can be treated with ordinary MH methods. Note that this works because the model has been unfolded so that *both* branches are considered. Otherwise, we would need a reversible-jump correction.

Our digression on `if` expressions illustrates two additional points about the design and implementation of probabilistic modeling languages. First, it is important that we have a semantics for probabilistic programs *independent* of another (equivalent) semantics useful for reasoning about inference algorithms. Indeed, our digression on `if` expressions, which AugurV2 intentionally leaves out, illustrates this point. The distributional semantics is agnostic to marginalization whereas the density factorization highlights the different algorithmic choices at our disposal.[2] In fact, the subtle difference between `if` and its encoding has been the root of buggy MCMC implementations in probabilistic programming languages (*e.g.* see

[2]Gated graphical models [cite] have been developed to add branching capabilities to graphical models, which subsequently impacts the inference algorithm.

Probabilistic Matlab [103] and Hakaru [2]). Second, we can expose algorithmic choices on the inference in the language itself. For instance, if we added `if` expressions to AugurV2, we can use that as a user hint to the compiler to generate reversible-jump MCMC algorithms instead of a more generic MCMC algorithm. This highlights the added dimension of considering *which* language features a probabilistic modeling language should provide, beyond just the standard programming language features.

8

Conclusion

In this dissertation, I investigated programming languages designed for probabilistic modeling. I began by exploring how to use *computable distributions* to give semantics to high-level programming languages designed for probabilistic modeling. The intuitive reason for why *computable distributions* provide a good foundation for the semantics of probabilistic programs is summarized by the following motto (repeated here): "Turing-complete probabilistic programming languages express computable distributions." The idea that computable distributions go hand in hand with probabilistic programs has been hinted at in the literature, although the

connection to the best of my knowledge has not been made precise at the level of a high-level probabilistic programming language until my work.

The second part of the dissertation focused on how to architect a PPL compiler. Towards this end, I considered the setting of fixed-structure, parametric models instead of general-purpose PPLs. Even in this restricted but practically useful setting, the design and implementation of compilers for PPLs is still a largely unexplored space. As a step towards advancing the construction of compilers for PPLs, I devised a sequence of ILs that enable a compiler to gradually transform a declarative specification of a model into an exectuable inference algorithm as well as layout the phases of compilation.

8.1 Future Work

In this section, I suggest a few directions for future work on both the semantics and implementation of PPLs.

Semantics The most pressing issue is to see if we can indeed embed the semantics based on topological domains into a realizability topos that I informally discussed in Section 4.8. This would enable us to rigorously make the connection to realizability. There is a third semantics that can possibly be given to λ_{CD} based on L-domains [37]. This approach encodes the notion of Type-2 computability using the order-theoretic notion of approximation instead of Type-2 Turing machines. Indeed, this may be the more natural way to extend a CPO model of PCF with reals and distributions instead of the way I have done it, which relies externally on oracle Turing machines.

Another direction to take the semantics of probabilistic programs based on computable distributions is to address the issues of *full abstraction* and *universality*. The hope would

be to obtain similar results as in Plotkin's landmark paper [81]: "LCF Considered as a Programming Language." As a model for how to extend the standard techniques to a PCF-like language extended with Type-2 computable elements, a good starting point is Longley's thesis [59]. As a first attempt, I think it is better to approach the problem of full abstraction and universality by adding language constructs, *i.e.*, parallel if (`pif`) and exists ($\exists$) respectively, instead of attempting to change the semantics. For example, as we saw in the Haskell implementation of λ_{CD}, we needed to use Haskell's imprecise exceptions mechanism to simulate $\exists$ so that we could implement a (Type-2) computable function that would not have been implementable otherwise. This is akin to adding $\exists$ to the language. Intuitively, I expect there to be a corresponding *full abstraction* and *universality* result to hold for a probabilistic language with a uniform distribution plus the standard constructs. Longely obtains full abstraction and universality results for PCF extended with parallel if and exists using realizability toposes.

One of the original motivations I had for basing the foundations of probabilistic programming on computable distributions was to create a probabilistic Chomsky hierarchy. The idea was to create a hierarchy of distributions which had different computational requirements with the hope that this would translate into corresponding sub-languages with guaranteed efficient inference. At first glance, this approach appears to be doomed because it does not take much expressive power to encode 3SAT (*e.g.*, $\wedge$, $\vee$, and coin flips) and ask an automated inference engine to solve it. Here, we can take a cue from the non-computability of conditioning—conditioning in the presence of bounded, computable noise is computable. From this perspective, it may be meaningful to attack the problem of designing sub-languages with guaranteed efficient inference by considering the kind of noise added.

Implementation It would be interesting to add language features to the AugurV2 modeling language and see how robust the current architecture of the AugurV2 compiler is to these additions. As a first step, adding non-parametric prior distributions to AugurV2 as primitives would likely require adding (1) additional compiler analysis to reason about properties of those distributions and (2) runtime library support to implement the functionality. This increases the complexity of the compiler, but does not change its architecture. As a second step, adding a Markovian looping construct (*i.e.*, one where the current loop iteration depends only on the previous loop iteration) would enable us to express time-series and state-space models in AugurV2 . We would need to extend (1) the Density IL to contain a sequential structure product, (2) the decomposition analysis to support this additional case, and (3) the implementation of AD. Importantly, the backend remains largely unchanged—the only parts of the compiler that are affected are those that work on the Density IL. Nevertheless, this additional sequential looping construct suggests that we should also extend the compiler with additional inference techniques such as Particle MCMC [6] that are designed specifically to handle such conditional independence structures. Thus, we would also likely want to extend the Kernel IL. This suggests that for each new language feature we add that requires extending the Density IL, we should also extend the Kernel IL.

Third, it would be interesting to see what other ILs are useful for compiling inference and what other inference techniques can be automated with them. In AugurV2, I focused on MCMC inference algorithms, but there are others such as variational inference and possibly combinations of the two. Fourth, what other architectures could be used to accelerate inference? For example, tensor-flow would be an interesting target of compilation. Fifth and a related point, it would be interesting to examine heterogenous computation. Currently, AugurV2 generates

either all CPU code or GPU code, but there may be situations where the memory transfer is worth it.

8.2 Final Thoughts

I conclude with some thoughts on probabilistic programming. On the aspect of semantics, I think that computable distributions are a *good* foundation for interpreting probabilistic programs. In fact, I intended the motto "Turing-complete probabilistic modeling languages express computable distributions" to be a rephrasing of the Church-Turing thesis itself: "Turing-complete programming languages express computable functions." The choice to use computable distributions goes against the prevailing wisdom for giving semantics to probabilistic programs using measure theory. Hence, I provide two reasons below why I choose computable distributions instead.

First, a purely measure-theoretic semantics loses its connection to the operational behavior of programs, unless one bakes in primitives that deal with reals and distributions in a black-box manner. In particular, there is a mismatch between a semantics that deals only with ideal quantities and an implementation in a general-purpose language that deals only with approximations (*e.g.*, with floating point). Instead, it seems much more fruitful to deal with computability directly, *i.e.*, embrace that we can only compute approximately with continuous values. Moreover, such a (Type-2) computability theory exists.

Second, what does it mean for a PPL to be Turing-complete if we interpret probabilistic programs in purely measure-theoretic terms? The obvious answer is that such a PPL can simulate a Turing machine. However, it seems that such a language would have more computational power than an ordinary Turing machine if it can compute directly on reals and

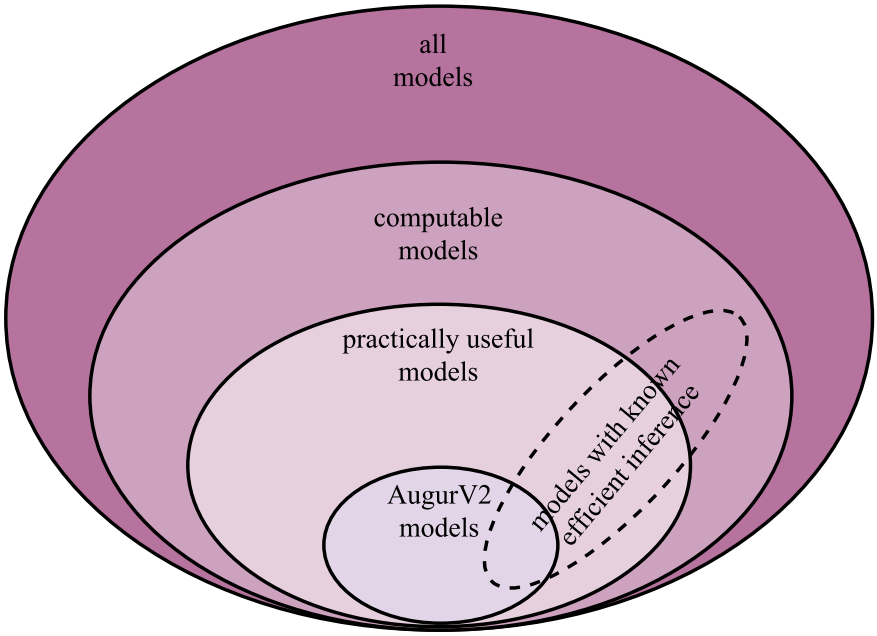


Figure 8.1: A cartoon of the design space of PPLs.

measures without approximation. In this case, we would violate the Church-Turing thesis. Alternatively, we can delimit the bounds of computation to what a Turing machine can compute, which places us in the realm of Type-2 computability. Indeed, the formulation of a computable distribution as a computable function that generates samples from bit-streams makes the connection back to probabilistic Turing machines, which is well known to be equivalent to Turing machines equipped with an oracle.

Figure 8.1 provides a cartoon view of where I think the design and implementation of practical probabilistic programming languages should head. I have included both non-computable models (*i.e.*, all models) and also importantly, practically-useful models. AugurV2 captures a subset of the practically-useful models. My hope was to design a programming language that would capture only the models with (known) efficient inference because inference (and not language expressivity) will be the bottleneck of probabilistic programming language's use-

fulness in practice. Hence, rather than focus on developing efficient inference algorithms for general-purpose languages, I believe we should focus instead on restricted language designs and improving inference in those settings.

To draw an analogy, consider a similar technology such as SMT/SAT solving. Like inference, SAT solving is intractable in general. Nevertheless, in practice, one can solve SAT problems and extend this capability to particular special cases of interest (*e.g.*, bit-vectors) via clever design of SMT solvers. In the context of probabilistic programming, the crafting of particular theories corresponds to designing restricted languages that handle particular classes of models efficiently.

