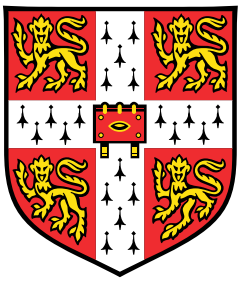
Formally justified and modular Bayesian inference for probabilistic programs



Adam Michał Ścibior

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Doctor of Philosophy

Trinity College September 2018

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as specified in the text.

It is not substantially the same as any that I have submitted, or, is being concurrently submitted for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution. I further state that no substantial part of my dissertation has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution.

This dissertation meets the word limit for the Engineering Degree Committee. Specifically, it does not exceed 65,000 words, including appendices, bibliography, footnotes, tables and equations, and it does not contain more than 150 figures.

Abstract

Formally justified and modular Bayesian inference for probabilistic programs

Adam Michał Ścibior

Probabilistic modelling offers a simple and coherent framework to describe the real world in the face of uncertainty. Furthermore, by applying Bayes' rule it is possible to use probabilistic models to make inferences about the state of the world from partial observations. While traditionally probabilistic models were constructed on paper, more recently the approach of probabilistic programming enables users to write the models in executable languages resembling computer programs and to freely mix them with deterministic code.

It has long been recognised that the semantics of programming languages is complicated and the intuitive understanding that programmers have is often inaccurate, resulting in difficult to understand bugs and unexpected program behaviours. Programming languages are therefore studied in a rigorous way using formal languages with mathematically defined semantics. Traditionally formal semantics of probabilistic programs are defined using exact inference results, but in practice exact Bayesian inference is not tractable and approximate methods are used instead, posing a question of how the results of these algorithms relate to the exact results. Correctness of such approximate methods is usually argued somewhat less rigorously, without reference to a formal semantics.

In this dissertation we formally develop denotational semantics for probabilistic programs that correspond to popular sampling algorithms often used in practice. The semantics is defined for an expressive typed lambda calculus with higher-order functions and inductive types, extended with probabilistic effects for sampling and conditioning, allowing continuous distributions and unbounded likelihoods. It makes crucial use of the recently developed formalism of quasi-Borel spaces to bring all these elements together. We provide semantics corresponding to several variants of Markov chain Monte Carlo and Sequential Monte Carlo

methods and formally prove a notion of correctness for these algorithms in the context of probabilistic programming.

We also show that the semantic construction can be directly mapped to an implementation using established functional programming abstractions called monad transformers. We develop a compact Haskell library for probabilistic programming closely corresponding to the semantic construction, giving users a high level of assurance in the correctness of the implementation. We also demonstrate on a collection of benchmarks that the library offers performance competitive with existing systems of similar scope.

An important property of our construction, both the semantics and the implementation, is the high degree of modularity it offers. All the inference algorithms are constructed by combining small building blocks in a setup where the type system ensures correctness of compositions. We show that with basic building blocks corresponding to vanilla Metropolis-Hastings and Sequential Monte Carlo we can implement more advanced algorithms known in the literature, such as Resample-Move Sequential Monte Carlo, Particle Marginal Metropolis-Hastings, and Sequential Monte Carlo squared. These implementations are very concise, reducing the effort required to produce them and the scope for bugs. On top of that, our modular construction enables in some cases deterministic testing of randomised inference algorithms, further increasing reliability of the implementation.

Acknowledgements

I would like to thank my supervisors, Zoubin Ghahramani and Bernhard Schölkopf, for having me as their student, for the research we did together, for letting me freely explore my ideas, and for all the wisdom they shared with me. My PhD was a truly transformative experience that I enjoyed immensely.

During my PhD I have had the pleasure of collaborating with many great scientists from research institutions around the world. I am especially grateful to Ohad Kammar for guiding me through the process of converting the core ideas underlying this thesis from vague concepts existing only in my head to a well-articulated contribution to knowledge and for teaching me everything I know about semantics of programming languages. I am grateful to all my collaborators who are too many to list here, but I would like to particularly thank Andrew Gordon for helping me write my first research paper and hosting me at Microsoft Research for a summer internship, as well as Ilya Tolstikhin and Carl-Johann Simon-Gabriel with whom I wrote my first NIPS paper.

I was fortunate to do my PhD at two great research intitutes, the University of Cambridge and MPI Tübingen. At both places I learned a lot and had great discussions with countless people - I would like to thank all of them and generally everyone who contributed to building the excellent research environments I had the privilage to be a part of. I am particularly grateful to everyone involved in the Cambridge-Tübingen programme that made this whole experience possible.

I have spent in total seven years in Cambridge during my undergraduate and PhD degrees. I have made many great friends over these years and watched most of them move away to start the next chapter of their lives. Now that it is time for me to go, I would like to thank everyone who made my stay in Cambridge enjoyable and with whom we forged great memories. I can not name all my friends here so I will only name four who particularly deserve it and were with me as I crossed the finish line: Piotr Wieprzowski, Paweł Budzianowski, Sławomir Tadeja, Michał Bogdan.

Finally, I would like to thank my parents for raising me in a way that made all of this possible, and for their continuing support of my life choices even when that means spending a lot of time away from them.

Table of contents

Chapter 1

Introduction

1.1 Bayesian Modelling

Computational models of natural phenomena are commonly applied throughout science and engineering. While often these are deterministic, in many cases it is beneficial to introduce probabilities into the model. It may be because the underlying process being modelled is indeed random, such as interactions in particle physics, or because it is not feasible to model the true process in sufficient detail, such as in analysis of car traffic.

The canonical task in probabilistic models is to draw inferences about unobserved properties of the system given some observations. For example, we might observe that a lawn is wet and ask whether it rained earlier in the day. This is in fact a classic toy problem used to explain probabilistic modelling, presented here in Figure 1.1. The model includes a sprinkler as an alternative mechanism for making the lawn wet. In this version of the model the sprinkler operates independently of the rain and the probability of the lawn being wet depends on both of them.

$$R \sim Bernoulli(0.2)$$
$$S \sim Bernoulli(0.1)$$
$$W \sim p(W|S,R)$$

(a) Equations

		p(W = T | S, R)
R = T	S = T	0.99
R = T	S = F	0.70
R = F	S = T	0.90
R = F	S = F	0.01

(b) Conditional probability table

Figure 1.1: The sprinkler model. R stands for rain, S for sprinkler, and W for the lawn being wet.

The conditional probability of rain given the observation is obtained by the application of Bayes' rule. Thus the task of computing the distribution of latent variables in probabilistic models conditionally on the observations is called Bayesian inference. Traditionally the distribution $p(R)$ considered before the observation is made, in this case $Bernoulli(0.2)$, is called the prior. The posterior $p(R|W)$ is the distribution that takes the observation into account and it is the primary quantity of interest in probabilistic models. It is computed by combining the prior and the likelihood $p(W|R)$ as

$$p(R|W) = \frac{p(R,W)}{p(W)} = \frac{p(W|R)p(R)}{p(W)} = \frac{\sum_S p(W|S,R)p(S)p(R)}{\sum_{S,R} p(W|S,R)p(S)p(R)}.$$

The numerator in this expression, taken as a function of R, is an unnormalised posterior distribution which tells use the relative probabilities of $R = T$ and $R = F$ given the observation. The denominator is the normalising constant known as the marginal likelihood or the model evidence. It tells us how likely the observation is under the model and it is of independent interest from the posterior for the task of model criticism.

In this simple case we can compute both expressions exactly

$$\begin{aligned} p(R = T|W = T) &= \frac{0.99 \cdot 0.1 \cdot 0.2 + 0.70 \cdot 0.9 \cdot 0.2}{0.99 \cdot 0.1 \cdot 0.2 + 0.70 \cdot 0.9 \cdot 0.2 + 0.90 \cdot 0.1 \cdot 0.8 + 0.01 \cdot 0.90 \cdot 0.80} \\ &= \frac{0.1458}{0.225} = 0.648. \end{aligned}$$

Probabilistic models are commonly used in statistics and machine learning. In statistics computing probabilities is of principal importance, while in machine learning probabilistic methods offer certain advantages for prediction such as preventing overfitting and providing uncertainty estimates. In the context of this dissertation the most important advantage of probabilistic models and Bayesian methods is that they offer a simple and coherent framework which can be used as a basis to build general-purpose tools for modelling and inference.

For an in-depth introduction to Bayesian methods there are many excellent textbooks [7, 53, 6].

1.2 Probabilistic Programming

Traditionally probabilistic models are written down on paper using informal mathematical notation, the relevant equations for performing inference are derived by hand, and then a program for computing the posterior is produced in an ad-hoc fashion. Since this process

is tedious and error prone, it is natural to try to devise a formal language for specifying probabilistic models from which a program that performs inference in that model can be synthesised automatically. This results in a domain-specific language (DSL) for writing models and has been done successfully for a long time. BUGS [30] was arguably the first popular system of this kind.

However, the users of DSLs for Bayesian modelling often find that they need to interface their models with traditional programs, for example in order to reuse existing simulator code or to integrate their model as a part of a larger application. Moreover, as the DSL incorporates more features that make writing models easier it often starts closely resembling a programming language. It is therefore reasonable to create a modelling language by extending an existing programming language with basic probabilistic operations. We call this approach probabilistic programming, although many authors take this term to be broader in scope.

Probabilistic programs are thus normal computer programs with two extra operations: one for sampling random variables and one for conditioning [33]. Here for conditioning we use a function `score`, which assigns a non-negative weight to the execution path of the program in which it was run. In practice we usually use `score` to provide explicitly the values of the likelihood, but it is also possible to use it to encode arbitrary constraints by giving the score of 1 when the condition is satisfied and 0 otherwise. We write all the probabilistic programs using Haskell syntax, but their meaning should not be too difficult to understand even for readers unfamiliar with Haskell. For example, the sprinkler model introduced above can be written as a probabilistic program in the following way.

```
rain      ← bernoulli 0.2
sprinkler ← bernoulli 0.1
let prob_lawn_wet = case (rain, sprinkler) of
      (True , True ) → 0.99
      (True , False) → 0.70
      (False, True ) → 0.90
      (False, False) → 0.01
score prob_lawn_wet --observe lawn wet
return rain
```

While probabilistic programming is a convenient tool for specifying standard probabilistic models, its real power shows when we need to incorporate existing code as a part of our model. For example, suppose we are in the realm of computer graphics and we want to generate a scene that satisfies certain conditions, but we already have access to a good generator that can produce various scenes that may or may not satisfy the condition. Using probabilistic programming, we can resuse this generator directly in a program that looks roughly as follows [76].

```
scene ← genereate_scene()
observe (condition(scene))
return scene
```

This is all very simple to define, but the problem remains how to compute the conditional distribution efficiently. It is sometimes postulated that a sufficiently smart compiler for a probabilistic programming language could synthesise a program that performs inference in the given model efficiently. While this can be achieved for some classes of models, in general the task is very difficult and arguably the speed of inference is the main barrier to widespread adoption of probabilistic programming systems.

1.3 Approximate Inference

Computing the posterior distribution for a probabilistic program requires integrating over all the possible execution paths of the program, which means computing a very complicated, multidimensional integral of arbitrary non-linear functions. It is very rare that such an integral would have a simple algebraic solution unless the model is specifically set up to make that happen, which places severe restrictions on the model structure.

Generally the inference problem is intractable, specifically it is #*P*-hard in the discrete case [77] and undecidable in the continuous case [2]. We therefore need to resort to approximation and even then specific inference algorithms only produce good results for certain classes of models. The need for efficient inference is the main design constraint for probabilistic programming languages and many such languages enforce restrictions on expressible programs to help with inference.

In this dissertation we focus on sampling-based algorithms for approximate inference. The idea is very simple. Instead of computing an integral over all the possible execution paths of the program, we will run it multiple times using different sources of randomness and average the results. This way we can estimate any statistics of the program outputs.

The only problem is how to sample from the conditional distribution defined by the probabilistic program. Sampling exactly from the posterior distribution is usually intractable so we need to resort to another approximation. Sampling-based approximate inference is an old and well-established research area. In this work we focus on two most popular approaches, namely importance sampling and Markov chain Monte Carlo (MCMC).

Sometimes probabilistic programming is presented as a tool that separates model specification from inference. A practitioner writes a probabilistic model that represents their prior knowledge and provides the observations, and a compiler automatically generates code that performs inference in this model and answers queries. While this is the intended experience for

most users of probabilistic programming languages (PPLs), someone still needs to write this compiler. Moreover, the hardness of the inference problem precludes the possibility of building a very smart compiler that can handle all possible probabilistic programs efficiently, so building such compilers is a never-ending process. The very fact that new inference algorithms are constantly being proposed, including in the probabilistic programming literature, demonstrates this need from a practical point of view.

Recognising development of backends for PPLs as a crucial task for the success of probabilistic programming, in this dissertation we develop methods for improving implementations of such backends. Specifically, we devise abstractions for modular implementation of inference algorithms, where simple building blocks can be combined to obtain sophisticated samplers. For example, the Sequential Monte Carlo (SMC) algorithm can be informally described as spawning N particles and then sequentially propagating them forward and resampling. In the library we develop in this dissertation SMC can be implemented literally by composition of those basic operations using the following code.

```
smc k n = finish .
  compose k (advance . hoistS resample) .
  hoistS (spawn n >>)
```

An algorithm called Resample-Move SMC is a modification of SMC that additionally includes Metropolis-Hastings (MH) transitions after each resampling. To implement it we simply add the MH steps into the code shown above.

```
rmsmc k n t = marginal . finish .
    compose k (advance . hoistS (
        compose t mhStep . hoistT resample)) .
    (hoistS . hoistT) (spawn n >>)
```

Note that this modification is obtained by inserting an existing building block `mhStep` into `smc` which leaves very little room for implementation bugs. In Chapter 7 we show even more elaborate algorithms constructed from these building blocks by simple composition.

We achieve the compositional implementation by breaking down the compiler for our PPL into a sequence of transformations between intermediate representations, as shown in Figure 1.2. Since compilers for normal programming languages are typically broken into phases in the same fashion, we can regard our approach as an application of a popular compiler implementation technique to PPLs. However, the way in which we construct intermediate representations and transformations between them is much less standard.

Each intermediate representation is itself composed of several layers that we call inference transformers, forming an inference stack. Individual inference transformations are associated

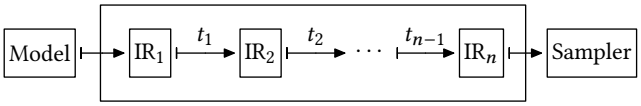


Figure 1.2: Conceptualisation of Bayesian inference. The model is written by the user in a probabilistic programming language and inference produces a sampler, which is a program implementing the selected inference algorithm for the specified model. Inference in most existing systems consists of a single conceptual step, while we can further decompose it into multiple passes through intermediate representations much like traditional compilers do. Crucially, all the intermediate transformations are exact and approximation is only made when executing the final sampler.

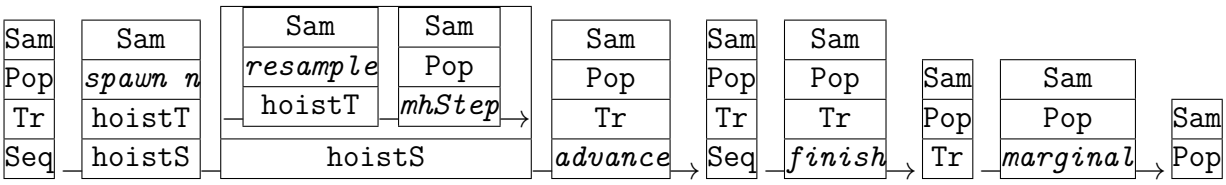


Figure 1.3: A graphical depiction of a sequence of transformations comprising the Resample-Move Sequential Monte Carlo algorithm implementation. Each inference transformation corresponds to a particular layer in the inference stack. To align with that layer it needs to be padded by a suitable inference stack on the top and a suitable `hoist` on the bottom.

with particular transformers in the stack. The `hoist` functions in the code are used to lift a transformation through a transformer. Figure 1.3 shows the particular representations and transformations used in the implementation of Resample-Move Sequential Monte Carlo.

In our setup all the inference transformations are exact in a sense that they do not introduce any approximation and the true posterior can still be recovered from any of the intermediate representations. All the approximation is contained in the final sampler. An alternative view would be that we only have one basic inference algorithm, importance sampling, and all the other building blocks form semantics-preserving program transformations, as advocated by Zinkov and Shan [94].

Our modular construction is implemented as a Haskell library for probabilistic programming called MonadBayes. It is freely available online[1].

[1] https://github.com/adscib/monad-bayes

1.4 Formal Semantics of Programming Languages

Programming languages are difficult to understand and their various features often combine in unexpected ways, which is a source of bugs in programs and difficulties in implementing compilers. To develop their better understanding programming languages are studied as formal languages using mathematical tools. The insights thus gained inform the design of languages and implementation of compilers.

Typically real programming languages are too complicated to formalise, so instead people study idealised calculi which encode the features in question without any of the clutter necessary to make a language practical. A calculus is a formal language for which a precise semantics is provided. There are two main benefits of such a construction. First, it communicates precisely the intended behaviour of programming constructs, unlike natural language descriptions which are often ambiguous in edge cases. Second, it allows certain properties of programs, such as correctness, to be proven rigorously, if only in an idealised setting. With those two objectives in mind we develop formal semantics corresponding to our compositional implementation for a suitably chosen formal calculus.

Unlike most authors developing formal semantics of probabilistic programming languages, we are interested in semantics corresponding to approximate inference algorithms that we actually execute, rather than idealised semantics corresponding to results of exact inference. This is necessary if we want to prove correctness of such algorithms in the context of probabilistic programming. While to a large extent it is possible to transfer proofs of these algorithms from the statistics literature, there are certain pitfalls to watch out for. For example, an influential paper of Wingate et al. [91] describes a generic technique for implementing a single-site Metropolis-Hastings algorithm for probabilistic programs. The paper was published with a serious error in the acceptance ratio formula, which was subsequently fixed, due to a behaviour exhibited by probabilistic programs but not by graphical models where this algorithm is usually applied. This example shows there is value in more rigorous analysis of inference algorithms applied to probabilistic programs.

In our semantic construction we prove correctness of a variant of the single-site MH, which has been done by other authors in other settings [9, 40]. Additionally we provide the first, to the best of our knowledge, rigorous proof of correctness of Sequential Monte Carlo applied to probabilistic programs. Our semantic construction is modular in precisely the same way as the implementation, so we automatically get proofs of correctness for all the compositions of these algorithms.

1.5 Outline of the Dissertation

Chapter 2 reviews background information for this work, in particular probabilistic programming, approximate Bayesian inference, functional programming, formal calculi and their semantics.

The next three chapters develop formal semantics corresponding to Monte Carlo inference algorithms. Chapter 3 introduces the formal calculus we use and constructs denotational semantics for it in a simplified case where only discrete distributions are allowed. The semantics correspond to naive enumeration and variable elimination algorithms and are used primarily for pedagogical reasons to demonstrate our approach to construction of inference algorithms in a simple setting. The calculus is a typed lambda calculus with function types and inductive types. It allows a form of primitive recursion but not full recursion so it only admits programs that always terminate. We use the same calculus in subsequent chapters, extending it with a primitive for sampling from continuous distributions.

Chapter 4 introduces the quasi-Borel spaces (QBS), which provide a basis for the semantics construction in the case involving continuous distributions. It shows that the QBS category has the properties needed to construct denotational semantics for our calculus and constructs a suitable probability monad. It further develops a synthetic measure theory for QBS which simplifies proofs of many theorems in the subsequent chapter. This chapter is not original work of the author of this dissertation but rather a review of work by other authors.

Chapter 5 extends our calculus with continuous distributions and constructs denotational semantics in QBS corresponding to popular Monte Carlo algorithms, in particular Sequential Monte Carlo (SMC) and Metropolis-Hastings (MH). The algorithms are constructed from simple building blocks using compositions that guarantee their correctness. This chapter constitutes the main theoretical development in this dissertation and its results are used as a basis for implementation in subsequent chapters.

Chapter 6 describes a Haskell library MonadBayes for probabilistic programming based on the semantic constructions presented in preceding chapters. Probabilistic programs in MonadBayes are ordinary Haskell programs using abstract probabilistic effects for sampling and scoring and a monadic interface. Inference algorithms are implemented using monad transformers and transformations between them, which constitute basic building blocks for compositional inference. We benchmark performance of MonadBayes against existing systems Anglican and WebPPL and find it competitive. We also show how a compositional approach enables deterministic testing of Monte Carlo algorithm implementations.

Chapter 7 demonstrates compositional implementations of several advanced Monte Carlo algorithms, namely Resample-Move SMC, Particle Marginal MH, and SMC^2. These are

obtained entirely by compositions of the building blocks defined in previous chapters, reflecting their informal descriptions.

Chapter 8 concludes and discusses directions for future work.

The work presented in this dissertation was published in two separate papers [80, 79] and is a result of collaboration with the remaining authors of these papers.

The original contributions to knowledge presented in this dissertation are the following:

- a novel theoretical framework for reasoning about correctness of Bayesian inference algorithms for probabilistic programs,
- a demonstration of the framework's power by means of providing multiple constructions of existing inference algorithms in it,
- a design of a Haskell library for probabilistic programming closely following the theoretical framework, thus obtaining unprecedented degree of modularity and a high degree of confidence in correctness,
- a novel method for testing implementations of randomized inference algorithms.

Chapter 2

Preliminaries

This chapter is an overview of basic concepts we use in this work. We discuss the relevant approximate inference algorithms in their general measure-theoretic formulation, selected concepts from denotational semantics including monads, and important design choices for the implementation of probabilistic programming languages.

2.1 Bayesian Inference Algorithms

As mentioned in Section 1.3, Bayesian inference is difficult and many different algorithms are used to perform it. In this section we briefly discuss the main classes of popular algorithms that we subsequently implement in this dissertation. A good general textbook on Bayesian inference is [60].

This section provides a classical treatment of the inference algorithms we use for probabilistic programs. Readers familiar with formal semantics of programming languages who find the derivations in this section difficult to follow can gloss over the equations since all the algorithms will be explained again in subsequent chapters in the context of probabilistic programs. Readers already familiar with these algorithms are nonetheless encouraged to read this section, since it provides a non-standard presentation that is easier to relate to the constructions in the subsequent chapters.

In Section 1.1 we have introduced Bayes' rule as the main equation of interest. However, the sprinkler model introduced there contains only discrete variables, which significantly simplifies the problem statement. We start this section by stating the problem of Bayesian inference in a way sufficiently general for probabilistic programming and then discuss inference algorithms used in this dissertation. Here we write X for the latent variable over which we want to do inference and Y for the observed variable. It is understood that X and Y can have arbitrary structure themselves, in particular comprising multiple discrete and continuous variables.

If X and Y are discrete we can state Bayes' rule in terms of probability mass functions that assign probabilities to individual values of the two variables

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} = \frac{p(y|x)p(x)}{\int_x p(y|x)p(x)}. \tag{2.1}$$

If X and Y are continuous Bayes' rule can be written exactly like above, but now p is a density rather than a mass function. These two cases can be unified by recognising that a probability mass function is in fact a density of the probability distribution with respect to the counting measure. Thus equation 2.1 states Bayes' rule using densities, with base measure determined from context. In the discrete case it is the counting measure, while in the continuous case it is the Lebesgue measure.

For more complicated models where X and Y involve mixtures of continuous and discrete distributions, possibly of varying dimensions, the posterior may not have a density with respect to any standard measure and constructing a measure for which all the densities in equation 2.1 exist is unnecessarily difficult. It is therefore easier to state Bayes' rule using measures directly rather than their densities. In subsequent chapters, when giving semantics to arbitrary probabilistic programs, we will use this statement of Bayes' rule as a starting point.

We can write the unnormalised posterior as a measure

$$\mu(A) = \int_A w(x)p(dx),$$

where p is the prior and $w(x)$ is the likelihood. The likelihood is usually written as $p(y|x)$, but here we use $w(x)$ instead, both to emphasise that the likelihood is a function of x and to account for probabilistic programs where the likelihood does not correspond to any density. In this dissertation we only deal with situations where the likelihood is available, not covering the likelihood-free cases. The normalised posterior is obtained by rescaling μ by a constant such that the measure of the whole space is 1. We often refer to the normalised posterior as the posterior distribution and to the unnormalised posterior as the posterior measure.

In our formulation we regard computing μ as Bayesian inference. Specifically, we are interested in integrals of measurable functions over this measure. This is slightly more general than the usual meaning of Bayesian inference, where the task is to compute expectation under the normalised posterior, since it also includes computing model evidence. Note that if we can integrate arbitrary functions over μ we can also compute expectations under the normalised posterior, since the normalisation constant is just an integral of a constant function 1 over μ.

2.1.1 Exact Inference

If we only have a finite number of random variables and each of them can only take a finite number of values then the inference problem can be solved by enumeration of all possible states of all variables, just like we did for the sprinkler model in Section 1.1. Unfortunately this is very expensive and can only be done for tiny problems. Realistically full enumeration is not a viable option for probabilistic programs, but nevertheless we find it useful for testing, as discussed in Section 6.4.

Sometimes it is possible to take advantage of the dependency structure between random variables to reduce the number of operations needed to do full enumeration. The variable elimination algorithm [93] provides a principled way of doing this. Even with that improvement doing inference by enumeration is prohibitively expensive, but we demonstrate a variant of variable elimination as a pedagogical device in Chapter 3.

If the variables can take values from an infinite set, whether discrete or continuous, exact inference is only possible if the relevant infinite sum or integral can be computed symbolically. In particular this is the case with exponential family distributions and conjugate priors [73] and many classical probabilistic models exploit these relationships for efficient inference. This approach is not sufficiently general to apply to probabilistic programs, but it is useful to perform symbolic exact inference where possible before falling back on approximate methods. In this dissertation we do not work with any symbolic computation or conjugate priors so we do not discuss these concepts any further.

There also exist algorithms that provide exact values for certain properties of the posterior, which can be regarded as exact inference even if they do not provide the full joint distribution. Examples include belief propagation [69], which provides exact posterior marginals. Since the applicability of those methods is also limited we do not consider them in this dissertation.

Another possibility is to draw samples from the posterior distribution and use them to form a Monte Carlo estimator to approximate the required expectations

$$E_{X\sim p}\left[f(X)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(X_i), \qquad \text{where} \qquad X_i \sim_{i.i.d.} p.$$

If the samples are drawn exactly from the posterior distribution, rather than from a different but similar distribution, we can still call this approach exact inference. A popular algorithm for drawing samples exactly from the posterior is called rejection sampling. However, in most cases it is not even feasible to draw samples exactly from the posterior so we need to resort to drawing samples from different distributions. In this dissertation we look at various sampling-based algorithms which approximate sampling from the true posterior. The algorithms we consider

fall into the so-called exact-approximate inference algorithm family, because they guarantee exact answers in a certain limit.

2.1.2 Markov Chain Monte Carlo

Markov chain Monte Carlo (MCMC) methods are possibly the most popular family of algorithms for approximate Bayesian inference [63]. Here we only provide a brief review of the relevant aspects of MCMC, referring interested readers to [11] for a more detailed survey.

The idea of MCMC, first proposed by Metropolis et al. [57] is to construct a Markov chain that converges to the posterior distribution, which in this context is also called the target distribution. A Markov chain is a sequence of random variables $X_1, X_2, \ldots$ such that each of them depends only on the previous variable and not on any variables that came before it. Technically we can write this condition as $p(X_{i+1}|X_1, \ldots, X_i) = p(X_{i+1}|X_i)$. Here we are only interested in stationary chains where the transition kernel K is the same at every time step, so the chain can be described as $X_{i+1} \sim K(X_i, \cdot)$.

Typically the transition kernel is set up to preserve the target distribution, that is if X_i is a sample from the target then so is X_{i+1}. In that case, and under certain technical conditions we will not get into here, the chain converges to the target distribution and satisfies a version of the Central Limit Theorem with variance decreasing as $O(n^{-\frac{1}{2}})$ asymptotically. Note that the normalisation constant of the measure obtained in the limit is the same as the normalisation constant for the measure from which X_1 was taken. Thus MCMC does not provide a direct way to estimate this constant but rather allows us to sample from the normalised posterior.

The Markov chain is fully determined by the initial distribution of X_1 and the transition kernel K. We now discuss some common ways to construct the transition kernel in a way that guarantees preservation of the target distribution. As argued by Geyer [28], this is the only property of the Markov chain used in MCMC that is feasible to guarantee generally and it is sufficient to satisfy our notion of correctness used in this dissertation. In practice a successful application of MCMC requires that the Markov chain converges sufficiently quickly, but this is usually determined empirically on a case-by-case basis.

A widely popular scheme for constructing transition kernels that preserve the target distribution is based on taking an arbitrary proposal kernel and combining it with an accept/reject step. The method was proposed by Metropolis et al. [57], then generalised by Hastings [37], and subsequently by Green [35]. Thus, following Geyer [28], we refer to it as the Metropolis-Hastings-Green (MHG) algorithm, even though this name is not very popular in the literature. Occasionally, particularly when referring to the work of others, we use the more established name Metropolis-Hastings (MH), but it should always be understood to mean its general version due to Green.

We now show a construction of a kernel K that preserves the target measure p based on an arbitrary Markov kernel Q using MHG. Q is a Markov kernel if $Q(x,\cdot)$ is a probability distribution for any x. A sufficient condition for a kernel K to preserve a measure p is for it to be reversible with respect to p, that is

$$\int\int f(x)g(y)p(dx)K(x,dy) = \int\int g(x)f(y)p(dx)K(x,dy)$$

for all measurable functions f and g. To construct such a kernel, we define measures m and n as

$$m(A) = \int\int 1_A(x,y)p(dx)Q(x,dy)$$
$$n(A) = \int\int 1_A(y,x)p(dx)Q(x,dy).$$

The acceptance ratio is defined as the Radon-Nikodym derivative of these measures $r = \frac{dn}{dm}$. If the relevant densities exist, it reduces to a more commonly used Metropolis-Hastings ratio $r = \frac{p(y)q(y,x)}{p(x)q(x,y)}$. However, for the Trace MH algorithm [91] that we implement in this work this is not the case so we need the more general version.

In the MHG algorithm the kernel $K(x,\cdot)$ is obtained by composing the kernel $Q(x,\cdot)$ with a step that accepts the new value y with probability $min(1, r(x,y))$ and otherwise keeps the old value x. This can be shown to result in a kernel reversible with respect to p, so K preserves p and we obtain a correct MCMC algorithm.

The question remains how to choose Q. We do not address this issue in this work, but rather provide a construction that can use any Q. Below we discuss some popular choices for Q.