For probabilistic programming to be as practical as SMT solving is today, I think we will likely need significant advances in inference techniques beyond MCMC that will push this technology further. Blackbox variational inference [55] seems like a promising alternative, but these techniques are still in its infancy. Hence, I do not think the aspects of AugurV2 specific to MCMC will last. Nevertheless, I believe its grounding on computable distributions and the use of ILs to aid compilation will influence the design and implementation of future languages.

A

Environment Lookup for CPO Semantics

This appendix contains the auxiliary argument that environment lookup is continuous, which is used in the CPO semantics (see [ref]). The intuition for why this result holds is because a variable x can be realized on a Turing machine as: "Turing machine code that looks at the tape indexed by x". Consequently, environment lookup is computable, and hence, continuous. The actual argument is quite tedious and is included mostly for those interested to see how the CPO semantics leverages computability. Indeed, the argument would not work if λ_{CD} was not designed with Type-2 computability in mind because then not all values would be

realizable on a Turing machine. Hence, the notion of Type-2 computability is crucial to the CPO semantics, even if it is not readily apparent at first.

The argument proceeds in several stages. First, we introduce a language that parameterizes a PCF-like language with constants denoting points in computable metric spaces and continuous operations on them called λ_R. λ_{CD} is an instantiation of λ_R. Second, we give λ_R both operational and denotational semantics. Third, we show *soundness* and *adequacy* to relate the two semantics. Fourth and crucially, we realize the operational semantics on an oracle Turing machine. Fifth and finally, we show the desired result that environment lookup is continuous.

A.1 A Language with Represented Spaces

The language λ_R extends a PCF-like language with constants denoting points in computable metric spaces and continuous operations on them. The syntax is given below.

$$\tau ::= \texttt{Nat} \mid \tau \to \tau \mid \tau \times \tau \mid \alpha \mid \beta$$

$$e ::= x \mid n \mid \oplus e \mid \lambda x.\, e \mid e\ e \mid \texttt{if0}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \mu x.\, e$$

$$\mid (e, e) \mid \texttt{fst}\ e \mid \texttt{snd}\ e$$

$$\mid c^{\alpha} \mid c^{\alpha} \mid \mathrm{op}^{r}(\overrightarrow{e})$$

The types α and β are base types that will be interpreted as computable metric spaces. For simplicity, we restrict our attention to two base represented spaces (*e.g.*, reals and distributions), but in general, we can have any number. The other types are standard. The first line of the expression syntax gives standard PCF-like expressions. The second line of the expression syntax adds pairs.

$\boxed{\vdash_R \tau}$ Representable type

$$\frac{}{\vdash_R \alpha} \qquad \frac{}{\vdash_R \beta} \qquad \frac{\vdash_R \tau_1 \qquad \vdash_R \tau_2}{\vdash_R \tau_1 \times \tau_2} \qquad \frac{\vdash_R \tau_1 \qquad \vdash_R \tau_2}{\vdash_R \tau_1 \to \tau_2}$$

$\boxed{\Gamma \vdash e : \tau}$ Expression typing

$$\frac{\Gamma(x)\tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \texttt{Nat}} \qquad \frac{\Gamma \vdash e : \texttt{Nat}}{\Gamma \vdash \oplus e : \texttt{Nat}} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau} \qquad \frac{\Gamma \vdash e_1 : \texttt{Nat} \qquad \Gamma \vdash e_2, e_3 : \tau}{\Gamma \vdash n : \texttt{if0}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mu x.\ e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}\ e : \tau_2} \qquad \frac{}{\Gamma \vdash c^\alpha : \alpha}$$

$$\frac{}{\Gamma \vdash c^\alpha : \beta} \qquad \frac{\Psi(\mathrm{op}^r) = \tau_1 \times \cdots \times \tau_n \to \tau \qquad \vdash_R \Psi(\mathrm{op}^r) \qquad \Gamma \vdash e_i : \tau_i \text{ for } 1 \le i \le n}{\Gamma \vdash \mathrm{op}^r(e_1, \ldots, e_n) : \tau}$$

Figure A.1: Type-system

The third line of the expression syntax adds constants c^α and c^α, which denote elements of the respective computable metric spaces. Let (X, d_X, S_X) and (Y, d_Y, S_Y) be the computable metric spaces that constants c^α and c^α denote elements of respectively. Concretely, constants c^α and c^α are given by fast Cauchy names with representations δ_X and δ_Y respectively. Hence, we can treat c^α and c^α as bit-streams. The syntax $\mathrm{op}^r(\overrightarrow{e})$ is a primitive application of a continuous map between computable metric spaces denoted by op^r to a list of arguments $\overrightarrow{e}$.

Figure A.1 summarizes the type system for λ_R. The type system is parameterized by a global environment Ψ that contains the types of primitive functions. The typing judgement $\Gamma \vdash e : \tau$ is mostly standard, so we will focus on the rules for constructs that operate on com-

putable metric spaces. As expected, constant elements of a computable metric space are assigned the appropriate base type. The rule for a primitive function application $\mathrm{op}^r(e_1, \ldots, e_n)$ checks that the function looked up from a global environment Ψ has a representable type (*i.e.*, derivable by $\vdash_R$) and that each of the arguments e_i has the appropriate type.[1]

A.2 Denotational Semantics

The denotational semantics of λ_R is mostly straightforward—we simply have to interpret additional constants and primitive functions.

A.2.1 Interpretation of Types

The interpretation of types $\mathcal{V}[\![\cdot]\!]$ is given by induction on types, where types are interpreted as CPOs. We use a call-by-name interpretation.

$$\mathcal{V}[\![\mathtt{Nat}]\!] \triangleq \mathrm{Disc}(\mathbb{N})_\perp$$

$$\mathcal{V}[\![\tau_1 \times \tau_2]\!] \triangleq (\mathcal{V}[\![\tau_1]\!] \times \mathcal{V}[\![\tau_2]\!])_\perp$$

$$\mathcal{V}[\![\tau_1 \to \tau_2]\!] \triangleq (\mathcal{V}[\![\tau_1]\!] \Rightarrow \mathcal{V}[\![\tau_2]\!])_\perp$$

$$\mathcal{V}[\![\alpha]\!] \triangleq S(L(X, d_X, S_X))$$

$$\mathcal{V}[\![\beta]\!] \triangleq S(L(X, d_Y, S_Y))$$

The interpretation of PCF-like types is standard. Recall that S applies the specialization order and L lifts a topological space to contain $\perp$. As computable metric spaces are Hausdorff,

[1] Although we require the base types to be computable metric spaces, arguments to primitive functions can come from more general represented spaces.

$$
\begin{aligned}
\mathcal{E}[\![x]\!]\rho &\triangleq \rho(x) \\
\mathcal{E}[\![n]\!]\rho &\triangleq \Upsilon(n) \\
\mathcal{E}[\![\oplus e]\!]\rho &\triangleq \Upsilon(\oplus)(\mathcal{E}[\![e]\!]\rho) \\
\mathcal{E}[\![\lambda x.\ e]\!]\rho &\triangleq \lfloor v \mapsto \mathcal{E}[\![e]\!]\rho[x \mapsto v] \rfloor \\
\mathcal{E}[\![e_1\ e_2]\!]\rho &\triangleq \text{unlift}(\mathcal{E}[\![e_1]\!]\rho)(\mathcal{E}[\![e_2]\!]\rho) \\
\mathcal{E}[\![\texttt{if0}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\rho &\triangleq \begin{cases} \mathcal{E}[\![e_2]\!]\rho & \text{if } \mathcal{E}[\![e_1]\!]\rho = \bar{0} \\ \mathcal{E}[\![e_3]\!]\rho & \text{if } \mathcal{E}[\![e_1]\!]\rho = \bar{n} \text{ for } \bar{n} \neq \bar{0} \\ \bot & \text{otherwise} \end{cases} \\
\mathcal{E}[\![\mu x.\ e]\!]\rho &\triangleq \text{fix}(v \mapsto \mathcal{E}[\![e]\!]\rho[x \mapsto v]) \\
\mathcal{E}[\![(e_1, e_2)]\!]\rho &\triangleq \lfloor (\mathcal{E}[\![e_1]\!]\rho, \mathcal{E}[\![e_2]\!]\rho) \rfloor \\
\mathcal{E}[\![\texttt{fst}\ e]\!]\rho &\triangleq \pi_1(\text{unlift}(\mathcal{E}[\![e]\!]\rho)) \\
\mathcal{E}[\![\texttt{snd}\ e]\!]\rho &\triangleq \pi_2(\text{unlift}(\mathcal{E}[\![e]\!]\rho)) \\
\mathcal{E}[\![c^\alpha]\!]\rho &\triangleq \Upsilon(c^\alpha) \\
\mathcal{E}[\![c^\alpha]\!]\rho &\triangleq \Upsilon(c^\alpha) \\
\mathcal{E}[\![\text{op}^r(e_1, \ldots, e_n)]\!]\rho &\triangleq \Upsilon(\text{op}^r)(\mathcal{E}[\![e_1]\!]\rho, \ldots, \mathcal{E}[\![e_n]\!]\rho)
\end{aligned}
$$

Figure A.2: The denotational semantics of λ_R

both $\mathcal{V}[\![\alpha]\!]$ and $\mathcal{V}[\![\alpha]\!]$ give lifted, flat CPOs.

A.2.2 Expression Denotation

Figure A.2 summarizes the denotational semantics of λ_R. The expression denotation function $\mathcal{E}[\![\Gamma \vdash e : \tau]\!] : \mathcal{V}[\![\Gamma]\!] \Rightarrow \mathcal{V}[\![\tau]\!]$ is defined by induction on expression typing jugements, but we will omit the judgement for brevity. It is parameterized by a well-formed global environment Υ that interprets constants and primitive functions.

A global environment is well-formed when:

1. For any constant c^α, $\Upsilon(c^\alpha) = \delta_X(c^\alpha)$ when $c^\alpha \in \text{dom}(\delta_X)$ and $glob(c^\alpha) = \bot$ otherwise.
2. For any constant c^α, $\Upsilon(c^\alpha) = \delta_Y(c^\alpha)$ when $c^\alpha \in \text{dom}(\delta_Y)$ and $glob(c^\alpha) = \bot$ otherwise.

3. For any primitive function op^r, $\Upsilon(\text{op}^r)$ is a strongly-continuous realizer of op^r (see [cite]). By strongly continuous, we mean that op^r diverges if any of its inputs diverges (*e.g.*, when input bit-streams are not names of elements in the appropriate represented spaces). This is also known as a strict continuous function.