If X consists of several random variables, we can take Q to update one of these variables using its conditional distribution p given the current values for the other variables. This algorithm is known as Gibbs sampling [26] and it is a special case of MHG where the acceptance ratio is always 1. This algorithm was used in an early probabilistic programming system BUGS [30] and later incorporated into many popular alternatives such as Infer.NET [58] and PyMC [68]. The main limitation of this approach is that the conditional distributions required can only be derived analytically in a limited number of cases, so Gibbs sampling is not applicable to general probabilistic programs and we do not use it in this dissertation.

For continuous variables a popular choice is to take Q that simulates Hamiltonian dynamics, leading to Hamiltonian (Hybrid) Monte Carlo (HMC) [20]. This algorithm, and in particular its black-box variant No-U-Turn Sampler (NUTS) [39] was popularised by the probabilistic programming language Stan [12]. Application of HMC in a probabilistic programming system requires the use of automatic differentiation, so we do not explore it in this dissertation for

reasons discussed in Chapter 8. We refer interested readers to [65] for a thorough review of HMC.

2.1.3 Importance Sampling

A popular generic alternative to MCMC for sampling from an intractable posterior is importance sampling. The idea is that instead of sampling from the target distribution p we sample from a different tractable distribution q and assign weights to the obtained samples in such a way that the weighted average targets p. We consider a more general case where we try to approximate integrals over a possibly unnormalised measure μ using samples from a probability measure q. If μ is absolutely continuous with respect to q, we can sample $X \sim q$ and set the weight to equal the Radon-Nikodym derivative $W = \frac{d\mu}{dq}(X)$, which in a less general setting can be written as a ratio of densities $\frac{p(X)}{q(X)}$.

It can be shown that for any measurable function f, $f(X)W$ is an unbiased estimator for the integral of f over μ. This follows directly from the definition of the Radon-Nikodym derivative

$$E_{X\sim q}\left[f(X)W\right] = \int f(x)\frac{d\mu}{dq}(x)q(dx) = \int f(x)\mu(dx).$$

The importance sampling estimator is unbiased, but it can have large variance. It is possible to take the weighted average of multiple samples, that is $\int f(x)\mu(dx) \approx \frac{1}{N}\sum_{i=1}^{N} f(X_i)W_i$, which reduces the variance proportionally to $\frac{1}{N}$. However, this can be computationally expensive so it is often necessary to make q a better proposal. In the absence of further information about f the best choice is to make q equal to μ up to a multiplicative constant. This is not possible in practice, since it would involve exact sampling from the posterior, but can be used as a goal for optimisation. In this dissertation we are not concerned with designing good proposal distributions so we will not elaborate on that further. In our constructions the proposal distribution will be simply the prior.

The problem with importance sampling is that often a small number of samples have very high weights, effectively making the other samples unimportant and reducing the effective sample size. This is particularly pronounced in sequential models, where multiplying likelihoods from many observations makes the weight differences even more extreme, since good trajectories tend to accumulate higher likelihoods at each time step.

This problem can be partially remedied by dynamically pruning samples with low weights early on in the sequential process. The particle filter algorithm [34, 17] performs this pruning in a carefully balanced way to ensure the resulting estimator remains unbiased. Specifically, let $x^{1:T}$ be a sequence of latent states and $y^{1:T}$ be a sequence of corresponding observations. We assume the state-space graphical model structure, that is $p(x^{t+1}|x^{1:t}) = p(x^{t+1}|x^t)$ and

$p(y^{t+1}|x^{1:t+1}, y^{1:t}) = p(y^{t+1}|x^{t+1})$ for all t. We are interested in the filtering distribution, that is $p(x^T|y^{1:T})$. We can sequentially approximate the filtering distribution up to time t, that is $p(x^t|y^{1:t})$. In a measure-theoretic notation we write μ^t for the corresponding unnormalised posterior measure.

In the particle filter at time step t we approximate μ^t as a collection of samples $\{(X_i^t, W_i^t)\}_{i=1}^N$ called particles in this context. This can be regarded as a discrete measure $\hat{\mu}^t(A) = \sum_{i=1}^N 1_A(X_i^t)W_i^t$. To obtain an approximation for time $t+1$ we would like to integrate the following unnormalised transition kernel

$$T(x^t, dx^{t+1}) = p(dx^{t+1}, y^{t+1}|x^t)$$

against $\hat{\mu}^t$. Thus ideally we would set $\hat{\mu}^{t+1}(A)$ to equal

$$\int\int 1_A(x^{t+1})T(x^t, dx^{t+1})\hat{\mu}^t(dx^t), \tag{2.2}$$

but the integral in question is intractable and instead we will approximate it by importance sampling using $p(x^{t+1}|x^t)$ as a proposal distribution.

The naive approach would be to observe that the integral over $\hat{\mu}^t$ is just a sum, resulting in a sum of integrals over $T(X_i^t, \cdot)$ and approximating each of these integrals by a single importance sample. This would correspond to choosing $X_i^{t+1} \sim p(X^{t+1}|X_i^t)$ and $W_i^{t+1} = W_i^t \frac{dp(X^{t+1}, y^{t+1}|X_i^t)}{dp(X^{t+1}|X_i^t)}(X_i^{t+1})$. Proceeding sequentially in this fashion is equivalent to using normal importance sampling on the unnormalized density $p(x^{1:T}, y^{1:T})$, taken as a function of $x^{1:T}$, using $p(x^{1:T})$ as a proposal distribution. This algorithm is sometimes called sequential importance sampling.

We can improve this algorithm by going back to equation 2.2 and instead of writing the outer integral as a sum doing importance sampling for both integrals simultaneously using the normalised version of $\int\int 1_A(x^{t+1})p(dx^{t+1}|x^t)\hat{\mu}^t(dx^{1:t})$ as a proposal density. For an equivalent computational budget we can now use N importance samples to approximate this integral. Operationally this corresponds to introducing a set of auxiliary "ancestor" variables A_i^t and using the following updates:

$$\begin{aligned} A_i^t &\sim Categorical(\{W_i^t\}) \\ X_i^{t+1} &\sim p(X^{t+1}|X_{A_i^t}^t) \\ W_i^{t+1} &= \frac{\sum_i W_i^t}{N} \frac{dp(X^{t+1}, y^{t+1}|X_{A_i^t}^t)}{dp(X^{t+1}|X_{A_i^t}^t)}(X_i^{t+1}). \end{aligned}$$

With this modification particles at intermediate stages survive with probability proportional to their weight, removing samples with low weights early on and ameliorating the problem of

low effective sample size. The estimator given by $\int f(x)\hat{\mu}^T(dx)$ is unbiased and typically has lower variance than the standard importance sampling estimator.

An alternative view of the particle filtering algorithm is as a sequential importance resampling. Specifically, we take the sequential importance sampling algorithm and include a resampling step after each time step. A resampling step replaces a collection of particles $\{(X_i^t, W_i^t)\}$ with another collection $\{(\tilde{X}_i^t, \tilde{W}_i^t)\}$ using the following recipe:

$$
\begin{aligned}
A_i^t &\sim Categorical(\{W_i^t\}) \\
\tilde{X}_i^t &= X_{A_i^t}^t \\
\tilde{W}_i^t &= \frac{1}{N}\sum_i W_i^t .
\end{aligned}
$$

This is completely equivalent to the formulation of particle filtering presented above, but it separates resampling and sequential importance sampling into two operations. We use this separation in our implementations subsequently in this dissertation. However, we find that it is the former construction that more clearly explains the utility of resampling. The reason resampling is useful is that it lets us allocate the computational budget more evenly in the subsequent transition. In the absence of such a transition resampling only serves to introduce noise. This is particularly important in the application of particle filters to probabilistic programs - if the program does not interleave conditioning with sampling a particle filter would perform worse than simple importance sampling. Finally, we presented above the simplest resampling scheme known as multinomial resampling, but other choices with better empirical performance are also possible [18].

The idea to apply particle filters to inference in probabilistic program was introduced by Wood et al. [92]. Following that work we refer to particle filters as Sequential Monte Carlo (SMC), which is a commonly used alternative name introduced by Liu and Chen [52]. For a more in-depth review of SMC we refer the reader to the survey by Doucet and Johansen [19].

SMC is sometimes combined with MCMC, either by incorporating MCMC transitions after resampling to increase diversity of the sample or by running SMC as a part of the MCMC transition kernel. We show in this dissertation that such compositions are naturally expressed in the framework we introduce. Specific examples are given in Chapter 7.

2.2 Types and Denotational Semantics

In this section we review the basic properties of the lambda calculus, a formal language commonly used as a model programming language. We discuss its syntax and denotational semantics to the extent it is relevant for the work presented in this dissertation. We exclude topics such as operational semantics and type inference since we do not make use of them in this work. This section is written in a tutorial style, targeted at readers with no previous experience in formal languages and semantics. At the end we also briefly discuss the Haskell language that we subsequently use for implementation.

2.2.1 Lambda Calculus Syntax and Semantics

Lambda calculus was originally introduced by Church [16] as a formal model of computation. Once it was established that typically programming languages can be reduced to a small calculus and a collection of derived forms [50] it became a popular vehicle for theoretical analysis of programming languages.

At the very core the lambda calculus is based on function definition and application. While that is sufficient to make the language Turing-complete, it is common to introduce additional primitives typically included in real programming languages. Below we present the syntax of a simple lambda calculus with product types, natural numbers, and addition.

t, s, r	$::=$		terms
		x	variable
	$\mid$	$\lambda x.t$	function abstraction
	$\mid$	$t\,s$	function application
	$\mid$	$()$	unit
	$\mid$	(t, s)	tuple creation
	$\mid$	**match** t	tuple inspection
		with $(x, y) \to s$	
	$\mid$	n	natural numbers
	$\mid$	$t + s$	addition

The lambda calculus provides a convenient notation for defining functions. For example, a function f defined by $f(x) = x + 1$ in normal notation would be written as $f = \lambda x.x + 1$. The lambda denotes a function definition, x is a variable name given to the argument, and the dot preceeds the function body. Function application is written using juxtaposition, so we can write fx instead of $f(x)$. Computation in the lambda calculus proceeds by reducing terms, for example $(\lambda x.x + 1)2$ reduces to $2 + 1$ and subsequently to 3.

One problem with the calculus presented above is that some syntactically correct programs are nonsensical. For example, the following expression attempts to increment a function

$$(\lambda x.x+1)(\lambda x.x).$$

Proceeding in the so-called Church style, we do not wish to give semantics to such programs and in practice programs like the one above simply crash. In order to exclude these programs from theoretical considerations, and to report them as incorrect before they are even run, we can introduce a type system. A type system can be viewed as a conservative approximation to the actual execution of the program which determines whether type errors would occur during execution. It is conservative in the sense that a program typechecks if type errors are guaranteed not to occur, but it may fail to typecheck even if no type error would actually occur.

To define a type system we first define the types themselves. The grammar of types mirrors the grammar of terms. Below we show types and typing rules for the grammar of terms introduced above, obtaining a variant of a simply-typed lambda calculus.

τ, σ, ρ	$::=$	types	
		$\mathbb{N}$	natural numbers
	$\mid$	$\tau \to \sigma$	function
	$\mid$	$\mathbf{1}$	unit
	$\mid$	$\tau * \sigma$	finite product

Typing rules are usually given by induction on program terms, with a term having a certain type provided that its subterms have certain types.

$$\frac{}{\Gamma \vdash () : \mathbf{1}} \qquad \frac{}{\Gamma \vdash n : \mathbb{N}} \qquad \frac{\Gamma \vdash t : \mathbb{N} \qquad \Gamma \vdash s : \mathbb{N}}{\Gamma \vdash t + s : \mathbb{N}}$$

$$\frac{}{\Gamma \vdash x : \tau}((x : \tau) \in \Gamma) \qquad \frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \lambda x : \tau . t : \tau \to \sigma} \qquad \frac{\Gamma \vdash t : \sigma \to \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash t\, s : \tau}$$

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash (t, s) : \tau * \sigma} \qquad \frac{\Gamma \vdash t : \sigma * \rho \qquad \Gamma, x : \sigma, y : \rho \vdash s : \tau}{\Gamma \vdash \mathbf{match}\, t\, \mathbf{with}\, (x, y) \to s : \tau}$$

In this notation each typing rule is associated with a horizontal bar. Above the bar are preconditions for the rule, which can be empty, and below is the typing judgement that follows. The typing relation is denoted using $\vdash$, which is followed by a term, then a colon and the term's type. For example, the three rules on the first line say that unit is of type unit, a number is of type number, and if two terms are of number types then their sum is also of number type.

On the left of $\vdash$ is the typing context. The typing context is a partial function from variable names to types, which holds the types of all variables currently in scope. The rules on the

second line show how it is used to type function terms. A variable can be typed only if it is found in the context, ensuring that in well-typed programs all variables are bound somewhere. A lambda term defining a function can be typed only if the function body can be typed in the current context extended with a type for the variable representing the function argument. Note that the type of the argument is given explicitly in syntax to make typing derivations easier, although in real languages it can often be inferred automatically. Finally, the third line gives typing rules for products, again showing how context is used to type expressions with pattern matching. The full program should be typeable in the empty context to be considered well-typed.

Recall that earlier we considered the following nonsensical but syntactically correct program, now presented with type annotations

$$(\lambda x : \mathbb{N}.x + 1)(\lambda x : \mathbb{N}.x).$$

It can be verified that this program does not typecheck using the type system given above. The second term would need to have the type $\mathbb{N}$ but no typing rule can give this type to a lambda term.

In statically typed languages an automatic type checker is run before the compiler to ensure that the program is type-safe. The type systems are set up carefully to enable this process to be done automatically, but the details of that are not important for this dissertation.

We have defined the syntax and the type system for our calculus, but we still do not know what the programs actually mean. It is customary at this point in textbooks to define the so-called operational semantics, which specify how the programs reduce to simpler forms emulating actual execution. These semantics can be used to prove properties of running programs, such as that well-typed programs never get stuck. However, since we do not introduce any operational semantics in this dissertation we are not going to discuss these here.

Another popular approach is to define denotational semantics [81], which associates well-typed program terms with mathematical objects such as numbers and functions. Such semantics is compositional in a sense that the meaning of an expression only depends on the meaning of its subexpressions and it is therefore particularly suitable for reasoning about equivalences between programs. Traditionally it is often used to prove correctness of transformations performed by a compiler, while in this dissertation we use it to show that transformations associated with inference building blocks are correct.

We start the construction by associating types with certain mathematical spaces. In the simple case of the calculus presented above those spaces will be just sets. These sets are constructed by induction on the structure of types. It is customary to use double brackets $[\![\cdot]\!]$ for denotational semantics.

$$[\![\mathbb{N}]\!] \coloneqq \mathbb{N} \qquad [\![\mathbf{1}]\!] \coloneqq \mathbb{1} \qquad [\![\tau * \sigma]\!] \coloneqq ([\![\tau]\!]) \times ([\![\sigma]\!]) \qquad [\![\tau \to \sigma]\!] \coloneqq ([\![\sigma]\!])^{[\![\tau]\!]}$$

We associate the set of natural numbers with the type of natural numbers and a one-element set with the unit type. For product types we use Cartesian products and for function types we use functions, written in the exponential notation by convention.