The meanings of PCF-like expressions are standard. The denotation of a constant c^α (c^α) is a global environment lookup $\Upsilon(c^\alpha)$ ($\Upsilon(c^\alpha)$). Concretely, the lookup converts the (fast Cauchy) name into its corresponding element. When c^α (c^α) is not a (fast Cauchy) name, it denotes $\perp$.[2] We interpret a primitive function application $\text{op}^r(e_1, \ldots, e_n)$ as a strict function application, which applies the semantic function $\Upsilon(\text{op}^r)$.

Before continuing, we should check that the expression denotation function is well-defined.

Lemma A.2.1 (Denotation well-defined)**.** *If* $\Gamma \vdash e : \tau$*, then* $\mathcal{E}[\![e]\!] : \mathcal{V}[\![\Gamma]\!] \Rightarrow \mathcal{V}[\![\tau]\!]$.

Proof. We proceed by induction on the typing derivation and note that the standard proof works for the PCF-like fragment of λ_R (*e.g.*, see Gunter [42]).

Case 8 ($\Gamma \vdash c^\alpha : \alpha$). We have $\mathcal{E}[\![c^\alpha]\!] = \text{const}(\Upsilon(c^\alpha))$, where const is continuous. The case $\Gamma \vdash c^\alpha : \alpha$ is similar.

Case 9 ($\Gamma \vdash \text{op}^r(e_1, \ldots, e_n) : \tau$). We need to show $\text{apply} \circ \langle \Upsilon(\text{op}^r), \mathcal{E}[\![e_1]\!], \ldots, \mathcal{E}[\![e_n]\!] \rangle : \mathcal{V}[\![\Gamma]\!] \Rightarrow \mathcal{V}[\![\tau]\!]$. By assumption, $\Upsilon(\text{op}^r)$ is a continuous map between represented spaces, so it is also a CPO continuous function (because it's topology is coarser than the Scott topology). By our induction hypothesis, we have that $\mathcal{E}[\![\Gamma \vdash e_i : \tau_i]\!] : \mathcal{V}[\![\Gamma]\!] \Rightarrow \mathcal{V}[\![\tau_i]\!]$. The results then follows because the denotation uses only continuous functions.

□

[2] Unlike standard PCF, the denotation of a value of "base types" α or β may result in $\perp$. Hence, referring to α and β as base types is somewhat of a misnomer.

$\boxed{e \Downarrow v}$ Large-step call-by-name evaluation

$$\frac{}{n \Downarrow n} \qquad \frac{e \Downarrow n}{\oplus e \Downarrow \Upsilon(\oplus)(n)} \qquad \frac{}{\lambda x.\, e \Downarrow \lambda x.\, e} \qquad \frac{e_1 \Downarrow \lambda x.\, e \qquad e[e_2/x] \Downarrow v}{e_1\ e_2 \Downarrow v}$$

$$\frac{e_1 \Downarrow 0 \qquad e_2 \Downarrow v}{\texttt{if0}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v} \qquad \frac{e_1 \Downarrow n \neq 0 \qquad e_3 \Downarrow v}{\texttt{if0}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v} \qquad \frac{e[\mu x.\, e/x] \Downarrow v}{\mu x.\, e \Downarrow v}$$

$$\frac{}{(e_1, e_2) \Downarrow (e_1, e_2)} \qquad \frac{e \Downarrow (e_1, e_2) \qquad e_1 \Downarrow v_1}{\Gamma \vdash \texttt{fst}\ e : v_1} \qquad \frac{e \Downarrow (e_1, e_2) \qquad e_2 \Downarrow v_2}{\Gamma \vdash \texttt{snd}\ e : v_2} \qquad \frac{}{c^\alpha \Downarrow c^\alpha}$$

$$\frac{}{c^\alpha \Downarrow c^\alpha} \qquad \frac{e_i \Downarrow v_i \text{ for } 1 \leq i \leq n}{\mathrm{op}^r(e_1, \ldots, e_n) \Downarrow (\Upsilon(\mathrm{op}^r))(v_1, \ldots, v_n)}$$

Figure A.3: Operational semantics

A.3 Operational Semantics

In this section, we give λ_R operational semantics. The values that λ_R expression evaluate to are given below.

$$v ::= n \mid \lambda x.\, e \mid (e, e) \mid c^\alpha \mid c^\alpha$$

An expression can evaluate to an element of a computable metric space in addition to the standard values. The value (e, e) indicates that evaluation of pairs is lazy.

Figure A.3 summarizes the large-step, call-by-name operational semantics of λ_R. It is parameterized by a global environment Υ that interprets primitive function symbols op^r. The operational semantics of the PCF-like expressions are standard and will not be discussed further. We describe the semantics of the additional constructs λ_R.

A constant (*i.e.*, bit-stream encoding a fast Cauchy sequence) is already a value, and thus, reduces to itself. The reduction of $\mathrm{op}^r(e_1, \ldots, e_n)$ reduces each e_i to a corresponding value v_i for $1 \leq i \leq n$, and then applies the semantic version of the primitive function obtained by looking up the symbol op^r in the global environment to the resulting arguments. Note that the operational semantics does not substitute the values v_i into the body of $\Upsilon(\mathrm{op}^r)$. Instead, the meaning of $\Upsilon(\mathrm{op}^r)$ is given externally by an oracle Turing machine realizing the function. By design of λ_{CD} and the well-formedness of Υ, such an oracle Turing machine exists.

A.4 Soundness and Adequacy

Now, we connect the operational semantics with the denotational semantics via soundness and adequacy. All statements are made under the additional hypothesis that the global environment Υ is well-formed. We first state a substitution lemma for λ_R.

Lemma A.4.1 (Substitution)**.** *If* $\Gamma, x : \tau_x \vdash e : \tau$ *and* $\cdot \vdash e_x : \tau_x$*, then* $\mathcal{E}[\![\Gamma, x : \tau_x \vdash e : \tau]\!]\rho = \mathcal{E}[\![\Gamma \vdash e[e_x/x] : \tau]\!]\rho$ *for any well-formed environment* ρ *(with respect to* Γ*).*

Proof. By induction on the structure of e. As λ_R does not introduce any additional binding constructs, the standard proof works. □

We will use the notation σe to indicate the substitution of multiple expressions where $\sigma = [e_1/x_1, \ldots, e_n/x_n]$.

Soundness is straightforward to show.

Lemma A.4.2 (Soundness)**.** *Let* $\Gamma \vdash e : \tau$*. If* $\sigma e \Downarrow v$*, then* $\mathcal{E}[\![e]\!]\rho = \mathcal{E}[\![v]\!]\rho$ *for any substitution* σ *and environment* ρ *that are well-formed with respect to* Γ*.*

Proof. By induction on the height of the derivation of $e \Downarrow v$. The standard proof works for the original constructs. The constant cases hold trivially.

Case 10 $(\mathrm{op}^r(e_1, \ldots, e_n) \Downarrow (\Upsilon(\mathrm{op}^r))(v_1, \ldots, v_n))$. We have to show $\mathcal{E}[\![\mathrm{op}^r(e_1, \ldots, e_n)]\!]\rho = \mathcal{E}[\![\mathrm{op}^r(v_1, \ldots, v_n)]\!]\rho$, which holds iff $\Upsilon(\mathrm{op}^r)(\mathcal{E}[\![e_1]\!]\rho, \ldots, \mathcal{E}[\![e_n]\!]\rho) = \Upsilon(\mathrm{op}^r)(v_1, \ldots, v_n)$. By the induction hypothesis, $\mathcal{E}[\![e_i]\!]\rho = \mathcal{E}[\![v_i]\!]$ for $1 \leq i \leq n$. Hence, the result holds.

□

Following a standard technique (*e.g.*, see Gunter [42]), we use a logical relation $d \lesssim_\tau e$ that relates CPO elements d to closed terms of e to show adequacy. It is defined mutually recursively with another relation on values $d \lesssim_\tau v$.

Definition A.4.1. Let $d \lesssim_\tau e$ if:

- $\mathcal{E}[\![\cdot \vdash e : \tau]\!] = \bot$ or
- there exists some v such that $e \Downarrow v$ and $\mathcal{E}[\![\cdot \vdash e : \tau]\!] \lesssim_\tau v$.

Let $d \lesssim_\tau v$ if:

- $\bar{n} \lesssim_{\mathtt{Nat}} n$ where $\bar{n}$ is the semantic value associated with the syntax n,
- $f \lesssim_{\tau_1 \to \tau_2} x \mapsto e$ if for any $d \lesssim_{\tau_1} v$, $f(d) \lesssim_{\tau_2} e[v/x]$,
- $(d_1, d_2) \lesssim_{\tau_1 \times \tau_2} (e_1, e_2)$ if $d_1 \lesssim_{\tau_1} e_1$ and $d_2 \lesssim_{\tau_2} e_2$,
- $\delta_X(c^\alpha) \lesssim_\alpha c^\alpha$, or
- $\delta_Y(c^\alpha) \lesssim_\beta c^\alpha$.

The relation $\lesssim_\tau$ on values for PCF types is standard. For the computable metric space type α, we have $\delta_X(c^\alpha) \lesssim_\alpha c^\alpha$, so a constant c^α is related to the name of the element $\delta_X(c^\alpha)$. Note that an element can have multiple names. The relation $\delta_Y(c^\alpha) \lesssim_\alpha c^\alpha$ is defined similarly.

The following (standard) strengthened lemma implies adequacy.

Lemma A.4.3. *Let* $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$. *If* $\Gamma \vdash e : \tau$ *and for* $1 \leq i \leq n$, $v_i \in \mathcal{V}[\![\tau_i]\!]$ *and* $v_i \lesssim_{\tau_i} e_i$ *for well-typed closed* e_i, *then*

$$\mathcal{E}[\![e]\!] \lesssim_\tau e[v_1/x_1, \ldots, v_n/x_n] \,.$$

where $\rho = [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ *is a well-formed environment and* $\sigma = [v_1/x_1, \ldots, v_n/x_n]$ *is a well formed substitution.*

Proof. We proceed by induction on the structure of e. As before, the cases for the standard constructs go through without modification.

Case 11 (c^α). We need to show $\mathcal{E}[\![c^\alpha]\!]\rho \lesssim_\alpha \sigma c^\alpha$, which holds iff $\Upsilon(c^\alpha) \lesssim_\alpha c^\alpha$. There are two cases. If $c^\alpha in \,\mathrm{dom}(\delta_X)$, then we have $\Upsilon(c^\alpha) = \delta_X(c^\alpha) \lesssim_\alpha c^\alpha$ by well-formedness of Υ. If $c^\alpha \notin \mathrm{dom}(\delta_X)$, then $\Upsilon(c^\alpha) \lesssim_\alpha c^\alpha$ because $\Upsilon(c^\alpha) = \bot$. The case for c^α is similar.