We then associate well-typed terms with elements of the sets corresponding to the type, again proceeding by induction on term structure. Specifically, we define the semantic relation, again denoted by $[\![\cdot]\!]$, such that for any well-typed closed term $\vdash t : \tau$, we have $[\![t]\!] \in [\![\tau]\!]$. The semantics is constructed by induction on term structure. Just as we needed a context for the typing rules to keep track of types of variables, here we need a context ρ to keep track of values of variables. For the simple calculus presented here the definition of semantics amounts mostly to pushing the semantics brackets inside the expression, so we only present a few selected rules.

$$[\![n]\!](\rho) \coloneqq n \quad [\![t+s]\!](\rho) \coloneqq [\![t]\!](\rho) + [\![t]\!](\rho) \quad [\![(t,s)]\!](\rho) \coloneqq ([\![t]\!](\rho), [\![t]\!](\rho))$$
$$[\![x]\!](\rho) \coloneqq \rho(x) \quad [\![\lambda x : \tau.t]\!](\rho) \coloneqq \lambda y.\, [\![t]\!](\rho[x \to y])$$

The semantics make precise the relationship between programs and the mathematical objects we use to reason about what programs do. In the case presented here the correspondence is trivial, but it becomes much more involved once we start adding new constructs to the calculus. Typically most of the rules in the semantic construction is standard, so only the interesting ones are presented to avoid clutter.

Equipped with denotational semantics we can prove certain program equivalences. For example, the following two programs are equivalent in the calculus presented so far, which can be shown by computing their semantics

$$\lambda f : \mathbb{N} \to \mathbb{N}.f(0) + f(1) = \lambda f : \mathbb{N} \to \mathbb{N}.f(1) + f(0).$$

However, these two programs may not be equivalent once the calculus is extended, as we show below.

2.2.2 Effectful Computation and Monads

For the simple calculus above we were able to define semantics in terms of simple sets. However, more generally we need spaces with additional structure, such as when the calculus includes

computation with side-effects. Note that side-effects are basically unavoidable, since all useful programs need to at least produce some output.

A common side effect of computation involves updating values of mutable variables. It is customary to include mutable variables in the lambda calculus in a form of references to mutable cells with the syntax shown below.

t,s,r	$::=$	terms	
		$\ldots$	as before
	$\mid$	**ref** t	allocate a mutable cell
	$\mid$	$t := s$	assign to a mutable cell
	$\mid$	$!t$	dereference a mutable cell
	$\mid$	$t;s$	sequence operations

A reference to a mutable cell holding values of type τ is itself given a type τ **ref**. On top of constructs for working with mutable cells we add a sequencing operations, so that we can return a result after updating a cell.

With this extension the equivalence shown at the end of the previous subsection no longer holds. To see that, consider the function

$$f = (\lambda r : \mathbb{N}\ \textbf{ref}.(\lambda x : \mathbb{N}.r := !r + x; !r))(\textbf{ref}\ 0).$$

Now $f(0) + f(1)$ evaluates to 1 while $f(1) + f(0)$ evaluates to 2, assuming left-to-right evaluation order. Once we have introduced computation with global effects into the calculus, the order of evaluation becomes important. To reason about equivalences of programs using mutable cells we need to somehow account for the state of those cells in the semantics. The standard solution is to associate each type with a function that additionally updates a global state.

Let G be a set of possible global states, with a state being a partial functions from locations in a set L to values - the details of this construction are not important and we skim over the fact that it would require restrictions on the types for which references can be constructed to avoid divergence. We associate types with functions that produce a value and a new state based on the current state. For example, $[\![\mathbb{N}]\!] := G \to \mathbb{N} \times G$ and $[\![\tau\ \textbf{ref}]\!] := G \to L \times G$.

Recall that for the basic calculus giving semantics to terms was trivial. Now with the extension to reference cells defining the semantics requires carefully feeding the state through, which is cumbersome and repetitive. For example,the semantics of addition would be

$$[\![t+s]\!](\rho)(g) := \textbf{let}\,(x,g') = [\![t]\!](\rho)(g)\,\textbf{in}\ \textbf{let}\,(y,g'') = [\![s]\!](\rho)(g')\,\textbf{in}\,(x+y,g'').$$

We do not show the other rules here, but in almost all them we would find the pattern of feeding through the updates to the global state. In the simple calculus we have shown doing this manually is manageable, but the effort would quickly get too much once we started adding more constructs and other effects. Worse yet, performing any kind of reasoning about the program would be nearly impossible due to all the clutter in the semantics. Fortunately, the pattern in question is very repetitive so we can hope to use additional abstractions to hide it.

At this point we step back a bit and consider more generally the problem of defining denotational semantics in a reusable fashion. In the construction presented above we map types to sets, but we might want to use spaces with more structure instead. Those could be measurable spaces for probabilistic programs or domains if the calculus allowed for non-terminating computation. Despite this change, the semantics of the constructs from the basic calculus would be essentially the same. We would therefore like to define their semantics more abstractly, assuming only that the collection of spaces in question has certain properties. Conveniently such a framework is provided by category theory.

Categories are defined in terms of objects and morphisms which generalise sets and functions respectively. Semantics of specific language constructs are then defined in terms of abstract properties of a category. For example, to accommodate product types the category needs to have products that satisfy certain axioms, and for function types it needs exponentials. A category with these and several more properties can be used to give semantics to a simple lambda calculus and is called a Cartesian-closed category. To recover the construction presented above we simply choose the *Set* category where objects are sets and morphisms are functions between them.

To give semantics to programs with side effects, such as mutable reference cells, we need a categorical structure called a (strong) monad [59]. For example, the construction above is the state monad for the global store in the *Set* category. More generally a monad defines two operations: how to embed pure computation in a monad and how to chain computations in the monad where one computation uses the result of the other. These operations are traditionally called return and $>\!\!>\!\!=$ (pronounced 'bind') and for a monad M they have the types

$$\text{return} : X \to MX \qquad >\!\!>\!\!= : MX \to (X \to MY) \to MY.$$

Going back to the concrete case of the calculus with reference cells, we can define a monad S such that $SX \coloneqq G \to X \times G$ and the required operations are defined as

$$\begin{aligned}\mathrm{return}(x)(g) &:= (x,g)\\ (t \gg\!= f)(g) &:= \mathbf{let}\,(x,g') = t(g)\,\mathbf{in}\,f(x)(g').\end{aligned}$$

This construction captures precisely the pattern of feeding through the global state as the computation proceeds. We can use the monadic interface to provide an equivalent definition of semantics for addition in our calculus, that is

$$[\![t+s]\!](\rho)(g) := [\![t]\!](\rho) \gg\!= \lambda x.\left([\![s]\!](\rho) \gg\!= \lambda y.\,\mathrm{return}(x+y)\right).$$

The advantage of this definition is that it is stated in terms of an abstract monad, so it can be ported to a different setting without any change. For example, we could construct a probability monad of measurable spaces and use exactly this definition for semantics of addition in a probabilistic program.

Readers unfamiliar with denotational semantics may be overwhelmed with various constructions introduced in this section. Our goal is not so much to teach the techniques for constructing semantics as it is to provide the readers with some understanding of why our developments in this dissertation make use of so many esoteric concepts. In essence, this is because constructing the semantics directly would be very complicated, likely to the point where neither the authors nor the readers would have a good grasp of what is going on. Thus in the subsequent chapters we take advantage of the standard abstractions established in the programming languages community to separate our construction into different levels that to a large extent can be appreciated separately.

Generally to construct denotational semantics for a calculus with effects we need to do the following:

1. construct a suitable category in which the programs can be interpreted,

2. show that the category has the required properties to interpret the standard constructs used in the calculus,

3. construct a suitable monad to interpret the effects.

This is precisely the approach we take in Chapter 4 to define the semantics for our probabilistic calculus. We do not review any category theory in this chapter, recognising that readers familiar with it do not need this review and readers who are not are unlikely to find such a review sufficient. All the category theoretic content in this dissertation is isolated to Chapter 4

and can be skipped without significantly affecting the understanding of the other parts of the dissertation.

As a final remark on monads, observe that we can take the definition of the semantics for addition above and introduce some line breaks to arrive at the following form:

$$\begin{aligned}
&[\![t+s]\!](\rho)(g) := \\
&\qquad [\![t]\!](\rho) >\!\!>\!= \lambda x. \\
&\qquad [\![s]\!](\rho) >\!\!>\!= \lambda y. \\
&\qquad \mathrm{return}(x+y).
\end{aligned}$$

We can now apply a line-by-line transformation to introduce a popular syntactic sugar for monads known as the do syntax

$$\begin{aligned}
&[\![t+s]\!](\rho)(g) := \mathbf{do} \\
&\qquad x \leftarrow [\![t]\!](\rho) \\
&\qquad y \leftarrow [\![s]\!](\rho) \\
&\qquad \mathrm{return}(x+y).
\end{aligned}$$

We make extensive use of this notation, both in the calculus and in the implementation.

2.2.3 Type System Extensions

The basic calculus presented in Section 2.2.1 is fairly conservative in terms of the types it allows. In this section we introduce two extensions to the type system which make the calculus more realistic as a model programming language.

The first extension is parametric polymorphism [85]. The idea is that some functions are to an extent agnostic to the type of argument they receive so they should be able to receive arguments of multiple types. For example, the identity function $\lambda x.x$ can safely accept arguments of any type. However, in the calculus presented above we are forced to pick a particular type for it. It would not be legal to apply the same function to both a number and a unit.

A well-known extension of the simple type system presented above introduces type variables $\alpha, \beta, \ldots$ which allows terms to have multiple different types simultaneously. For example, the identity function would be given the type $\alpha \to \alpha$, where α is implicitly universally quantified

over. Such an extension is known as parametric polymorphism, since it treats all the types uniformly, in contrast to ad-hoc polymorphism where the behaviour of the function can be specified independently for each input type.

Since type variables complicate the type system significantly, it is common to place restrictions on where polymorphic values can used. In the basic case they are only allowed in a let binding, which in particular prohibits polymorphic values being passed as arguments to functions. Relaxing this assumption leads to higher-rank polymorphism.

In real languages it is very important for the users to be able to define their own types. Often it is convenient to define such a type polymorphically, for example to have a single definition for a list regardless of the type of the element in the list. In that case instead of defining a type the user defines a type constructor, which can be applied to a type to yield another type. Notationally List would be a type constructor, $\mathsf{List}\,\mathbb{N}$ would be a monomorphic type, and $\mathsf{List}\,\alpha$ would be a polymorphic type.

If the names of type constructors are not syntactically distinguished from the names of types it becomes possible to construct types that are syntactically correct but nonsensical. An example would be $\mathsf{List}\,\mathsf{List}$, not to be confused with $\mathsf{List}(\mathsf{List}\,\alpha)$ which is a list of lists. To statically prevent that we need a type system for the type system, which is called a kind system. Usually types are given kind $*$, while type constructors are given kind $* \to *$. Allowing type variables to have kinds other than $*$ is known as higher-order polymorphism.

All types of polymorphism serve the purpose of simplifying the programs by avoiding code repetitions. This is important in practice and our implementation makes crucial use of both higher-rank and higher-order polymorphism to make its code more concise and modular. However, in a calculus we can exclude the polymorphism and pretend that we have defined a family of functions, one for each monomorphic type that we use them for, in order to simplify the presentation. We take this approach in this dissertation and formally avoid all types of polymorphism in the calculus.

Independently of polymorphism an important extension of the simple type system is that of recursive types. These are types defined by reference to themselves and the list type discussed above is a prime example of that. Typically a definition of the list type would be

$$\mathsf{List}\,\alpha := \{\mathsf{Nil} \mid \mathsf{Cons}\,(\alpha * \mathsf{List}\,\alpha)\}.$$

The definition says that a list is either empty or an element and another list. Typically such a recursive definition is written using a type variable and a special symbol μ

$$\mathsf{List}\,\alpha := \mu\beta.\{\mathsf{Nil} \mid \mathsf{Cons}\,(\alpha * \beta)\}.$$

Without such an extension to the type system it would not be possible to construct lists of length unknown statically as a user-defined type. In our calculus we allow a weaker form of recursive types known as inductive types. Intuitively the difference is that in inductive types the type variable can not appear on the left of the function arrow, which is not a problem for lists. We use inductive types because their semantic treatment is easier but they are sufficiently expressive to include all the recursive types we use. The details of our use of inductive types are given in Chapter 3.

2.2.4 Haskell

As this dissertation features a Haskell library for probabilistic programming, it includes a lot of Haskell code. Since Haskell is a pure functional programming language, to a large extent it looks like the lambda calculus presented above. However, there are a few differences to keep in mind. First of all, in Haskell a type signature follows the double colon `::` while the single colon `:` is the cons for a list, the other way around from the lambda calculus. The type signature is usually provided on a separate line preceding the actual definition. Haskell code does not include Greek letters so type variables are written as lowercase Latin letters. Furthermore, at type level concrete types and type constructors are uppercase while type variables are lowercase. At the expression level constructors are uppercase and variables are lowercase. The lambda is replaced with a backslash and in a lambda expression the dot is replaced with $\rightarrow$. On top of that, Haskell has a notion of a typeclass, which is similar to an interface in many other languages. Below is the definition of the `Monad` typeclass corresponding to the monadic interface discussed above.

```
class Monad m where
  return :: a → m a
  >>= :: m a → (a → m b) → m b
```

Monads are ubiquitous in Haskell and they have their own syntactic sugar in a form of the `do` expressions. Most monads are implemented as monad transformers to enable compositions with other monads. For example, the state monad transformer extends a monad with the global state, forming another monad. The following snippet defines the state monad transformer type constructor `StateT` and implements a `Monad` instance for it using the monadic operations, including the `do` syntax, for the transformed monad.

```
newtype StateT s m a = StateT (s → m (a, s))
instance Monad m ⇒ Monad (StateT s m) where
  return x = StateT (\s → return (x, s))
  (StateT c) >>= f =
    StateT (\s → do
                    (x, s') ← c s
                    let (StateT d) = f x
                    (y, s'') ← d s'
                    return (y, s'')
            )
```

The symbol ⇒ denotes a type constraint. The instance reads that `StateT s m` is a `Monad` if `m` is a `Monad`.

Our overview of Haskell is necessarily brief and it is not feasible to review here all the constructs used in the listings in subsequent chapters. The Haskell 2010 Language Report [55] is the authoritative reference on the Haskell language that can be consulted to understand the meaning of program snippets presented in this dissertation. Finally, Haskell is famous for being lazy, or to be precise non-strict. This feature is not important for any code presented in this dissertation and the reader is safe to ignore it.

2.3 Probabilistic Programming

As explained in the introduction, in probabilistic programming we devise computer languages for defining probabilistic models, such that inference algorithms can be applied to them automatically. The field is vast with multiple different systems available and we can not review them all. However, in this section we discuss some important design choices and explain which ones we make in our formal calculus and implementation. We also include a brief survey of the existing literature on semantics of probabilistic programs.

2.3.1 Discrete and Continuous Random Variables

The first choice is what distributions can be sampled from in the program. Typically a PPL provides a collection of parameteric primitive distributions that the user can sample from. These distributions can be divided into discrete, such as Bernoulli, categorical, and Poisson, and continuous, such as normal, gamma, and beta. Some PPLs, especially ones not targeting machine learning or statistics applications, only allow discrete distributions. The advantages of this approach are that semantics are much simpler and that it is possible to perform inference by enumerating different execution paths of the program. Systems that take this approach include

IBAL [70], Hansei [46], and the work on probability monads [74, 22]. In this dissertation we restrict ourselves to discrete variables in Chapter 3 for pedagogical reasons, but the formal calculus in Chapter 5 and the library in Chapter 6 both allow continuous distributions as well.