Case 12 ($\mathrm{op}^r(e_1, \ldots, e_n)$). We need to show $\Upsilon(\mathrm{op}^r)(\mathcal{E}[\![e_1]\!]\rho, \ldots, \mathcal{E}[\![e_n]\!]\rho) \lesssim_\tau \Upsilon(\mathrm{op}^r)(v_1, \ldots, v_n)$, where $e_i \Downarrow v_i$ for $1 \leq i \leq n$. By our inductive hypothesis, we have that $\mathcal{E}[\![e_1]\!]\rho \lesssim_{\tau_i} v_i$ for $1 \leq i \leq n$. At representable types, $\mathcal{E}[\![e_1]\!]\rho \lesssim_{\tau_i} v_i$ when v_i and $\mathcal{E}[\![e_i]\!]\rho$ name the same element of a computable metric space. As our global environment glob is well-formed, $\Upsilon(\mathrm{op}^r)$ is a strongly-continuous realizer, so the output must name the same element as well (or diverge if any of its inputs diverge).

□

As usual, we obtain adequacy as a corollary.

Corollary A.4.4. *(Adequacy) Let* $\cdot \vdash e : \tau$. *If* $expD\cdot \vdash e : \tau \neq \bot$, *then there is a* v *such that* $e \Downarrow v$.

A.5 Oracle Turing Machines

Now, we show that the operational semantics can be realized by an oracle Turing machine. First, we will encode λ_R expressions on a Turing machine tape. Second, we describe an oracle Turing machine which takes an encoding of an expression on its input tape and writes the resulting value to its output tape. As a warning, the exercise is tedious. Nevertheless, the crucial point is that this is possible only because λ_R (and consequently λ_{CD}) is designed with Type-2 computability in mind—we will not be able to realize measures on measurable spaces on an oracle Turing machine.

A.5.1 Preliminaries

Before we begin, we establish some notation and recall some standard facts about Turing machines.

The notation $\langle\, x_1, \ldots, x_n \,\rangle$ is a standard tupling function from computability theory which can be used to pair streams of characters from an alphabet together (*e.g.*, by dovetailing). The notation $\mathrm{bin} : \mathbb{N} \to \{0,1\}^*$ converts a natural into its binary encoding.

Recall that we can treat a single working tape as an unbounded number of working tapes, again by dovetailing. Thus, a Turing machine can always create a new working tape if needed.

When we say that a Turing machine does an operation "on-demand", we mean that a Turing machine does the appropriate dovetailing to ensure that tape contents are computed lazily. Moreover, recall that we can run a Turing machine for a certain number of steps, inspect the results, and continue.

A.5.2 Encoding

The encoding function encode(·) translates an expression into a stream of characters on a Turing machine tape encoding the expression. The Turing machine alphabet Δ consists of enough characters to encode the syntax of λ_R. We reify the primitive functions $\oplus_n$ and op_n^r by indexing them with a natural n (in its binary encoding). That is, $\oplus ::= \oplus_1 \mid \oplus_2 \mid \ldots$ and $\mathrm{op}^r ::= \mathrm{op}_1^r \mid \mathrm{op}_2^r \mid \ldots$.

$$
\begin{aligned}
\Delta ::= {} & 0 \mid 1 \\
& \mid \texttt{Nat} \mid \texttt{Var} \mid \oplus_n \mid \texttt{If0} \mid \texttt{Lam} \mid \texttt{App} \mid \texttt{Mu} \\
& \mid \texttt{Pair} \mid \texttt{Fst} \mid \texttt{Snd} \\
& \mid \texttt{Rep}_\alpha \mid \texttt{Rep}_\beta \mid \mathrm{op}_n^r
\end{aligned}
$$

Figure A.4 gives the encoding function, which is in essence an *identity* function. Note that the tupling function $\langle \cdot \rangle$ is needed because the translation of c^α and c^α give infinite bitstreams. Also, we assume that the expressions have been converted to DeBruijn indices before the encoding.

A.5.3 Interpreter

Once an expression has been encoded on the tape, note that we can perform DeBruijn index shifting (lshift to decrement the indices and rshift to increment the indices) of a syntactically well-formed expression eagerly because expressions have finite height. That is, we can recursively descend into each subexpression and eagerly perform the appropriate shifting.

The function extend$\langle t_1, t_2, x \rangle$ is realized by a machine where mentions of the variable x

$$
\begin{aligned}
\mathrm{encode}(x) &\triangleq \texttt{Var}\langle\ \mathrm{bin}(x)\ \rangle \\
\mathrm{encode}(n) &\triangleq \texttt{Nat}\langle\ \mathrm{bin}(n)\ \rangle \\
\mathrm{encode}(\oplus_i e) &\triangleq \oplus_i\langle\ \mathrm{encode}(e)\ \rangle \\
\mathrm{encode}(if e e_1 e_2 e_3) &\triangleq \texttt{If}\langle\ \mathrm{encode}(e_1), \mathrm{encode}(e_2), \mathrm{encode}(e_3)\ \rangle \\
\mathrm{encode}(\lambda e) &\triangleq \texttt{Lam}\langle\ \mathrm{encode}(e)\ \rangle \\
\mathrm{encode}(e_1\ e_2) &\triangleq \texttt{App}\langle\ \mathrm{encode}(e_1), \mathrm{encode}(e_2)\ \rangle \\
\mathrm{encode}(\mu e) &\triangleq \texttt{Mu}\langle\ \mathrm{encode}(e)\ \rangle \\
\mathrm{encode}((e_1, e_2)) &\triangleq \texttt{Pair}\langle\ \mathrm{encode}(e_1), \mathrm{encode}(e_2)\ \rangle \\
\mathrm{encode}(\texttt{fst}\ e) &\triangleq \texttt{Fst}\langle\ \mathrm{encode}(e)\ \rangle \\
\mathrm{encode}(\texttt{snd}\ e) &\triangleq \texttt{Snd}\langle\ \mathrm{encode}(e)\ \rangle \\
\mathrm{encode}(c^\alpha) &\triangleq \texttt{Rep}_\alpha\langle\ c^\alpha\ \rangle \\
\mathrm{encode}(c^\alpha) &\triangleq \texttt{Rep}_\beta\langle\ c^\alpha\ \rangle \\
\mathrm{encode}(\mathrm{op}_i^r(e_1, \ldots, e_n)) &\triangleq \mathrm{op}_i^r\langle\ \mathrm{encode}(e_1), \ldots, \mathrm{encode}(e_n)\ \rangle
\end{aligned}
$$

Figure A.4: Encoding function

in expression whose encoding is on tape t_1 is treated as "look at the contents of t_2." That is, instead of substituting x away for t_2 (which may be infinite), replace x in t_1 with Turing machine code that looks at the contents of t_2.

Now, we describe an oracle Turing machine that realizes the reduction of an expression e. The machine has an oracle tape, an auxiliary working tape which can be treated as a countable number of working tapes, an input tape, and a (one way) output tape. We start with encode(e) on the input tape.

The machine is constructed in the *obvious* way. We just need to ensure that all operations are done in an on-demand fashion as we work with infinite bit-streams. Here, the choice of call-by-name evaluation corresponds nicely with tape contents being computed on-demand.

- interp($\texttt{Var}\langle\ x\ \rangle$): copy the machine code x and (on-demand) copy its result to the output tape.
- interp($\texttt{Nat}\langle\ n\ \rangle$): copy $\texttt{Nat}\langle\ n\ \rangle$ to the output tape.
- interp($\oplus_n\langle\ t\ \rangle$): run a Turing machine realizing $\oplus_n$ with input tape set to the output of running interp(t).
- interp($\texttt{If0}\langle\ t_1, t_2, t_3\ \rangle$): run interp($t_1$) redirecting the result to a temporary working tape. If that tape contains $\texttt{Nat}\langle\ \bar{0}\ \rangle$, run interp($t_2$). Otherwise, run interp(t_3).
- interp($\texttt{Lam}\langle\ t\ \rangle$): (on-demand) copy $\texttt{Lam}\langle\ t\ \rangle$ to the output tape.
- interp($\texttt{App}\langle\ t_1, t_2\ \rangle$): the goal is to produce lshift$\langle$ subst$\langle\ t_1', \text{rshift}\langle\ t_2\ \rangle, 0\ \rangle\ \rangle$, where interp($t_1$) $=$ $\texttt{Lam}\langle\ t_1'\ \rangle$. Run interp($t_1$) until we match the tag `Lam`, where the rest of the tape is $\langle\ t_1'\ \rangle$. Then, run rshift$\langle\ t_2\ \rangle$ which can be done in finite time to produce t_2'. Next, run the environment extension code extend$\langle\ t_1', t_2', 0\ \rangle$. Before we read from the output tape, run lshift.
- interp($\texttt{Mu}\langle\ t\ \rangle$): the goal is to produce lshift$\langle$ subst$\langle\ t, \text{rshift}\langle\ \texttt{Mu}\langle\ t\ \rangle\ \rangle, 0\ \rangle\ \rangle$. This can be done similarly to the application case.
- interp($\texttt{Pair}\langle\ t_1, t_2\ \rangle$): (on-demand) copy $\texttt{Pair}\langle\ t_1, t_2\ \rangle$ to the output tape.
- interp($\texttt{Fst}\langle\ t\ \rangle$): run interp($t$) until we match the tag `Pair`. If we match the tag, run interp(t) further to obtain $\langle\ t_1, \cdot\ \rangle$ and (on-demand) copy t_1 to the output tape.

- interp($\texttt{Snd}\langle\ t\ \rangle$): run interp($t$) until we match the tag `Pair`. If we match the tag, run interp(t) further to obtain $\langle\ \cdot, t_2\ \rangle$ and (on-demand) copy t_2 to the output tape.
- interp($\texttt{Rep}_\alpha\langle\ c^\alpha\ \rangle$): (on-demand) copy $\texttt{Rep}_\alpha\langle\ c^\alpha\ \rangle$ to the output tape.
- interp($\texttt{Rep}_\beta\langle\ c^\alpha\ \rangle$): (on-demand) copy $\texttt{Rep}_\beta\langle\ c^\alpha\ \rangle$ to the output tape.
- interp($\mathrm{op}_n^r\langle\ t_1, \ldots, t_n\ \rangle$): get the Turing machine code for op_n^r (note that the name of op_n^r contains an oracle). By convention, suppose op_n^r has n-input tapes. Set the input tapes to point to each t_i. Finally, run the Turing machine code for op^r, redirecting its output to the output tape.