Conversely some PPLs only allow continuous distributions, a prominent example being Stan [12]. Even though defining semantics is more challenging in the continuous case, performing inference can be easier. While technically the continuous case is strictly more challenging [2], in practice often inference in continuous models can be done well using gradient methods such as Hamiltonian Monte Carlo [20, 64], which are not available in the discrete case. In this dissertation we do not consider the restriction to continuous distributions.

2.3.2 Conditioning

Another choice in the design of the PPL is what conditioning operations are allowed. Conceptually the simplest choice is to condition using a predicate supplied by the user. Specifically the user inserts guards with Boolean conditions in certain places in the program. If the condition is true then the particular execution trace is retained, otherwise it is discarded. The total probability is then renormalised over all retained traces. For example, the sprinkler model written using this style of conditioning can be written like this.

```
rain      ← bernoulli 0.2
sprinkler ← bernoulli 0.1
let prob_lawn_wet = case (rain, sprinkler) of
      (True , True ) → 0.99
      (True , False) → 0.70
      (False, True ) → 0.90
      (False, False) → 0.01
lawn_wet ← bernoulli prob_lawn_wet
observe (lawn_wet == True)
return rain
```

While simple to understand and convenient to use, this style of conditioning makes it challenging to do inference, since finding program traces that satisfy an arbitrary predicate is not easy [31]. It is possible to some extent to propagate the constraints backwards through the program code [66, 87], but the required program analysis is involved to the point of being difficult to apply to feature-rich languages. In practice inference in models with such hard constraints is usually done using likelihood-free methods, such as rejection sampling or Approximate Bayesian Computation.

In PPLs with continuous random variables a special case of this problem arises when the constraint is only satisfied on the measure zero subset of the execution traces, but nonetheless

one that is contained within the support of the prior. An example of that is conditioning on a particular outcome of a normal distribution.

```
mean ← normal 0 1
obs ← normal mean noise
observe (obs == 0.34)
return mean
```

In this case we can not apply the usual formula for conditional probability, since both the numerator and denominator are zero. However, the model still has a sensible meaning which can be made precise through disintegration [14]. To some extent the disintegration transformation can be applied in probabilistic programs [82], but the problems faced are similar to the ones encountered with propagating constraints up the program.

A common way to avoid these problems is to require that the users explicitly supply the likelihood resulting from the conditioning operation. An example of that is the sprinkler model given in Section 1.2, which is equivalent to the formulation shown above. This style of writing probabilistic programs essentially shifts the burden of performing the required transformations of constraints onto the user. It is a common approach of many probabilistic programming systems, which often additionally provide syntactic sugar that allows observations of random variables just after they were sampled, in which case the likelihood is simply the density of the primitive distribution used. We take this approach in this dissertation, both in the formal calculus and in the implementation.

2.3.3 Domain-Specific Languages

Regardless of which distributions are available and which conditioning operations are supported, there remains a choice of how to implement a PPL. One possibility is to devise a domain-specific language (DSL), that is a special language designed only for specifying probabilistic models. Such a DSL can be completely separate from any programming language, having its own syntax, compiler, IDEs, debuggers, and file formats. The advantage of this approach is that such a PPL is portable and easy to teach. It is also simpler to do sophisticated program analysis on a DSL. On the other hand providing all the developer tools requires a tremendous amount of effort and even then the PPL is usually feature poor compared with normal programming languages. Furthermore, incorporating a probabilistic model into a larger application forces the users to work in multiple programming languages simultaneously, sometimes resorting to file-based communication, which is tedious and error-prone. Examples of stand-alone PPLs include BUGS [30], Stan [12], LibBi [61] and Hakaru [62].

An alternative to a stand-alone implementation is to embed a DSL into an existing programming language. The host language can then be used to put together expressions in the PPL, but

the final model only includes constructs from the DSL and not arbitrary host language code. The advantages of embedding a DSL mirror those of stand-alone DSLs. It takes relatively little effort to implement, allows reuse of all the developer tools from the host language, and makes it easy to incorporate a probabilistic model as a part of a larger application. On the other hand an embedded DSL is tied to a particular host language and can be difficult to learn for people unfamiliar with the host language. Both embedded and stand-alone DSLs make it relatively simple to perform sophisticated analysis of the model structure and to enforce restrictions on what types of models are allowed. On the other hand they tend to be less flexible than PPLs described in the next section. It is also difficult to incorporate existing code into a model written in a DSL. Examples of PPLs implemented as embedded DSLs include Infer.NET [58], PyMC [68], and Edward [88].

Finally, although this does not have to be the case, probabilistic programming DSLs often enforce a constraint that the model only has a finite number of latent variables that is known before the model is run. This restriction makes inference significantly easier, since it is easy to maintain the identity of different latent variables across different execution traces. However, it limits what kinds of models can be expressed, in particular excluding Bayesian nonparametric models.

2.3.4 Extensions of General-Purpose Programming Languages

Another possibility for implementing a PPL is to extend an existing programming language with operations for sampling and conditioning. This can be done through a generic mechanism for extending the language with computational effects or in an ad-hoc fashion. Such an extension provides a very flexible PPL that can easily incorporate any existing code from the host language and use its full power within the model specification. Additionally the implementation is often relatively concise. The main disadvantage is that it can be difficult to devise good inference algorithms that can handle all the possible ways in which a model can be written, in particular performing program analysis in such a language is difficult. The flexibility of the host language usually means that the model can include a varying and unbounded number of random variables and it can be difficult to align them across different execution traces. On top of that it forces the users to use a particular programming language, even if they are unfamiliar with it. Examples of PPLs implemented as extensions to existing languages include Pyro, Turing [25], and Hansei [46].

Finally, some PPLs are difficult to classify as either DSLs or extensions of an existing language. One family takes a subset of an existing language, extend it with probabilistic effects, and write inference algorithms by compilation to the original language. Sometimes they also provide an interface to that language, allowing existing code to be included into the model

under specific conditions and incorporating a model as a part of an application in the existing language. Examples of such PPLs include Anglican [86], WebPPL [32], and R2 [66]. Another PPL difficult to classify is Figaro [71] which looks like a DSL embedded in Scala but can incorporate arbitrary Scala functions.

Our library MonadBayes is developed as an extension of Haskell using monads as a mechanism to introduce custom computational effects. It is similar in scope to other languages described in this subsection, so we only compare its performance to these languages, in particular to Anglican and WebPPL.

2.3.5 Semantics for Probabilistic Programs

Formal semantic treatment of probabilistic programs dates back at least to the work of Kozen [49] who gave semantics to probabilistic while-programs using partial measurable functions and continuous linear operators. However, the early work on probabilistic programming semantics did not include any constructs for specifying observations. For our purposes the most relevant papers are that of Jones and Plotkin [44] and Ramsey and Pfeffer [74] who describe several different probability monads.

Formal semantics for probabilistic programs including conditioning and targeting specifically machine learning applications only started to appear later, with particularly influential papers by Park et al. [67] and Borgström et al. [10]. Since then, many authors have considered semantics of probabilistic programs with conditioning, but most of these papers consider idealised semantics in terms of measures and other intractable mathematical constructs. Such semantics is well-suited to reasoning about program equivalence, but it in general does not correspond to approximate inference algorithms actually being executed.

Semantic treatment of MCMC-based inference is scarce, with notable exceptions of Hur et al. [40] and Borgström et al. [9] who prove correctness of MCMC samplers for imperative and functional languages respectively. For inference algorithms based on particle filters, van de Meent et al. [89] provide a formal specification of the Particle Gibbs with Ancestor Sampling algorithm but do not attempt to prove its correctness. To the best of our knowledge a semantic construction including a proof of correctness of particle filtering algorithms was missing from the literature before the publication of papers containing the work presented in this dissertation. Independently of our work, Zinkov and Shan [94] developed a framework for reasoning about correctness of inference algorithms using program transformations but they do not consider particle filtering algorithms.

Chapter 3

Formal Calculus and Discrete Inference Semantics

In this chapter we introduce the formal probabilistic calculus for which we subsequently give formal semantics. The calculus uses two standard probabilistic primitives, one for sampling from a given probability distribution and one for scoring a trace of the program. In this chapter we only allow discrete distributions, which makes the semantics much simpler. We use this setting to introduce the structures we use to construct inference algorithms in a setting where the main ideas of the construction are not obscured by the details of quasi-Borel spaces. The developments in this chapter set the stage for the full semantic construction given in the subsequent two chapters.

We use a variant of the simply-typed λ-calculus with sums and inductive types, base types and constructors, primitives, and primitive recursion, but without effects. We also use monad-like constructs in the spirit of Moggi's computational λ-calculus [59]. The core calculus is very simple, and at places we need an inherently semantic treatment, which the core calculus alone cannot express. In those cases, we resort directly to the semantic structures, sets or spaces. Generally, however, we use the calculus as much as possible to take advantage of the fact that the types and functions expressed in it are by construction well-formed objects and morphisms respectively. In the continuous case, using this calculus yields correct-by-construction quasi-Borel spaces and their morphisms, avoiding a tedious and error-prone manual verification. Using the core calculus also brings our theoretical development closer to potential implementations in functional languages.

While the calculus contains many popular programming language features, it only allows a form of primitive recursion but not general recursion. This means that every well-typed program is guaranteed to terminate so the semantics does not need to deal with diverging computation. While we believe our constructions can be extended to languages with general

$$
\begin{array}{lll}
\tau, \sigma, \rho & ::= & \text{types} \\
& \alpha & \text{positive variable} \\
& \mid \{\ell_1\, \tau_1 \mid \ldots \mid \ell_n\, \tau_n\} & \text{variant} \\
& \mid \mathbf{1} \mid \tau * \sigma & \text{finite product} \\
& \mid \mu\alpha.\tau & \text{inductive type}
\end{array}
\qquad
\begin{array}{ll}
\mid \tau \to \sigma & \text{function} \\
\mid \mathsf{A} & \text{base} \\
\mid \mathsf{F}\,\tau & \text{base constructors} \\
\Gamma ::= x_1 : \tau_1, \ldots, x_n : \tau_n & \text{variable contexts}
\end{array}
$$

$$
\begin{array}{lll}
t, s, r & ::= & \text{terms} \\
& x & \text{variable} \\
& \mid \tau.\ell\, t & \text{variant constructor} \\
& \mid () \mid (t, s) & \text{nullary and binary tuples} \\
& \mid \tau.\mathbf{roll} & \text{iso-inductive constructor} \\
& \mid \lambda x : \tau.t & \text{function abstraction} \\
& \mid \mathbf{match}\, t & \text{pattern matching: variants} \\
& \quad \mathbf{with}\, \{\ell_1\, x_1 \to s_1 \mid \cdots \mid \ell_n\, x_n \to s_n\} &
\end{array}
$$

$$
\begin{array}{ll}
\mid \mathbf{match}\, t & \text{binary products} \\
\quad \mathbf{with}\, (x, y) \to s & \\
\mid \mathbf{match}\, t & \text{inductive types} \\
\quad \mathbf{with}\, \mathbf{roll}\, x \to s & \\
\mid \tau.\mathbf{fold}\, t & \text{inductive recursion} \\
\mid t\, s & \text{function application} \\
\mid \varphi & \text{primitive}
\end{array}
$$

Figure 3.1: Core calculus types (top) and terms (bottom)

recursion, this is left for future work. See Chapter 8 for details. The implementation given in Chapter 6 allows general recursion and its behaviour is discussed therein.

3.1 Syntax

Fig. 3.1 (top) presents the types of our core calculus. To support inductive types, we include type variables, taken from a countable set ranged over by $\alpha, \beta, \gamma, \ldots$. Our kind system will later ensure these type variables are *strictly positive*: they can only appear free covariantly — to the right of a function type. Variant types use constructor labels taken from a countable set ranged over by $\ell, \ell_1, \ell_2, \ldots$. Variant types are in fact partial functions with a finite domain from the set of constructor labels to the set of types. When σ is a variant type, we write $(\ell\, \tau) \in \sigma$ for the assertion that σ assigns the type τ to ℓ. We include the standard unit type, binary products, and function types. We include unary uninterpreted base types and constructors. While we use a list syntax for variable contexts Γ, they are in fact partial functions with a finite domain from the countable set of variables, ranged over by $x, y, z, \ldots$, to the set of types.

We desugar stand-alone labels in a variant type $\{\cdots \mid \ell \mid \cdots\}$ to the unit type $\{\cdots \mid \ell\,() \mid \cdots\}$. We also desugar seemingly-recursive type declarations $\tau := \sigma[\alpha \mapsto \tau]$ to $\tau := \mu\alpha.\sigma$.

Example 1. The type of booleans is given by $\mathsf{bool} := \{\mathsf{True} \mid \mathsf{False}\}$. The type of natural numbers is given by $\mathbb{N} := \{\mathsf{Zero} \mid \mathsf{Succ}\,\mathbb{N}\}$ desugaring to $\mathbb{N} := \mu\alpha.\{\mathsf{Zero} \mid \mathsf{Succ}\,\alpha\}$. The type of α-lists is given by $\mathsf{List}\,\alpha := \{\mathsf{Nil} \mid \mathsf{Cons}\,\alpha * \mathsf{List}\,\alpha\}$, desugaring to $\mathsf{List}\,\alpha := \mu\beta.\{\mathsf{Nil} \mid \mathsf{Cons}\,\alpha * \beta\}$. □

Base types and constructors allow us to include semantic type declarations into our calculus. For example, we will always include the following base types:

• $\mathbb{I}$: unit interval $[0,1]$; • $\mathbb{R}$: real line $(-\infty,\infty)$; • $\overline{\mathbb{R}}$: extended real line $[-\infty,\infty]$;
• $\mathbb{R}_+$: non-negative reals $[0,\infty)$; • $\overline{\mathbb{R}}_+$: non-negative extended reals $[0,\infty]$.

In addition, once we define a type constructor such as $\mathsf{List}\,\alpha$, we will later reuse it as a base type constructor $\mathsf{List}\,\tau$, effectively working in an extended calculus. Thus we are working with a family of calculi, extending the base signature with each type definition in our development.

Fig. 3.1 (bottom) presents the terms in our core calculus. Variant constructor terms $\tau.\ell\,t$ are annotated with their variant type τ to avoid label clashes. The tupling constructors are standard. We use *iso-inductive* types: construction of inductive types requires an explicit rolling of the inductive definition such as $\mathbb{N}.\mathbf{roll}\,(\mathsf{Zero}())$. Variable binding in function abstraction is *intrinsically typed* in standard Church-style. We include standard pattern matching constructs for variants, binary products, and inductive types. We include a structural recursion construct $\tau.\mathbf{fold}$ for every inductive type τ. Function application is standard, as is the inclusion of primitives.

To ease the construction of terms, we use the standard syntactic sugar (e.g. $\mathbf{let}\,x = t\,\mathbf{in}\,s$ for $(\lambda x.\,t)s$, $\mathbf{if}\,\mathbf{then}\,\mathbf{else}$ for pattern matching booleans), informally elide types from the terms, elide $\mathbf{roll}$ing/unrolling inductive types, and informally use nested pattern matching.

Example 2. For $\mathsf{List}\,\tau = \mu\alpha.\{\mathsf{Nil} \mid \mathsf{Cons}\,\tau * \alpha\}$, we can express standard list manipulation:

$$x :: x_s = \mathsf{Cons}(x, x_s) \qquad \mathsf{foldr}\,a\,f = \mathsf{List}\,\tau.\mathbf{fold}\,\lambda\{\,\mathsf{Nil} \to a \mid \mathsf{Cons}\,(x,b) \to f(x,b)\}$$
$$x_s +\!\!+\, y_s = \mathsf{foldr}\,y_s\,(::)\,x_s \qquad \mathsf{map}\,f\,x_s = \mathsf{foldr}\,[\,]\,(\lambda\{\,(y,y_s) \to (f(y),y_s)\})$$

where we abbreviate $[a_1,\ldots,a_n]$ to $\mathsf{Cons}\,(a_1,\ldots,\mathsf{Cons}\,(a_n,\mathsf{Nil})\ldots)$.

□

3.2 Type System

To ensure the well-formedness of types, which involve type variables, we use a simple kind system, presented in Fig. 3.2. Each kinding judgement $\Delta \vdash_{\mathrm{k}} \tau : \mathsf{type}$ asserts that a given type τ is well-formed in the *type variable context* Δ, which is finite set of type variables.

The kinding judgements are standard. All type variables must be bound by the enclosing context, or by an inductive type binder. The contravariant position in the function type $\tau \to \sigma$ must contain a *closed* type, ensuring that free type variables can only appear in strictly positive positions. Variable contexts Γ must only assign closed types.

$$\frac{}{\Delta \vdash_{\mathrm{k}} \alpha : \mathsf{type}}(\alpha \in \Delta) \qquad \frac{\text{for all } 1 \le i \le n\text{: } \Delta \vdash_{\mathrm{k}} \tau_i : \mathsf{type}}{\Delta \vdash_{\mathrm{k}} \{\ell_1\, \tau_1 \mid \ldots \mid \ell_{\mathsf{n}}\, \tau_n\} : \mathsf{type}} \qquad \frac{}{\Delta \vdash_{\mathrm{k}} \mathbf{1} : \mathsf{type}}$$

$$\frac{\Delta \vdash_{\mathrm{k}} \tau : \mathsf{type} \qquad \Delta \vdash_{\mathrm{k}} \sigma : \mathsf{type}}{\Delta \vdash_{\mathrm{k}} \tau * \sigma : \mathsf{type}} \qquad \frac{\Delta, \alpha \vdash_{\mathrm{k}} \tau : \mathsf{type}}{\Delta \vdash_{\mathrm{k}} \mu\alpha.\tau : \mathsf{type}} \qquad \frac{\vdash_{\mathrm{k}} \tau : \mathsf{type} \qquad \Delta \vdash_{\mathrm{k}} \sigma : \mathsf{type}}{\Delta \vdash_{\mathrm{k}} \tau \to \sigma : \mathsf{type}}$$

$$\frac{}{\Delta \vdash_{\mathrm{k}} \mathrm{A} : \mathsf{type}} \qquad \frac{\Delta \vdash_{\mathrm{k}} \tau : \mathsf{type}}{\Delta \vdash_{\mathrm{k}} \mathsf{F}\,\tau : \mathsf{type}} \qquad \frac{\text{for all } (x : \tau) \in \Gamma\text{: } \vdash_{\mathrm{k}} \tau : \mathsf{type}}{\vdash_{\mathrm{k}} \Gamma : \mathsf{context}}$$

Figure 3.2: Core calculus kind system. The highlighted precondition ensures that free type variables can not appear in the contravariant position. This restriction excludes fully recursive types and only allows inductive types.

$$\frac{}{\Gamma \vdash x : \tau}((x : \tau) \in \Gamma) \qquad \frac{\Gamma \vdash t : \tau_i}{\Gamma \vdash \tau.\ell_i t : \tau}((\ell_{\mathsf{i}}\, \tau_i) \in \tau) \qquad \frac{}{\Gamma \vdash () : \mathbf{1}}$$

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash (t,s) : \tau * \sigma} \qquad \frac{}{\Gamma \vdash \tau.\mathbf{roll} : \big(\sigma[\alpha \mapsto \tau]\big) \to \tau}(\tau = \mu\alpha.\sigma) \qquad \frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \lambda x : \tau.t : \tau \to \sigma}$$

$$\frac{\Gamma \vdash t : \{\ell_1\, \tau_1 \mid \ldots \mid \ell_{\mathsf{n}}\, \tau_n\} \qquad \text{for each } 1 \le i \le n\text{: } \Gamma, x_i : \tau_i \vdash s_i : \tau}{\Gamma \vdash \mathbf{match}\, t\, \mathbf{with}\, \{\ell_1\, x_1 \to s_1 \mid \cdots \mid \ell_{\mathsf{n}}\, x_n \to s_n\} : \tau}$$

$$\frac{\Gamma \vdash t : \sigma * \rho \qquad \Gamma, x : \sigma, y : \rho \vdash s : \tau}{\Gamma \vdash \mathbf{match}\, t\, \mathbf{with}\, (x,y) \to s : \tau} \qquad \frac{\Gamma \vdash t : \mu\alpha.\sigma \qquad \Gamma, x : \sigma[\alpha \mapsto \mu\alpha.\sigma] \vdash s : \tau}{\Gamma \vdash \mathbf{match}\, t\, \mathbf{with}\, \mathbf{roll}\, x \to s : \tau}$$

$$\frac{\Gamma \vdash t : \big(\sigma[\alpha \mapsto \rho]\big) \to \rho}{\Gamma \vdash \tau.\mathbf{fold}\, t : \tau \to \rho}(\tau = \mu\alpha.\sigma) \qquad \frac{\Gamma \vdash t : \sigma \to \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash t\, s : \tau} \qquad \frac{}{\Gamma \vdash \varphi : \tau_\varphi}$$

Figure 3.3: Core calculus type system.

Example 3. The types from Example 1 are well-kinded: $\alpha \vdash_{\mathrm{k}} \mathsf{bool}, \mathbb{N}, \mathsf{List}\,\alpha : \mathsf{type}$. □

We define capture avoiding substitution of types for type variables in the standard way, which obeys the usual structural properties. Henceforth we consider only well-formed types in context, leaving the context implicit wherever possible, and gloss over issues of alpha-convertibility of bound type variables.

To type terms, we assume each primitive φ has a well-formed type $\vdash_{\mathrm{k}} \tau_\varphi : \mathsf{type}$ associated with it. Fig. 3.3 presents the resulting type system. Each typing judgement $\Gamma \vdash t : \tau$ asserts that a given term t is well-typed with the well-formed closed type $\vdash_{\mathrm{k}} \tau : \mathsf{type}$ in the variable context $\vdash_{\mathrm{k}} \Gamma : \mathsf{context}$.

The rules are standard. By design, every term has at most one type in a given context.

Example 4. Once desugared, the list manipulation terms from Example 2 have types:

$$(::) : \tau * \mathsf{List}\,\tau \to \mathsf{List}\,\tau \qquad \mathsf{foldr} : \sigma \to (\tau * \sigma \to \sigma) \to \mathsf{List}\,\tau \to \sigma$$
$$\mathsf{map} : (\tau \to \sigma) \to (\mathsf{List}\,\tau \to \mathsf{List}\,\sigma) \qquad (+\!\!+) : (\mathsf{List}\,\tau) * (\mathsf{List}\,\tau) \to \mathsf{List}\,\tau$$

□

3.3 Primitive Recursion

As is well-known [41, 27], structural recursion on inductive types allows us to express primitive recursion. By 'primitive recursion', we mean recursing through values of an inductive type $\mu\alpha.\sigma$ using a term of the form: $\Gamma, k : \sigma[\alpha \mapsto (\mu\alpha.\sigma) * \rho] \vdash t : \rho$ with the intention that t can use either arbitrary (total) processing on the sub-structures of its input k, or make a primitive recursive call to itself with a sub-structure. In order to desugar such a term into a function of type $\tau * (\mu\alpha.\sigma) \to \rho$, we use terms of the following type, defined by induction on types:

$$\pi_{\alpha.\sigma,\rho} : \sigma[\alpha \mapsto (\mu\alpha.\sigma) * \rho] \to \sigma[\alpha \mapsto \mu\alpha.\sigma]$$

and interpret the primitive recursive declaration t embodied by:

$$\Gamma, x : \mu\alpha.\sigma \vdash \mathbf{match}\,(\mu\alpha.\sigma).\mathbf{fold}\left(\lambda k : \sigma[\alpha \mapsto (\mu\alpha.\sigma) * \rho].\,(\mathbf{roll}\,\pi_{\alpha.\sigma,\rho}\,k, t)\right) x.$$
$$\mathbf{with}\,(\_, r) \to r : \sigma$$

This translation is global in nature: the structure of the term π depends on the type of t. Thus, it does not constitute a *macro* translation [23]. With this point in mind, we will allow ourselves to use primitive recursive definitions.

Example 5. We define a function $\mathsf{aggr} : \mathsf{List}(\mathbb{R}_+ * X) \to \mathsf{List}(\mathbb{R}_+ * X)$ which takes a list of weighted values and aggregates all the weights based on their values. We make use of the auxiliary function $\mathsf{add} : (\mathbb{R}_+ * X) * \mathsf{List}(\mathbb{R}_+ * X) \to \mathsf{List}(\mathbb{R}_+ * X)$, which adds a weighted value to an already aggregated list. We define add by primitive recursion:

$$\begin{aligned}
\mathsf{add}((s,a),x_s) \coloneqq \mathbf{match}\,x_s\,\mathbf{with}\,\{[] &\to [(s,a)] && \text{-- new entry}\\
(r,x) :: x_s &\to \mathbf{if}\quad x = a\\
&\quad\ \mathbf{then}\,(s+r,a) :: x_s && \text{-- accumulate}\\
&\quad\ \mathbf{else}\ \ (r,x) :: \mathsf{add}((s,a),x_s)\} && \text{-- recurse}
\end{aligned}$$

and set aggr := foldr [ ] add. This example makes use of an equality predicate between X elements, restricting its applicability. □

3.4 Denotational Semantics

We give a set-theoretic semantics to the calculus. In such set-theoretic semantics, types-in-context $\Delta \vdash_{\mathrm{k}} \tau : \mathsf{type}$ are interpreted as functors $[\![\tau]\!] : \mathbf{Set}^{\Delta} \to \mathbf{Set}$, i.e., $[\![\tau]\!]$ assigns a set $[\![\tau]\!](X_\alpha)_{\alpha\in\Delta}$ for every Δ-indexed tuple of sets, and a function

$$[\![\tau]\!](f_\alpha : X_\alpha \to Y_\alpha)_{\alpha\in\Delta} : [\![\tau]\!](X_\alpha) \to [\![\tau]\!](Y_\alpha)$$

for every Δ-indexed tuple of functions between the sets with corresponding index, and this assignment preserves composition and identities.

In order to interpret iso-inductive types $\mu\alpha.\tau$, we need canonical isomorphisms between the sets $[\![\tau]\!]([\![\mu\alpha.\tau]\!]) \cong [\![\mu\alpha.\tau]\!]$. We will do this in a standard way, by interpreting $[\![\mu\alpha.\tau]\!]$ as the initial algebra for the functor $[\![\tau]\!] : [\mathbf{Set}^{\Delta} \to \mathbf{Set}] \to [\mathbf{Set}^{\Delta} \to \mathbf{Set}]$. This means that for every functor $A : \mathbf{Set}^{\Delta} \to \mathbf{Set}$ with a natural family of functions $\{a_X : ([\![\tau]\!]A)(X) \to A(X)\}_{X\in\mathbf{Set}^{\Delta}}$, there is a canonical natural family of functions $\{\mathsf{fold}_X : [\![\mu\alpha.\tau]\!](X) \to A(X)\}_{X\in\mathbf{Set}^{\Delta}}$.

A technical requirement is needed to ensure that this initial algebra exists: we fix a regular cardinal κ, and demand that each type denotes a κ-*ranked* functor (ranked functor for short), that is, that it denotes a functor that preserves κ-filtered colimits[1]. The κ-ranked functors are closed under composition, products, sums, and initial algebras. Initial algebras for κ-ranked functors on locally presentable categories always exist, because they can be built in an iterative way by transfinite induction (see e.g. [45]).

Set-Theoretic Interpretation

To interpret types, we assume a given interpretation $\mathscr{B}[\![-]\!]$ of the base types A as sets $\mathscr{B}[\![\mathrm{A}]\!]$ and of base type constructors F as ranked functors $\mathscr{B}[\![\mathsf{F}]\!] : \mathbf{Set} \to \mathbf{Set}$. We then interpret each well-formed type in context $\Delta \vdash_{\mathrm{k}} \tau : \mathsf{type}$ as a ranked functor $[\![\tau]\!] : \mathbf{Set}^{\Delta} \to \mathbf{Set}$, as depicted in Fig. 3.4.

In this definition, the parameter d is either a tuple of sets or a tuple of functions, depending on whether the relevant equations define how the functor in question transforms objects or morphisms. When interpreting type variables, we write $d(\alpha)$ for the α-indexed component

[1] We do not use simpler classes of functors, such as *polynomial functors* or *containers*, as they are not closed under subfunctors, given by subsets in the discrete case and subspaces in the continuous case, which we need in the sequel.

$$[\![\alpha]\!]d \coloneqq d(\alpha) \qquad \left[\!\!\left[\{\ell_1\,\tau_1 \mid \ldots \mid \ell_n\,\tau_n\}\right]\!\!\right]d \coloneqq \sum_{i=1}^{n} [\![\tau_i]\!]d \qquad [\![\mathbf{1}]\!]d \coloneqq \mathbb{1}$$

$$[\![\tau * \sigma]\!]d \coloneqq ([\![\tau]\!]d) \times ([\![\sigma]\!]d) \qquad [\![\mathrm{A}]\!]d \coloneqq \mathscr{B}[\![\mathrm{A}]\!]$$

$$[\![\mathsf{F}\,\tau]\!]d \coloneqq \mathscr{B}[\![\mathsf{F}]\!]([\![\tau]\!]d) \qquad [\![\mu\alpha.\tau]\!]d \coloneqq \mu X.\,[\![\tau]\!]d[\alpha \mapsto X] \qquad [\![\tau \to \sigma]\!]d \coloneqq ([\![\sigma]\!]d)^{[\![\tau]\!]()}$$

Figure 3.4: Core calculus type-level semantics. The highlighted part guarantees existence of the relevant initial algebras. This construction is enabled by the restriction of the type system highlighted in Figure 3.2.

of d. The interpretation of simple types uses disjoint unions, singletons, finite products, and exponentials, i.e. the bi-cartesian closed structure of **Set**. We interpret inductive types $[\![\mu\alpha.\tau]\!]d$ using the initial algebra for the ranked functor $\lambda X.\,[\![\tau]\!]d[\alpha \mapsto X] : \mathbf{Set} \to \mathbf{Set}$. In the semantics of the function type $\tau \to \sigma$, the exponential makes no use of the functor's arguments, and relies on the fact that all type variables are strictly positive. We use the given interpretation of base types and type constructors to interpret them.

Lemma 1. *The semantics of types is well-defined: every well-formed type* $\Delta \vdash_{\mathrm{k}} \tau : \mathsf{type}$ *denotes a ranked functor* $[\![\tau]\!] : \mathbf{Set}^{\Delta} \to \mathbf{Set}$. *In particular, every closed type denotes a set.*

The proof is by induction on the kinding judgements, using well-known properties of **Set**. We will always interpret the base types $\mathbb{I}$, $\mathbb{R}$, etc. by the sets they represent.