The operational semantics is realizable by an oracle Turing machine.

Lemma A.5.1 (Realizable by oracle Turing machine)**.** *If* $e \Downarrow v$*, then* interp(encode(e)) = encode(v)*.*

Proof. The proof is by induction on the height of the derivation of $e \Downarrow v$. The cases are straightforward because every step is essentially an identity. □

A.6 Environment Lookup is Continuous

We can finally show that environment lookup is continuous.

Theorem A.6.1 (Environment lookup continuous)**.** *Let* $\Gamma, x : \alpha_x \vdash e : \alpha$*. Then,* $v \mapsto \mathcal{E}[\![\Gamma, x : \alpha_x \vdash e : \alpha]\!]\rho[x \mapsto v] \in \mathscr{C}(\mathcal{V}[\![\alpha_x]\!], \mathcal{V}[\![\alpha]\!])$ *(on its domain) for any well-formed environment* ρ *with respect to* Γ*.*

Proof. It is sufficient to consider when $\mathcal{E}[\![\Gamma, x : \alpha_x \vdash e : \alpha]\!]\rho[x \mapsto v] \neq \bot$ for some v as we only need to show the result on the domain.

Let σ be the corresponding substitution to the environment ρ. By the substitution lemma, $\mathcal{E}[\![\Gamma \vdash e : \alpha]\!]\rho[x \mapsto v] = \mathcal{E}[\![\cdot \vdash (\sigma e)[v/x] : \alpha]\!]$. By adequacy (corollary A.4.4), there is a value v' such that $(\sigma e)[v/x] \Downarrow v'$. Because the operational semantics is realizable by an oracle Turing

machine (lemma A.5.1), we have that interp(encode($(\sigma e)[v/x]$)) = encode(v'). Thus, we have a realizer where the variable x is treated as an environment lookup from a Turing machine tape. Hence, the machine with encode(v) on the environment tape indexed by x realizes $v \mapsto \mathcal{E}[\![\Gamma, x : \alpha_x \vdash e : \alpha]\!]\rho[x \mapsto v]$. □

A.7 Relation to λ_{CD}

λ_{CD} as an instantiation of λ_R. To see this, let $\alpha = \texttt{Real}$ and β be the (countable) collection of types generated by well-formed types $\{\texttt{Samp}\ \tau \mid\vdash_D \tau\}$. The primitive functions on reals correspond with the primitive functions on represented spaces in λ_R. Given a collection of unary functions $(\text{ret}_\tau)_{\vdash_D \tau}$ with type $\tau \to \texttt{Samp}\ \tau$ indexed by well-formed types τ, the expression `return` e corresponds to the λ_R expression $\text{ret}_\tau(e)$, *i.e.*, the application of a function between represented spaces. Given a collection of binary functions $(\text{sample}_{\tau_1,\tau_2})_{\vdash_D \tau_1,\tau_2}$ with type $\texttt{Samp}\ \tau_1 \times \tau_1 \to \texttt{Samp}\ \tau_2$, the expression $x \leftarrow e_1\ ;\ e_2$ corresponds to the λ_R expression $\text{sample}(e_1, \lambda x.\ e_2)$. Note that the second argument $\lambda x.\ e_2$ has the form of the substitution function that we showed was continuous as a map between represented spaces (see lemma A.6.1).

B

AugurV2 Supplementary

This appendix contains supplementary material related to AugurV2. FOOBAR what else?

B.1 Model: HLR

Figure B.1 contains an AugurV2 program encoding a HLR. As a reminder, each x_{nk} (for any $0 \leq n < N$) is called a *predictor* (or feature or attribute). A HLR can be used to construct a classifier, *e.g.*, by determining the posterior distribution on the vector of coefficients `theta` given class labels `y`.

```
(K: Int, N: Int, sigma2: Real, x: Vec (Vec Int)) => {
  param b ~ Normal(0.0, sigma2) ;
  param theta[k] ~ Normal(0.0, sigma2)
      for k <- 0 until K ;
  data y[n] ~ Bernoulli(sigmoid(theta .* x[n] + b))
      for n <- 0 until N ;
}
```

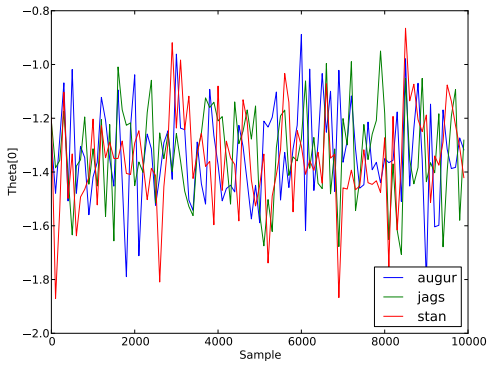
Figure B.1: A HLR model in AugurV2.

DATASETS We use two datasets from the UCI machine learning repository [58]. The German Credit dataset has 24 predictors and 1000 datapoints. The Adult Income dataset has 14 predictors and 48842 datapoints. We preprocess each dataset as follows before feeding it to each system. Discrete predictors are treated numerically, *i.e.*, encoded as a number. Then, we standardize all predictors so that they have mean 0 and variance 1 [cite]. We ignore the datapoints in a dataset with missing predictor values.

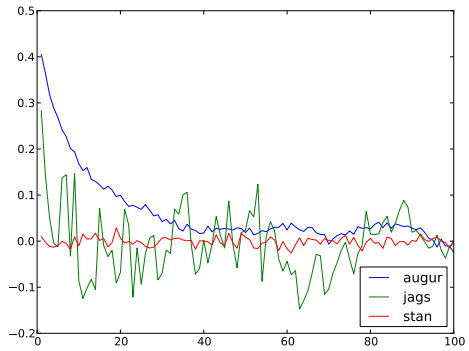
ADDITIONAL EXPERIMENTS Figure B.2 summarizes the results of the additional experiments, including (1) time series and (2) k-lag autocorrelation plots. These plots give a visual diagnostic of the performance of the MCMC inference algorithm that each system generates. To create the plots, we generate 10000 samples from each chain with a burn-in of 1000 samples to estimate each chain's mean and variance. Then, we run a separate chain generating 10000 samples, also with a burn-in of 1000 samples, to estimate the k-lag autocorrelation.

$$\hat{\rho}_k = \frac{1}{\hat{\sigma}^2(S-k)} \sum_{s=1}^{S-1} (\theta^s - \hat{\mu})(\theta^{s+k} - \hat{\mu})$$

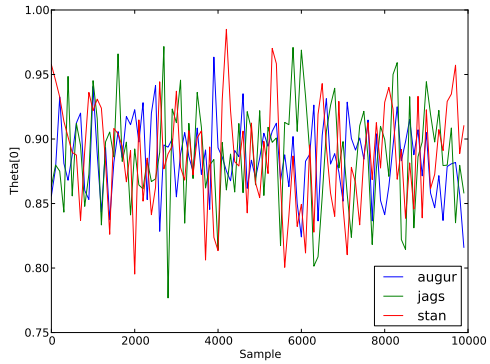
As we can see from the time series plots (Figures B.2a and B.2c) for one the predictors (the other predictors look similar), the Markov chains produced by all three systems seem to have



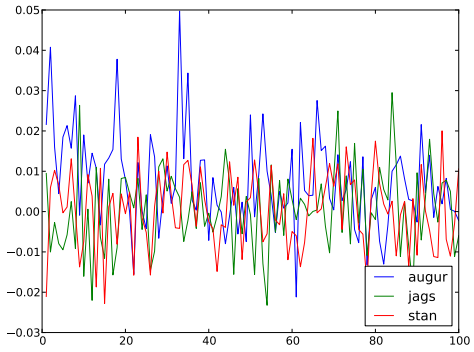
(a) Time series plot (for θ_0) on German credit dataset.



(b) Autocorrelation plot (for θ_0) on German credit dataset.



(c) Time series plot (for θ_0) on Adult Income dataset.



(d) Autocorrelation plot (for θ_0) on Adult Income dataset.

Figure B.2: Additional experimental results for the HLR model.

reached their stationary distribution. Moreover, the spread of the sampled values produced by all three systems also look similar, which increases our confidence that each system implementation of HLR is *correct*, *i.e.*, targeting the correct posterior distribution.

The empirical autocorrelation function (Figures B.2b and B.2d) provides a visual diagnostic of the performance of a MCMC inference algorithm. Theoretically, the autocorrelation function (of a reversible Markov Chain) should be a convex, decreasing function [cite]. Intu-

itively, at higher lags between states, there should be less correlation between them. On the German Credit dataset, the empirical autocorrelation function exhibited by AugurV2 has a theoretically expected shape. Jags and Stan produce much noisier shapes. On the Adult Income dataset, all three systems exhibit noisier autocorrelation shapes.

We attempted to estimate the effective sample size (ESS), but we found most of the autocorrelation functions to be too noisy to provide a meaningful estimate. Hoffman *et al.* [45] provide a method for estimating the ESS of a HLR. However, the method is invalid due to its treatment of negative autocorrelation (see [cite]). We also tried other methods such as the initial convex sequence (ICS) estimator [cite] and initial positive sequence (IPS) estimator [cite] that deal with negative autocorrelation for reversible Markov chains. However, a visual diagnostic of the autocorrelation shows that the empirical estimates are too noisy. Consequently, we have just provided time series and autocorrelation plots as a visual diagnositic of each system's stationarity and mixing properties.

B.2 HGMM

Figure B.3 contains an AugurV2 program encoding a HGMM. Compared to the GMM model that we have used as a running example throughout this dissertation, the HGMM has priors on the mixture proportions, the cluster means, and the cluster covariance matrices. Consequently, it is a hierarchical version of the GMM model.

Datasets We use synthetic datasets with a varying number of mixtures and observed points for our HMVGMM experiments to assess how well each system's inference efficiency scales with the size of the data and model. As a proxy to determine the overall efficiency of each system, we estimate the predictive probability on a held-out test set as a function of wall-

```
(K, N, mu0, sigma0, df, scale0, alpha) => {
  param pii ~ Dirichlet(alpha) ;
  param mu[k] ~ MvNormal(mu0, sigma0)
     for k <- 0 until K ;
  param sigma[k] ~ IWishart(df, scale0)
     for k <- 0 until K ;
  param z[n] ~ Categorical(pii)
     for n <- 0 until N ;
  data y[n] ~ MvNormal(mu[z[n]], sigma[z[n]])
     for n <- 0 until N ;
}
```

Figure B.3: A HGMM model in AugurV2.

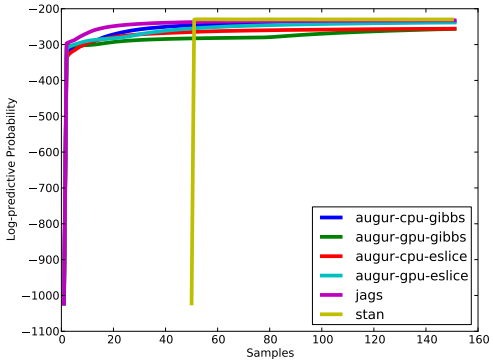
clock time.[1]

Additional Experiments To compute the predictive probability, we first use each system to draw S samples ${}^s\pi$, ${}^s\mu_k$, and ${}^s\sigma_k$ (for $1 \leq s \leq S$). Then, we estimate the predictive probability of M held-out datapoints $\widetilde{y}_m$ as:

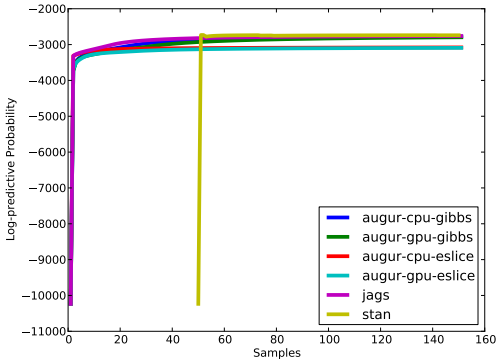
$$\prod_{m=1}^{M} \sum_{s=1}^{S} \sum_{k=1}^{K} p(\widetilde{y}_m \mid^s \mu_k, {}^s\sigma_k^2) p(k; {}^s\pi) \ .$$

Figure B.4 provides plots of the log-predictive probability for the first 150 samples as a function of wall-clock time for each system and for different settings. The compile time has been included. Note that Stan requires an initial tuning period of 50 samples that we discard for the purposes of computing the log predictive-probability.

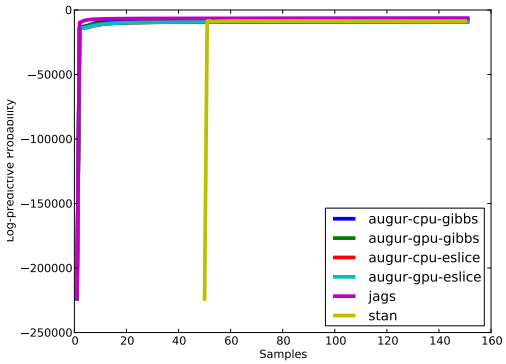
[1]The predictive probability can be measured as a function of MCMC *steps*. For example, a MCMC sampler on a 2D space that samples each dimension in sequence takes two steps to generate one sample. Hence, this metric more directly measures the algorithm's statistical efficiency because (1) the time to draw a sample is removed and (2) it penalizes unblocked proposals.



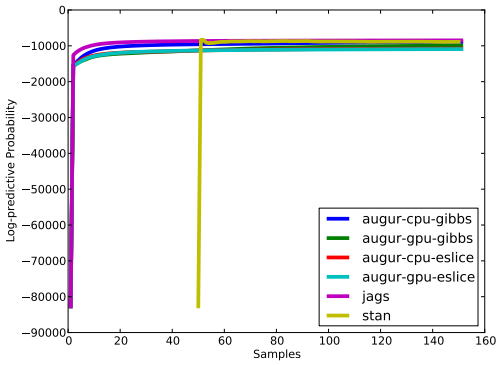
(a) HMVGMM with $K = 3$, $D = 2$, and 1000 datapoints.



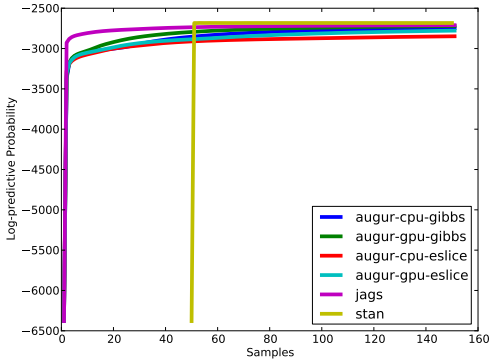
(b) HMVGMM with $K = 3$, $D = 2$, and 10000 datapoints.



(c) HMVGMM with $K = 3$, $D = 10$, and 10000 datapoints.



(d) HMVGMM with $K = 10$, $D = 10$, and 10000 datapoints.



(e) HMVGMM with $K = 10$, $D = 2$, and 10000 datapoints.

Figure B.4: Plots of the log predictive-probability with respect to (wall-clock) time for each system and for different settings. We use each system to draw 150 samples. Note that Stan requires an initial tuning period of 50 samples that we discard for the purposes of computing the log predictive-probability.

System	Compile	Sample (150)
AugurV2-cpu-gibbs	0.449	0.16894
AugurV2-gpu-gibbs	8.271	0.470396
AugurV2-cpu-eslice	0.468	0.669486
AugurV2-cpu-eslice	10.547	1.798497
Jags	0.131	1.116865
Stan	42.454	7.875823

(a) HMVGMM with $K = 3$, $D = 2$, and 1000 datapoints.

System	Compile	Sample (150)
AugurV2-cpu-gibbs	0.607	1.445688
AugurV2-gpu-gibbs	8.575	7.428644
AugurV2-cpu-eslice	0.621	6.441652
AugurV2-cpu-eslice	10.998	8.37323
Jags	1.724	17.423054
Stan	42.71	165.53241

(b) HMVGMM with $K = 3$, $D = 2$, and 10000 datapoints.

System	Compile	Sample (150)
AugurV2-cpu-gibbs	0.668	15.582646
AugurV2-gpu-gibbs	8.622	24.001733
AugurV2-cpu-eslice	0.672	107.341997
AugurV2-cpu-eslice	10.951	59.682671
Jags	1.877	92.996583
Stan	42.635	297.939999

(c) HMVGMM with $K = 3$, $D = 10$, and 10000 datapoints.

System	Compile	Sample (150)
AugurV2-cpu-gibbs	0.726	48.496309
AugurV2-gpu-gibbs	8.626	19.170607
AugurV2-cpu-eslice	0.726	360.218174
AugurV2-cpu-eslice	11.308	127.631192
Jags	3.464	301.868114
Stan	42.609	60407.637089

(d) HMVGMM with $K = 10$, $D = 10$, and 10000 datapoints.

System	Compile	Sample (150)
AugurV2-cpu-gibbs	0.613	3.65864
AugurV2-gpu-gibbs	8.687	3.180839
AugurV2-cpu-eslice	0.612	20.382706
AugurV2-cpu-eslice	10.987	7.961469
Jags	3.329	51.534129
Stan	42.542	12678.632276

(e) HMVGMM with $K = 10$, $D = 2$, and 10000 datapoints.

B.3 Model: LDA

Figure **??** summarizes the Latent Dirichlet Allocation model (LDA) and gives the corresponding AugurV2 program. LDA can be used to learn the topics in a corpus of documents. As a reminder, each ϕ_k gives the proportion of words in each topic and each θ_d gives the proportion of topics in each document. The variable z_{dw} gives the topic assignment of each word w_{dj} in a document.

Datasets We used two datasets from the UCI repository for our LDA experiments. The first is the KOS dataset, which contains approximately 3.4K documents, 470K words, and 6.9K vocabulary words. The second is the NIPS dataset, which contains approximately 1.5K documents, 1.9M words, and 12.4K vocabulary words. The datasets come preprocessed with common stop words removed.

Additional Experiments We measure the statistical efficiency of each system by computing the log-predictive probability of a held-out test dataset.[2] For clarity, we now summarize operationally how we compute the log-predictive probability, and refer the reader to the references above for their derivations.

1. Let $\vec{w}$ be the (entire) corpus of words. We split this into two disjoint sets, a training set $\vec{w}^{\mathbf{train}}$ and a testing set $\vec{w}^{\mathbf{test}}$. We use a 90/10 split.
2. We split the words in each document in the testing set into an observed set $\vec{w}^{\mathbf{test}}_{\mathbf{obs}}$ and a held-out set $\vec{w}^{\mathbf{test}}_{\mathbf{ho}}$. We ensure that the held-out words and the observed words in each

[2] There are many other metrics that have been proposed to evaluate both the LDA model and a corresponding inference algorithm (*e.g.*, see [44]). We choose this one because it rewards the algorithm which learns the best set of parameters for *prediction* no matter *how* it is computed. Hence, an oracle that knows the best setting of parameters for prediction for the given training and testing set will do the best. As we are keeping the model fixed and vary the inference algorithm, this should be a proxy for measuring how well an inference algorithm is performing.

```
(D: Int, K: Int, N: Int, alpha: Vec Real, beta: Vec Real) => {
  param theta[d] ~ Dirichlet(alpha)
      for d <- 0 until D ;
  param phi[k] ~ Dirichlet(beta)
      for k <- 0 until K ;
  param z[d, j] ~ Categorical(theta[d])
      for d <- 0 until D, j <- 0 until N[d] ;
  data w[d, j] ~ Categorical(phi[z[d, w]])
      for d <- 0 until D, j <- 0 until N[d] ;
}
```

Figure B.6: The LDA model in AugurV2.

document are disjoint (for that document). We use a 90/10 split.

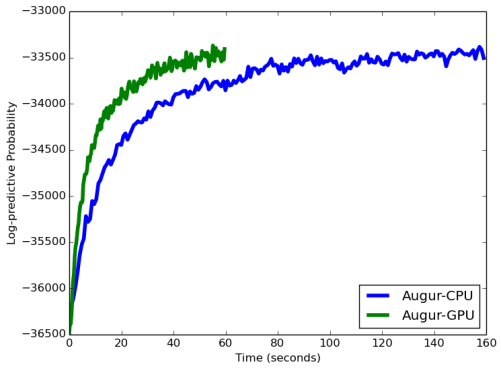
3. We run an inference algorithm to infer the distribution of words over topics using the training set $\vec{w}^{\mathbf{train}}$, *i.e.*, we target $p(\phi, \theta, z \mid \vec{w}^{\mathbf{train}})$ to infer a value $\hat{\phi}$.