Example 6. We calculate the denotations of the types from Example 1. Booleans denote a two-element set $[\![\mathsf{bool}]\!] = \{\mathsf{False}, \mathsf{True}\}$, and the natural numbers denote the set of natural numbers $[\![\mathbb{N}]\!] = \mathbb{N}$. By Lemma 1, $[\![\mathsf{List}]\!]$ denotes a ranked functor $\mathsf{List} : \mathbf{Set} \to \mathbf{Set}$, and this functor is given by the set of sequences of X-elements $\mathsf{List}\,X \coloneqq \bigcup_{n\in\mathbb{N}} X^n$. □

Beyond establishing the well-definedness of the semantic interpretation, Lemma 1 equips us with syntactic means to define ranked functors. Once defined, we can add these functors to our collection of base types (in an extended instance of the core calculus). In the sequel, we will often restrict a given ranked functor $F : \mathbf{Set} \to \mathbf{Set}$ by specifying a family of subsets $GX \subseteq FX$. Doing so is analogous to imposing an *invariant* on a datatype. The subsets GX form a *subfunctor* $G \subseteq F$ precisely if they are closed under the functorial action of F, i.e., for every function $f : X \to Y$ and $a \in GX$, $Ff(a) \in GY$.

Lemma 2. *Subfunctors of ranked functors over* **Set** *are ranked.*

We can prove this lemma directly, but it also follows from a higher-level argument using the commutation of finite limits and κ-directed colimits in **Set**.

instance *Monad* (List) **where**
  return $x = [x]$
  $x_s >\!\!>\!\!= f = \mathsf{foldr}\,[\,]\,\left(\lambda(x, y_s).\, f(x) +\!\!+ y_s\right)\; x_s$

(a) Declaring monadic interfaces

Sugar	Elaboration
• $x \leftarrow t; s$	$t >\!\!>\!\!= \lambda x.\, s$
• **return** t	$\mathrm{return}^{\underline{T}}\, t$
• $t; s$	$\_ \leftarrow t; s$

(b) Haskell's do-notation

Figure 3.5: Monadic programming notation

3.5 Monadic Programming

In the sequel, we will be working with types that support a monadic programming style. More precisely, a *monadic interface* $\underline{T}$ consists of a triple $\underline{T} = (T, \mathrm{return}^{\underline{T}}, >\!\!>\!\!=^{\underline{T}})$ where: T assigns to each set X a set TX; $\mathrm{return}^{\underline{T}}$ assigns to each set X a function $\mathrm{return}^{\underline{T}}_X : X \to TX$; and $>\!\!>\!\!=^{\underline{T}}$ assigns to each pair of sets X and Y a function $>\!\!>\!\!=^{\underline{T}}_{X,Y} : TX \times (TY)^X \to TY$. We borrow Haskell's type-class syntax to define such interfaces. As an example, Fig. 3.5a defines a monadic interface over List.

Each such monadic interface $\underline{T}$ allows us to use standard do-notation summarised in Fig. 3.5b. Though simple in principle, we must take care when treating this notation as syntactic sugar, as choosing the appropriate function return_X or $>\!\!>\!\!=_{X,Y}$ at each desugaring step must take typing information into account. When we use do-notation in the sequel, we ensure that such choices can be disambiguated. Finally, we will delimit our use of do-notation to within a *do-block* $\underline{T}.\mathbf{do}\{\ldots\}$, omitting the monadic interface $\underline{T}$ or the entire delimiter when either is clear from the context.

Importantly, we do not insist that a monadic interface satisfies the monad associativity and unit laws: $(\mathrm{return}\, x) >\!\!>\!\!= f = f(x)$, $a >\!\!>\!\!= \mathrm{return} = a$, and $(a >\!\!>\!\!= f) >\!\!>\!\!= g = a >\!\!>\!\!= (\lambda x.\, (f\, x >\!\!>\!\!= g))$.

3.6 Discrete Inference

We can now lay-out the core ideas in the simpler, set-theoretic case: a semantic structure for higher-order (discrete) probabilistic programs, intermediate representations of these programs for the purpose of inference, valid transformations between these representations, and modular building blocks for creating new representations and transformations from existing ones. For simplicity, we consider representations and transformations from simple rather naive inference algorithms only in this section. In Chapter 5 we show how the core ideas here apply to advanced algorithms when aided with further technical developments.

3.6.1 The Mass Function Monad

For our purposes, probabilistic programming languages contain standard control-flow mechanisms and data types, such as our core calculus, together with *probabilistic choice* and *conditioning* operations. In the discrete case, these are given by two effectful operations:

$$\frac{}{\Gamma \vdash_{\text{comp}} \mathbf{flip} : \mathsf{bool}} \qquad \frac{\Gamma \vdash t : \mathbb{R}_+}{\Gamma \vdash_{\text{comp}} \mathbf{score}\, t : \mathbf{1}}$$

In Bayesian probabilistic programming, we think of **flip** as drawing from a (uniform) prior distribution on bool, and of **score** as recording a likelihood. Typically, one calls $\mathbf{score}(f(x))$ where f is a density function of a distribution, which records the likelihood of observing data x from the distribution f. The score might be zero, indicating a hard constraint - this path is impossible. The score might be in the unit interval, corresponding to the probability of a discrete observation. In general the score is a likelihood function can take any positive real value. The inference problem is to approximate the posterior distribution, from the unnormalized posterior defined by the program, combining a prior and likelihood.

To give a set-theoretic semantic structure to such a higher-order language with these two constructs, it suffices to give a monadic interface $\underline{T}$ for which the associativity and unit laws hold, together with two functions:

$$\mathrm{flip} : [\![\mathbf{1}]\!] \to T\,[\![\mathsf{bool}]\!] \qquad\qquad \mathrm{score} : [\![\mathbb{R}_+]\!] \to T\,[\![\mathbf{1}]\!]\,.$$

For the purposes of the discrete development, the following monad fits the bill. A (finite) *mass function* over a set X is a function $\mu : X \to \mathbb{R}_+$ for which there exists a finite set $F \subseteq X$ such that μ is 0 outside F: in other words, the support set $\operatorname{supp}\mu \coloneqq \{x \in X \,|\, \mu(x) \neq 0\}$ is finite. For every set X, let $\mathsf{Mass}\,X \coloneqq \{\mu : X \to \mathbb{R}_+ \,|\, \mu \text{ is a mass function}\}$. The *mass function* monad is given by:

$$\begin{aligned}\underline{\mathsf{Mass}} \coloneqq \mathbf{instance}\ \mathit{Monad}\,(\mathsf{Mass})\ \mathbf{where}&\\ \mathrm{return}\,x_0 &= \lambda x.\ \mathbf{if}\ (x = x_0)\ \mathbf{then}\ 1\ \mathbf{else}\ 0\\ \mu \gg\!= f &= \lambda y.\ \textstyle\sum_{x \in \operatorname{supp}\mu} \mu(x) \cdot (f(x)(y))\end{aligned}$$

and we set $\mathrm{flip} = \lambda\_.\,\frac{1}{2}$ and $\mathrm{score}\,r = \lambda\{\,() \to r\}$. Intuitively, values of $\mathsf{Mass}\,X$ represent unnormalized probabilistic computations of a result in X. From the Bayesian perspective, the meaning of a program is the unnormalized posterior.

Lemma 3. *The monadic interface* $\underline{\mathsf{Mass}}$ *defines a ranked monad over* **Set**.

This monad is also known as the *free positive cone monad*, as it constructs the 'positive fragment' of a vector space over the field of reals with basis X.

3.6.2 Inference Representations

The mass function semantics is accurate, but idealised: realistic implementations cannot be expected to compute mass functions at arbitrary types, and especially at higher-order types. Instead, probabilistic inference engines would manipulate some representation of the program, while maintaining its semantics.

Definition 4. *A* discrete inference representation $\underline{T}$ *is a sextuple*

$$\underline{T} = \left(T, \mathrm{return}^{\underline{T}}, \gg\!=^{\underline{T}}, \mathrm{flip}^{\underline{T}}, \mathrm{score}^{\underline{T}}, m^{\underline{T}}\right)$$

consisting of:

- *a monadic interface* $\left(T, \mathrm{return}^{\underline{T}}, \gg\!=^{\underline{T}}\right)$*;*
- *two functions* $\mathrm{flip}^{\underline{T}} : \mathbb{1} \to T2$ *and* $\mathrm{score}^{\underline{T}} : \mathbb{R}_+ \to T\mathbb{1}$*, where* $\mathbb{1} := [\![\mathbf{1}]\!]$*,* $2 := [\![\mathsf{bool}]\!]$*; and*
- *an assignment of a* meaning *function* $m^{\underline{T}}_X : TX \to \mathsf{Mass}X$ *for every set* X

such that the following laws hold for all sets X*,* Y*, and* $x \in X$*,* $a \in TX$*,* $r \in \mathbb{R}_+$*, and* $f : X \to TY$*:*

$$\mathrm{return}^{\underline{\mathsf{Mass}}} x = m(\mathrm{return}^{\underline{T}} x) \qquad m(a \gg\!=^{\underline{T}} f) = (ma) \gg\!=^{\underline{\mathsf{Mass}}} \lambda x.\, m(f\ x)$$

$$m(\mathrm{flip}^{\underline{T}}) = \mathrm{flip}^{\underline{\mathsf{Mass}}} \qquad m(\mathrm{score}^{\underline{T}} r) = \mathrm{score}^{\underline{\mathsf{Mass}}} r$$

As with monadic interfaces, we use a type-class notation for defining inference representations.

Example 7 (Discrete weighted sampler)**.** Consider the type

$$\mathsf{Term}\,\alpha := \{\mathsf{Return}\,(\mathbb{R}_+ * \alpha) \mid \mathsf{Flip}\,(\mathsf{Term}\,\alpha * \mathsf{Term}\,\alpha)\},$$

which induces a ranked functor Term. The elements of $\mathsf{Term}X$ are binary trees, which we call terms, whose leaves contain weighted values of type X. Fig. 3.6a presents the inference representation structure of the functor Term. Flip represents a probabilistic choice while Return holds the final value and the total weight for the branch. Thus an immediately returning computation is represented by a leaf with weight 1. The auxiliary function *scale* in the definition of $\gg\!=$ scales the leaves of its input term by the input weight. The function $\gg\!=$ itself substitutes terms for the leaves according to its input function f, making sure the newly grafted terms are scaled appropriately. The probabilistic choice operation **flip** constructs a single node with each leaf recording the probabilistic choice *unweighted*. Conditioning records the input weight.

$$
\begin{array}{l}
\textbf{instance}\ \textit{Discrete Monad}\,(\mathsf{Term})\ \textbf{where}\\
\quad \textbf{return}\,x = \mathsf{Return}\,(1,x)\\
\quad a >\!\!>\!= f = \textbf{let}\,(\mathit{scale} : \mathbb{R}_+ * \mathsf{Term}\,X \to \mathsf{Term}\,X) = \qquad -\!-\ \text{uses primitive recursion}\\
\qquad\qquad \lambda s.\,\lambda\{\mathsf{Return}\,(r,x) \to \mathsf{Return}\,(s\cdot r,x)\\
\qquad\qquad\quad \big|\mathsf{Flip}\,(k_{\mathsf{False}},k_{\mathsf{True}}) \to \mathsf{Flip}\,(\mathit{scale}(s,k_{\mathsf{False}}),\mathit{scale}(s,k_{\mathsf{True}}))\}\\
\qquad\quad \textbf{in}\ \ \textbf{match}\,a\,\textbf{with}\,\{\\
\qquad\qquad\quad \mathsf{Return}\,(r,x) \to \mathit{scale}(r,f\ x)\\
\qquad\qquad\quad \big|\mathsf{Flip}\,(k_{\mathsf{False}},k_{\mathsf{True}}) \to \mathsf{Flip}\,(k_{\mathsf{False}} >\!\!>\!= f,\ -\!-\ \text{uses primitive recursion}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad k_{\mathsf{True}} >\!\!>\!= f)\}\\
\quad \textbf{flip} \quad = \mathsf{Flip}\,(\mathsf{Return}\,(1,\mathsf{False}),\mathsf{Return}\,(1,\mathsf{True}))\\
\quad \textbf{score}\,r \ = \mathsf{Return}\,(r,())\\
\quad m\,a \quad = \textbf{fold}\,\lambda\{\mathsf{Return}\,(r,x) \to \mathsf{Mass}\,.\textbf{do}\,\{\textbf{score}\,r;\textbf{return}\,x\}\\
\qquad\qquad\quad \big|\mathsf{Flip}\,(\mu_{\mathsf{False}},\mu_{\mathsf{True}}) \to \mathsf{Mass}\,.\textbf{do}\,\{x \leftarrow \textbf{flip};\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{if}\ x\ \textbf{then}\ \mu_{\mathsf{True}}\ \textbf{else}\ \mu_{\mathsf{False}}\}\}
\end{array}
$$

(a) Discrete weighted sampler representation

$$
\begin{array}{l}
\textbf{instance}\ \textit{Discrete Monad}\,(\mathsf{Enum})\ \textbf{where}\\
\quad \textbf{return}\,x = [(1,x)]\\
\quad x_s >\!\!>\!= f = \textbf{let}\,(\mathit{scale} : \mathbb{R}_+ * \mathsf{Enum}\,X \to \mathsf{Enum}\,X) =\\
\qquad\qquad \lambda\{(r,x_s) \to \mathrm{map}\,\lambda\{(s,y) \to (r\cdot s,y)\}\\
\qquad\qquad\qquad\qquad x_s\}\\
\qquad \textbf{in}\ \ \mathsf{foldr}\,[\,]\\
\qquad\qquad \lambda\{((r,x),y_s) \to \mathit{scale}(r,f\ x) +\!\!+ y_s\}\\
\qquad\qquad x_s\\
\quad \mathrm{flip} \quad = [(\tfrac{1}{2},\mathsf{False}),(\tfrac{1}{2},\mathsf{True})]\\
\quad \mathrm{score}\,r \ = [(r,())]\\
\quad m\,x_s \quad = \lambda a. \qquad -\!-\ \ m\,x_s\ a = \sum_{\substack{(r,x)\in x_s\\ x=a}} r\\
\qquad\qquad \mathsf{foldr}\,0\left(\begin{array}{l}\lambda\{((r,x),s) \to\\ \quad \textbf{if}\ x = a\ \textbf{then}\ r+s\ \textbf{else}\ s\}\end{array}\right)\ x_s
\end{array}
$$

(b) Discrete enumeration sampler

$$
\begin{array}{l}
\textbf{instance}\ \textit{Inf Trans}\,(\mathsf{W})\ \textbf{where}\\
\quad \textbf{lift}_{\underline{T}}\,a \quad = \underline{T}.\textbf{do}\,\{x \leftarrow a;\\
\qquad\qquad\qquad\qquad \textbf{return}(1,x)\}\\
\quad \textbf{return}_{\mathsf{W}\,\underline{T}}\,x = \mathrm{return}^{\underline{T}}(1,x)\\
\quad a >\!\!>\!=_{\mathsf{W}\,\underline{T}}\,f = \underline{T}.\textbf{do}\,\{(r,x) \leftarrow a;\\
\qquad\qquad\qquad\qquad (s,y) \leftarrow f(x);\\
\qquad\qquad\qquad\qquad \textbf{return}(r\cdot s,y)\}\\
\quad \textbf{flip}_{\mathsf{W}\,\underline{T}} \quad = \textbf{lift}\,\textbf{flip}^{\underline{T}}\\
\quad \textbf{score}_{\mathsf{W}\,\underline{T}}\,r \ = \mathrm{return}^{\underline{T}}(r,())\\
\quad m_{\mathsf{W}\,\underline{T}}\,a \quad = \lambda x.\ \sum_{(r,x)\in \mathrm{supp}\,m^T(a)} r\\
\quad (\textbf{tmap}\,\underline{t})_X \ = \underline{t}_{\mathbb{R}_+ * X}
\end{array}
$$

(c) Discrete weighting transformer

Figure 3.6: Example inference representations (a,b) and transformers (c)

The meaning function recurses over the term, replacing each node representing a probabilistic choice by probabilistic choice of the mass function monad, and reweighting the end result appropriately.