4. We run an inference algorithm to infer the topic assignments using the observed testing set $\vec{w}^{\mathbf{test}}_{\mathbf{obs}}$, *i.e.*, we target $p(\theta, z \mid \hat{\phi}, \vec{w}^{\mathbf{test}}_{\mathbf{obs}})$ to infer $\hat{z}$.

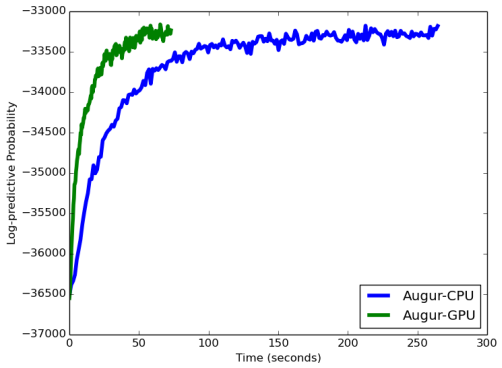
5. The predictive probability of the held-out testing set is computed as:

$$\sum_{s} \prod_{d} \prod_{j} \sum_{k} p(\phi_k) p(\theta^s_d) p(z_{dj} = k \mid \theta^s_d) p(w^{\mathbf{test}}_{\mathbf{ho},dj} \mid \phi_k)$$

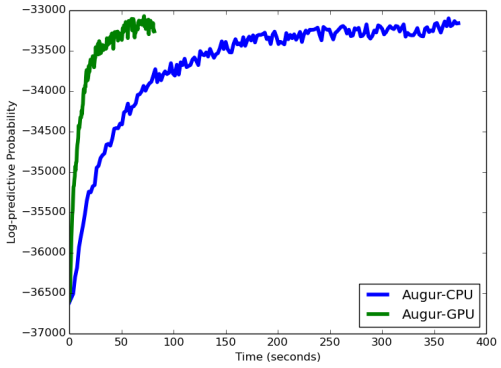
Figure B.7 shows the log-predictive probability of AugurV2's CPU and GPU inference algorithms as a function of wall-clock time.



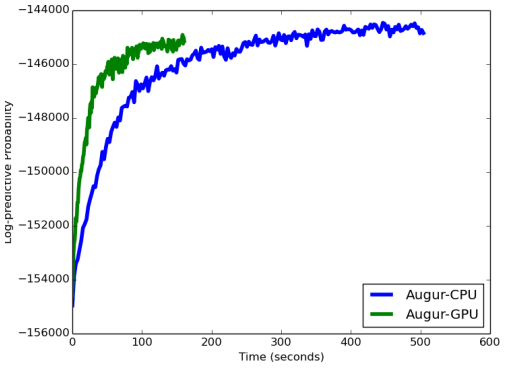
(a) Kos with 50 topics



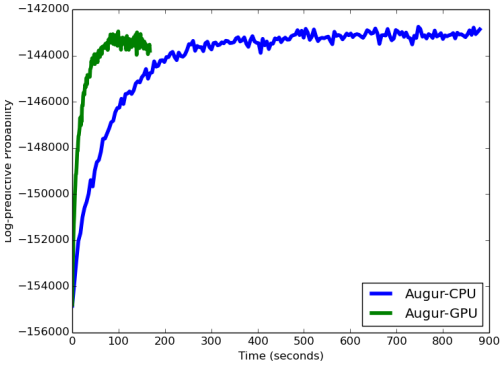
(b) Kos with 100 topics



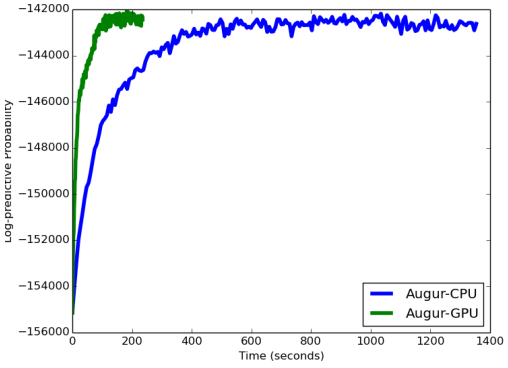
(c) Kos with 150 topics



(d) Nips with 50 topics



(e) Nips with 100 topics



(f) Nips with 150 topics

Figure B.7: The log-predictive probability of AugurV2's CPU verus GPU inference on LDA as a function of wall-clock time. The samplers are stopped after we obtain 200 samples. We do not include time for compilation, which is reported separately.

References

[1] MIT Press, 2007.

[2] Hakaru, 2016.

[3] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[4] Abramsky, S., and Jung, A. Handbook of logic in computer science. vol. 3. Oxford University Press, 1994, ch. Domain Theory, pp. 1–168.

[5] Ackerman, N. L., Freer, C. E., and Roy, D. M. Noncomputable conditional distributions. In *Proceedings of the 26th Annual Symposium on Logic in Computer Science* (2011), IEEE, pp. 107–116.

[6] Andrieu, C., Doucet, A., and Holenstein, R. Particle markov chain monte carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology) 72*, 3 (2010), 269–342.

[7] Aumann, R. J. Borel Structures for Function Spaces. *Illinois Journal of Mathematics 5*, 4 (1961), 614–630.

[8] Bailey, D., Borwein, P., and Plouffe, S. On the rapid computation of various polylogarithmic constants. *Mathematics of Computation 66*, 218 (1997), 903–913.

[9] Bartholomew-Biggs, M., Brown, S., Christianson, B., and Dixon, L. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics 124*, 1 (2000), 171–190.

[10] Battenfeld, I. *A category of topological predomains*. PhD thesis, Diploma Thesis, TU Darmstadt, 2004.

[11] Battenfeld, I. *Topological Domain Theory*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 2008.

[12] Battenfeld, I., Schröder, M., and Simpson, A. Compactly generated domain theory. *Mathematical Structures in Computer Science 16*, 2 (Apr. 2006), 141–161.

[13] Battenfeld, I., Schröder, M., and Simpson, A. A convenient category of domains. *Electronic Notes in Theoretical Computer Science 172* (2007), 69–99.

[14] Bauer, A. *The Realizability Approach to Computable Analysis and Topology*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 2000.

[15] Billingsley, P. *Probability and Measure*, 3 ed. Wiley, 1995.

[16] Bonawitz, K. A. *Composable Probabilistic Inference with Blaise*. PhD thesis, Massachusetts Institute of Technology, 2008.

[17] Borgström, J., Gordon, A. D., Greenberg, M., Margetson, J., and Van Gael, J. Measure transformer semantics for bayesian machine learning. In *Programming Languages and Systems* (2011), Springer, pp. 77–96.

[18] Brooks, S., Gelman, A., Jones, G. L., and Meng, X.-L., Eds. *Handbook of Markov Chain Monte Carlo*, 1 ed. Chapman and Hall/CRC, 2011.

[19] Carpenter, B., Lee, D., Brubaker, M. A., Riddell, A., Gelman, A., Goodrich, B., Guo, J., Hoffman, M., Betancourt, M., and Li, P. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* (2016). in press.

[20] Chang, J. T., and Pollard, D. Conditioning as Disintegration. *Statistica Neerlandica 51*, 3 (1997), 287–317.

[21] Danos, V., and Ehrhard, T. Probabilistic coherence spaces as a model of higher-order probabilistic computation. *Information and Computation 209*, 6 (2011), 966–991.

[22] De Raedt, L., Kimmig, A., and Toivonen, H. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI* (2007), vol. 7, pp. 2462–2467.

[23] Domingos, P., and Richardson, M. Markov Logic: A Unifying Framework for Statistical Relational Learning. In *Introduction to Statistical Relational Learning* [1], pp. 339–371.

[24] Domingos, P. M., and Webb, W. A. A Tractable First-Order Probabilistic Logic. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence* (2012), AAAI, pp. 1902–1909.

[25] Durbin, R., Eddy, S. R., Krogh, A., and Mitchison, G. *Biological sequence analysis: Probabilistic models of proteins and nucleic acids*. Cambridge University Press, 1998.

[26] Edalat, A. Domain theory and integration. In *Logic in Computer Science, 1994. LICS'94. Proceedings., Symposium on* (1994), IEEE, pp. 115–124.

[27] Edalat, A. A computable approach to measure and integration theory. In *Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on* (2007), IEEE, pp. 463–472.

[28] Escardó, M., Lawson, J., and Simpson, A. Comparing cartesian closed categories of (core) compactly generated spaces. *Topology and its Applications 143*, 1 (2004), 105–145.

[29] Faissole, F., and Spitters, B. Synthetic topology in homotopy type theory for probabilistic programming. Workshop on probabilistic programming semantics (PPS 2017), Jan. 2017. Poster.

[30] Freer, C. E., and Roy, D. M. Posterior distributions are computable from predictive distributions. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* [87], pp. 233–240.

[31] Freer, C. E., and Roy, D. M. Computable de finetti measures. *Annals of Pure and Applied Logic 163*, 5 (2012), 530–546.

[32] Gács, P. Uniform test of algorithmic randomness over a general space. *Theoretical Computer Science 341*, 1 (2005), 91–137.

[33] Galatolo, S., Hoyrup, M., and Rojas, C. Effective symbolic dynamics, random points, statistical behavior, complexity and entropy. *Information and Computation 208*, 1 (2010), 23–41.

[34] Gehr, T., Misailovic, S., and Vechev, M. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification* (2016), S. Chaudhuri and A. Farzan, Eds., Springer International Publishing, pp. 62–83.

[35] Gelman, A., Carlin, J. B., Stern, H. S., and Rubin, D. B. *Bayesian Data Analysis*, vol. 2. Chapman & Hall/CRC Boca Raton, FL, USA, 2014.

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

[37] Gierz, G., Hofmann, K. H., Keimel, K., Lawson, J. D., Mislove, M., and Scott, D. S. *Continuous lattices and domains*, vol. 93. Cambridge University Press, 2003.

[38] Giry, M. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis* (1982), Springer Berlin Heidelberg, pp. 68–85.

[39] Goodman, N., Mansinghka, V., Roy, D. M., Bonawitz, K., and Tenenbaum, J. B. Church: A language for generative models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence* (2008), AUAI, pp. 220–229.

[40] Gordon, A. D., Graepel, T., Rolland, N., Russo, C., Borgström, J., and Guiver, J. Tabular: A Schema-driven Probabilistic Programming Language. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2014), ACM, pp. 321–334.

[41] Green, P. J. Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika 82*, 4 (1995), 711–732.

[42] Gunter, C. A. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

[43] Hershey, S., Bernstein, J., Bradley, B., Schweitzer, A., Stein, N., Weber, T., and Vigoda, B. Accelerating inference: towards a full language, compiler and hardware stack. *CoRR abs/1212.2991* (2012).

[44] Hoffman, M. D., Blei, D. M., Wang, C., and Paisley, J. W. Stochastic variational inference. *Journal of Machine Learning Research 14*, 1 (2013), 1303–1347.

[45] Hoffman, M. D., and Gelman, A. The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research 15*, 1 (2014), 1593–1623.

[46] Hoyrup, M., and Rojas, C. Computability of probability measures and martin-löf randomness over metric spaces. *Information and Computation 207*, 7 (2009), 830–847.

[47] Huang, D., and Morrisett, G. An Application of Computable Distributions to the Semantics of Probabilistic Programming Languages. In *Programming Languages and Systems* (2016), pp. 337–363.

[48] III, H. D. Hierarchical Bayesian Compiler. http://www.umiacs.umd.edu/~hal/HBC/, 2008.

[49] Jones, C. *Probabilistic Non-determinism*. PhD thesis, University of Edinburgh, 1989. UMI Order No. GAXDX-94930.

[50] Jones, C., and Plotkin, G. D. A Probabilistic Powerdomain of Evaluations. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science* (1989), IEEE, pp. 186–195.

[51] Jung, A., and Tix, R. The Troublesome Probabilistic Powerdomain. *Electronic Notes in Theoretical Computer Science 13* (1998), 70–91.

[52] Koller, D., and Friedman, N. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009.

[53] Koller, D., McAllester, D., and Pfeffer, A. Effective Bayesian Inference for Stochastic Programs. In *Proceedings of the 14th AAAI Conference on Artificial Intelligence* (1997), AAAI.

[54] Kozen, D. Semantics of probabilistic programs. *Journal of Computer and System Sciences 22*, 3 (1981), 328–350.

[55] Kucukelbir, A., Ranganath, R., Gelman, A., and Blei, D. Automatic Variational Inference in Stan. In *Advances in Neural Information Processing Systems* (2015), pp. 568–576.

[56] Lago, U. D., and Zorzi, M. Probabilistic operational semantics for the lambda calculus. *RAIRO-Theoretical Informatics and Applications 46*, 3 (2012), 413–450.

[57] Leitz, P. *From Constructive Mathematics to Computable Analysis via the Realizability Interpretation*. PhD thesis, TU Darmstadt, 2004.

[58] Lichman, M. UCI Machine Learning Repository, 2013.

[59] Longley, J. R. *Realizability Toposes and Language Semantics*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1995.

[60] Lunn, D., Spiegelhalter, D., Thomas, A., and Best, N. The BUGS project: Evolution, critique and future directions. *Statistics in Medicine 28*, 25 (2009), 3049–3067.

[61] Mansinghka, V. K., Selsam, D., and Perov, Y. N. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR abs/1404.0099* (2014).

[62] McCallum, A., Schultz, K., and Singh, S. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems 23* (2009), Neural Information Processing Systems Foundation, pp. 1249–1257.

[63] Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D. L., and Kolobov, A. Blog: Probabilistic Models with Unknown Objects. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence* (2005), IJCAII, pp. 1352–1359.

[64] Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D. L., and Kolobov, A. BLOG: Probabilistic Models with Unknown Objects. [1], pp. 373–398.

[65] Minka, T., Winn, J., Guiver, J., Webster, S., Zaykov, Y., Yangel, B., Spengler, A., and Bronskill, J. Infer.NET 2.6, 2014. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[66] Muggleton, S. Stochastic logic programs. In *New Generation Computing* (1996), Academic Press.

[67] Munkres, J. R. *Topology*. Prentice Hall, 2000.

[68] Murray, I., Adams, R. P., and MacKay, D. J. Elliptical slice sampling. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* [87], pp. 541–548.

[69] Narayanan, P., Carette, J., Romano, W., Shan, C.-c., and Zinkov, R. Probabilistic inference by program transformation in hakaru (system description). In *International Symposium on Functional and Logic Programming* (2016), Springer, pp. 62–79.

[70] Neal, R. M. Markov chain sampling methods for dirichlet process mixture models. *Journal of computational and graphical statistics 9*, 2 (2000), 249–265.

[71] Nori, A. V., Hur, C.-K., Rajamani, S. K., and Samuel, S. R2: An efficient mcmc sampler for probabilistic programs. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence* (2010), AAAI, pp. 2476–2482.

[72] Ntzoufras, I. Bayesian Modeling Using WinBUGS, 2009.

[73] Paige, B., and Wood, F. A Compilation Target for Probabilistic Programming Languages. In *Proceedings of the 31st International Conference on Machine Learning* (2014), JMLR, pp. 1935–1943.

[74] Panangaden, P. Probabilistic relations. In *MIV98* (1998), Citeseer.

[75] Park, S., Pfenning, F., and Thrun, S. A probabilistic language based upon sampling functions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2005), ACM, pp. 171–182.

[76] Pauly, A. On the topological aspects of the theory of represented spaces. *Computability*, Preprint (2012), 1–22.

[77] Peyton Jones, S., Reid, A., Henderson, F., Hoare, T., and Marlow, S. A Semantics for Imprecise Exceptions. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation* (1999), ACM, pp. 25–36.

[78] Pfeffer, A. IBAL: A Probabilistic Rational Programming Language. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence* (2001), IJCAII, pp. 733–740.

[79] Pfeffer, A. The Design and Implementation of IBAL: A General-Purpose Probabilistic Language. In *Introduction to Statistical Relational Learning* [1], pp. 399–433.

[80] Pfeffer, A. Figaro: An object-oriented probabilistic programming language. Tech. rep., 2009.

[81] Plotkin, G. D. LCF considered as a programming language. *Theoretical Computer Science 5*, 3 (1977), 223–255.

[82] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2013), ACM, pp. 519–530.

[83] Ramsey, N., and Pfeffer, A. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2002), ACM, pp. 154–165.

[84] Rao, M. M., and Swift, R. J. *Probability theory with applications*, vol. 582. Springer US, 2006.

[85] Saheb-Djahromi, N. Probabilistic LCF. *Mathematical Foundations of Computer Science 64* (1978), 442–451.

[86] Saheb-Djahromi, N. CPO's of measures for nondeterminism. *Theoretical Computer Science 12*, 1 (1980), 19–37.

[87] SAIS. *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (2010).

[88] Schröder, M. Admissible Representations of Probability Measures. *Electronic Notes in Theoretical Computer Science 167* (2007), 61–78.

[89] Sipser, M.

[90] Staton, S., Yang, H., Wood, F., Heunen, C., and Kammar, O. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (2016), ACM, pp. 525–534.

[91] Streicher, T. Realizability. http://www.mathematik.tu-darmstadt.de/ streicher/REAL/REAL.pdf. Course Lecture Notes.

[92] Team, S. D. Stan Modeling Language: User's Guide and Reference Manual. http://mc-stan.org/documentation/, 2015.

[93] Teh, Y. W. Dirichlet Process. In *Encyclopedia of Machine Learning*. Springer, 2011, pp. 280–287.

[94] Thomas, A., Spiegelhalter, D. J., and Gilks, W. R. BUGS: A program to perform Bayesian inference using Gibbs sampling. *Bayesian Statistics 4*, 9 (1992), 837–842.

[95] Thrun, S., Burgard, W., and Fox, D. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[96] Toronto, N., and McCarthy, J. Computing in Cantor's Paradise with λ ZFC. In *International Symposium on Functional and Logic Programming* (2012), Springer, pp. 290–306.

[97] Toronto, N., McCarthy, J., and Van Horn, D. Running probabilistic programs backwards. In *Programming Languages and Systems*. 2015, pp. 53–79.

[98] Tran, D., Kucukelbir, A., Dieng, A. B., Rudolph, M., Liang, D., and Blei, D. M. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787* (2016).

[99] Tristan, J.-B., Huang, D., Tassarotti, J., Pocock, A. C., Green, S., and Steele, G. L. Augur: Data-Parallel Probabilistic Modeling. In *Advances in Neural Information Processing Systems 28* (2014), Neural Information Processing Systems Foundation, pp. 2600–2608.

[100] Van Oosten, J. *Realizability: An Introduction to its Categorical Side*, vol. 152. Elsevier, 2008.

[101] Weihrauch, K. Computability on the probability measures on the borel sets of the unit interval. *Theoretical Computer Science 219*, 1 (1999), 421–437.

[102] Weihrauch, K. *Computable analysis: an introduction*. Springer Science & Business Media, 2000.

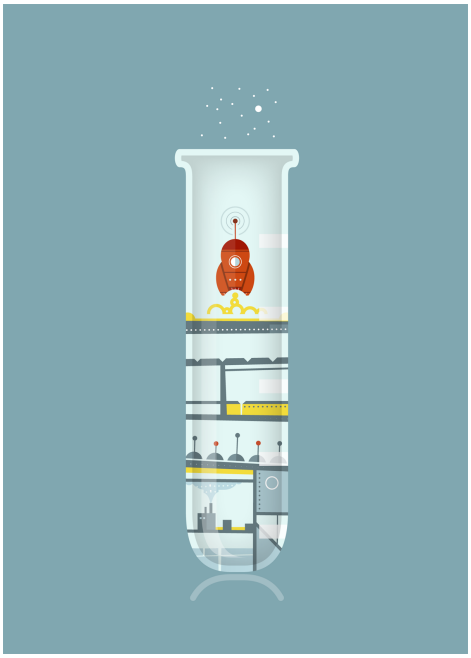
[103] Wingate, D., Stuhlmüller, A., and Goodman, N. D. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics* (2011), SAIS, pp. 770–778.

[104] Winskel, G. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[105] Wood, F., van de Meent, J. W., and Mansinghka, V. A new approach to probabilistic programming inference. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics* (2014), SAIS, pp. 2–46.

[106] Wu, Y., Li, L., Russell, S., and Bodík, R. Swift: Compiled inference for probabilistic programming languages. *CoRR abs/1606.09242* (2016).

[107] Yao, A. C.-C. Probabilistic computations: Toward a unified measure of complexity. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science* (1977), IEEE, pp. 222–227.



This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Egenolff-Berner Garamond, a revival of Claude Garamont's humanist typeface. The above illustration, *Science Experiment 02*, was created by Ben Schlitter and released under CC BY-NC-ND 3.0. A template that can be used to format a PhD dissertation with this look & feel has been released under the permissive AGPL license, and can be found online at github.com/asm-products/Dissertate or from its lead author, Jordan Suchow, at suchow@post.harvard.edu.