The main step in validating the inference representation laws involves >>=: first show that composing the meaning function with the auxiliary function *scale* scales the meaning of the input term appropriately, and then proceed by structural induction on terms. □

The weighted sampler representation in fact forms a proper monad over **Set**: it is the free monad for an algebraic theory with a binary operation flip and unary operations score_r subject to $\mathbf{flip}(\mathbf{score}_r(x), \mathbf{score}_r(y)) = \mathbf{score}_r(\mathbf{flip}(x,y))$. As the mass function monad also validates these equations, the meaning function is then the unique monad morphism from Term to Mass preserving the operations **flip** and **score**.

However, we emphasise that an inference representation need not form a proper monad, and that the meaning function need not be a monad morphism. Indeed, the PopSam representation introduced in Section 5.2 is not a monad and most of the non-trivial inference transformations we discuss are not monad morphisms.

The weighted sampler representation allows us to incorporate both intensional and operational aspects into our development. Bayesian inference ultimately reduces a representation into probabilistic simulation. The weighted sampler representation can thus act as an internal representation of this simulation. Moreover, its continuous analogue will allow us to manipulate traces when analysing the Trace Markov Chain Monte Carlo algorithm in Section 5.4.

Example 8 (Enumeration)**.** The type $\mathsf{Enum}\,\alpha := \mathsf{List}(\mathbb{R}_+ * \alpha)$ induces a ranked functor Enum. Elements of $\mathsf{Enum}\,X$ form an enumeration of the mass function they represent, with the same value x potentially appearing multiple times with different weights. Values not appearing in the list at all have weight 0.

Fig. 3.6b presents an inference representation structure using Enum. Returning a value lists the unique non-zero point mass. The >>= operation applies the given function to each element listed, scales the list appropriately and accumulates all intermediate lists. The choice operation enumerates both branches with equal probability, and conditioning inserts a scaling factor. The meaning function assigns to an element the sum of its weights. This definition uses an equality predicate.

Establishing the inference representation laws is straightforward. □

3.6.3 Inference Transformations

We can now define the central validity criterion in our development. We decompose Bayesian inference algorithms into smaller transformations between inference representations. To

be correct, these transformations need to preserve the meaning of the representation they manipulate:

Definition 5. *Let $\underline{T}$, $\underline{S}$ be two inference representations. A* discrete inference transformation *$\underline{t} : \underline{T} \to \underline{S}$ assigns to each set X a function $\underline{t}_X : TX \to SX$ satisfying $m^{\underline{T}}(a) = m^{\underline{S}}(\underline{t}_X(a))$ for every $a \in \underline{T}X$.*

This validity criterion guarantees nothing beyond the preservation of the overall mass function of our representation. The transformed representation may not be better for inference along any axis, such as better convergence properties or execution time. It is up to the inference algorithm designer to convince herself of such properties by other means: formal, empirical, or heuristic.

Some transformations change the representation type:

Example 9 (Enumeration)**.** Define a transformation: $\underline{t} : \underline{\mathsf{Term}} \to \underline{\mathsf{Enum}}$ by:

$$\begin{aligned}\underline{t} \coloneqq \lambda\{\, \mathsf{Return}\,(r,x) \quad &\to \mathsf{Enum}\,.\mathbf{do}\,\{\mathbf{score}\,r; \mathbf{return}\,x\} \\ \mid \mathsf{Flip}\,(x_s^{\mathsf{False}}, x_s^{\mathsf{True}}) &\to \mathsf{Enum}\,.\mathbf{do}\,\{b \leftarrow \mathbf{flip}; \mathbf{if}\,b\,\mathbf{then}\,x_s^{\mathsf{True}}\,\mathbf{else}\,x_s^{\mathsf{False}}\}\}\end{aligned}.$$

Straightforward calculation shows it preserves the meaning functions. □

The last example is a special case: analogous functions form inference transformations $\underline{t_T} : \underline{\mathsf{Term}} \to \underline{T}$ for every discrete inference representation $\underline{T}$. To establish meaning preservation, calculate that both $m^{\underline{\mathsf{Term}}}$ and $m^{\underline{T}} \circ \underline{t_T}$ are monad morphisms that preserve probabilistic choice and conditioning and appeal to the initiality of $\underline{\mathsf{Term}}$.

An inference transformation need not be natural:

Example 10 (Aggregation)**.** Recall the functions $\mathrm{aggr}_X : \mathsf{List}(\mathbb{R}_+ * X) \to \mathsf{List}(\mathbb{R}_+ * X)$ from Example 5 which aggregate list elements according to their X component by summing their weights. It forms an inference transformation $\mathrm{aggr} : \underline{\mathsf{Enum}} \to \underline{\mathsf{Enum}}$. The meaning preservation proof uses straightforward structural induction. Note that aggr is not a natural transformation. □

3.6.4 Inference Transformers

We can decompose the weighted sampler representation $\underline{\mathsf{Term}}$, which forms a monad, by transforming the *discrete sampler* representation $\mathsf{DSam}\,X \coloneqq \{\mathsf{Return}\,X \mid \mathsf{Sample}\,(\mathsf{DSam}\,X * \mathsf{DSam}\,X)\}$ with the following *writer monad transformer* $\mathsf{W}\,T\,X \coloneqq T(\mathbb{R}_+ * X)$, i.e. $\mathsf{Term} = \mathsf{W}\,\mathsf{DSam}$. Such decompositions form basic building blocks for constructing and reasoning about more sophisticated representations.

Definition 6. *An* inference transformer $\underline{F}$ *is a triple* $(F, \mathsf{tmap}^{\underline{F}}, \mathsf{lift}^{\underline{F}})$ *whose components assign:*

- *inference representation* $F\,\underline{T}$ *to every inference representation* $\underline{T}$*;*
- *inference transformation* $\mathsf{tmap}^{\underline{F}}\,\underline{t} : F\,\underline{T} \to F\,\underline{S}$ *to every inference transformation* $\underline{t} : \underline{T} \to \underline{S}$*; and*
- *inference transformation* $\mathsf{lift}_T : \underline{T} \to F\,\underline{T}$ *to every inference representation* $\underline{T}$*.*

We use type-class notation for defining inference transformers.

Example 11. The *weighting* inference transformer structure on $\mathsf{W}\,T\,X \coloneqq T(\mathbb{R}_+ * X)$ is given in Fig. 3.6c. We lift a representation in $\underline{T}$ into $\mathsf{W}\,\underline{T}$ by assigning weight 1 to it. The monadic interface uses the standard writer monad for the multiplication structure on $\mathbb{R}_+$, accumulating the weights as computation proceeds. We lift the probabilistic choice from $\underline{T}$, but crucially we reimplement a *new* conditioning operation using the explicitly given weights. The mass function meaning of a representation then accumulates the mass of all weights associated to a given value. We transform an inference transformation by picking the component of the appropriate type.

It is straightforward to show that $\mathsf{W}\,\underline{T}$ is an inference representation, using preservation of return and $\gg\!=$ by the meaning function to reduce the proof to manipulations of weighted sums over $\mathbb{R}_+$. Establishing the validity of lift and tmap is straightforward. □

The weighting transformer *augments* the representation with a new conditioning operation, but *transforms* its choice operation to the new representation. We will later see more examples of both kinds.

3.6.5 Summary

We have introduced our three core abstractions, inference representations, transformations, and transformers, in relation to a mathematical semantic structure, the mass function monad. The examples so far show that the higher-order structure in our core calculus acts as a useful glue for manipulating and defining these abstractions. In the continuous case, we will also use this higher-order structure to represent computations over the real numbers.

Chapter 4

Mathematical Tools for Continuous Semantics

In this chapter we develop the mathematical machinery required to give denotational semantics to the calculus extended with continuous distributions. The principal difficulty associated with such a construction lies in giving measurable space structure to sets of higher-order functions in a systematic way. Attempting to do this naively results in the application functional, which takes a function and an argument and returns the result of applying the function to the argument, not being a measurable function itself [5]. In technical terms the category of measurable spaces is not Cartesian closed. This issue proved to be challenging with many authors such as Staton et al. [84] and Ehrhard et al. [21] devoting considerable effort to dealing with it.

Intuitively this problem is artificial, in the sense that mathematical objects that cause issues with measurability do not correspond to actual probabilistic programs that can be expressed in the relevant languages. This is because probabilistic programs usually only include distributions over discrete and Euclidean spaces as primitives and it is not possible to construct arbitrary distributions over function spaces. Any such distribution is in fact a push-forward of a distribution over numeric types.

This insight was used by Heunen et al. [38] to devise quasi-Borel spaces (QBS), which only allow certain types of distributions, namely ones obtained by a push-forward of a distribution over the real line. QBS form a Cartesian closed category which is suitable for developing denotational semantics of probabilistic programs with continuous distributions and higher-order functions.

In this chapter we review the relevant concepts in category theory and the construction of QBS. We also construct a probability monad suitable for our semantic construction and show that it has the required properties. Since working directly in QBS is counterintuitive, we also provide a synthetic measure theory for QBS which allows us to work with notation very

similar to classical measure theory in the subsequent chapter. The readers not already familiar with category theory will likely find this chapter difficult to follow. Fortunately, the technical developments presented in this chapter are well encapsulated, so the reader can afford to skim this chapter and still be able to follow the rest of the dissertation. We would ask such a reader to focus their attention in this chapter on Section 4.2, in particular on Figures 4.1 and 4.2 which list the notations and properties of the synthetic measure theory that we use in Chapter 5.

This chapter is not original work of the author of this dissertation but rather the author's review of work done by others. Some of the constructions presented in this chapter were introduced as novel in [80] where they were mostly developed by the remaining authors.

4.1 Category Theory

Basic Notions. We assume basic familiarity with categories $\mathscr{C}$, $\mathscr{D}$, functors $F, G : \mathscr{C} \to \mathscr{D}$, and natural transformations $\alpha, \beta : F \to G$, and their theory of limits, colimits, and adjunctions. To fix notation, a *cartesian closed category* is a category with finite products, denoted by $\mathbb{1}$, $\times$, $\prod_{i=1}^{n}$, and exponentials, denoted by X^Y. In this subsection, we use the fragment of our core calculus consisting of the simply-typed λ-calculus (with sums, if necessary) to more compactly review the relevant concepts.

Monads. A *strong monad* $\underline{T}$ over a cartesian closed category is a triple $(T, \mathrm{return}, >\!\!>\!\!=)$ consisting of an assignment of an object TX and a morphism $\mathrm{return}_X : X \to TX$ for every object X, and an assignment of a morphism $>\!\!>\!\!=_{X,Y} : TX \times (TY)^X \to TY$, satisfying the monad laws from § 3.5. Given a monad $\underline{T}$, a *$\underline{T}$-algebra* A is a pair $(|A|, >\!\!>\!\!=^A)$ consisting of an object $|A|$, called the *carrier*, and an assignment of a morphism $>\!\!>\!\!=^A_X : |A|^X \to |A|^{TX}$ to every object X satisfying

$$(\mathrm{return}\, x >\!\!>\!\!=^A f) = f\, x \quad \text{and} \quad ((a >\!\!>\!\!= f) >\!\!>\!\!=^A g) = a >\!\!>\!\!= (\lambda x.\, f(x) >\!\!>\!\!=^A g).$$

The pair $(TX, >\!\!>\!\!=)$ always forms a T-algebra called the *free* $\underline{T}$-algebra over X. The *Eilenberg-Moore* category $\mathscr{C}^{\underline{T}}$ for a monad $\underline{T}$ consists of $\underline{T}$-algebras and their homomorphism. The *Kleisli* category $\mathscr{C}_{\underline{T}}$ consists of the same objects as $\mathscr{C}$, but morphisms from X to Y in $\mathscr{C}_{\underline{T}}$ are morphisms $X \to TY$ in $\mathscr{C}$. The Kleisli category $\mathscr{C}_{\underline{T}}$ inherits any coproducts $\mathscr{C}$ has. A strong monad $\underline{T}$ is *commutative* when, for every

$$a : \underline{T}X, b : \underline{T}Y \vdash \ \underline{T}.\mathbf{do}\{x \leftarrow a; y \leftarrow b; \mathbf{return}(x,y)\} \quad = \quad \underline{T}.\mathbf{do}\{y \leftarrow b; x \leftarrow a; \mathbf{return}(x,y)\}.$$

(The notion of strong/commutative monad is due to [47]; our formulation of algebras also appears in [56].)

Biproducts. A zero object $\mathbb{Z}$ is both initial and terminal. A category has *(finite, countable, etc.) biproducts* if it has a zero object (and hence zero morphisms $\mathbf{0}_{X,Y} : X \to \mathbb{Z} \to Y$) and the following canonical morphisms are invertible:

$$\left[\left(\delta_{i,j}\right)_{j\in I}\right]_{i\in I} : \sum_{i\in I} X_i \to \prod_{j\in I} X_j \qquad \text{where: } \delta_{i,i} \coloneqq \mathrm{id}_{X_i},\ \delta_{i,j} \coloneqq \mathbf{0}_{X_i,X_j} \text{ for } i \neq j.$$

Algebraic Structure. Recall the notion of a *commutative monoid* $(M, 1, \cdot)$ in a category with finite products. We extend it to countably many arguments. Let $\mathscr{C}$ be a category with countable products. A *σ-monoid* (see also [36]) is a triple $(M, 0, \Sigma)$ consisting of: an object M; a morphism $0 : \mathbb{1} \to M$; and a morphism $\Sigma : M^{\mathbb{N}} \to M$ such that:

- setting $\delta_0 \coloneqq \mathrm{id}_M : M \to M$ and $\delta_i \coloneqq 0 \circ ! : M \to \mathbb{1} \to M$, $i > 0$, we have $\Sigma \circ (\delta_i)_{i\in\mathbb{N}} = \delta_0$; and
- for every bijection $\varphi : \mathbb{N} \cong \mathbb{N} \times \mathbb{N}$, $a_{(-,-)} : M^{\mathbb{N}\times\mathbb{N}} \vdash \Sigma\left(\Sigma\left(a_{(i,j)}\right)_{j\in\mathbb{N}}\right)_{i\in\mathbb{N}} = \Sigma\left(a_{\varphi(k)}\right)_{k\in\mathbb{N}}$.

4.2 Synthetic Measure Theory

Synthetic mathematics identifies structure and axioms from which we can recover the main concepts and results of specific mathematical theories, and transport them to new settings. We now briefly recount the relevant parts of Kock's development [48]. (In the finite discrete case, this is also related to Jacob's work on effectuses [42].)

4.2.1 Axioms and Structure

Let $\mathscr{C}$ be a cartesian closed category with countable products and coproducts, and let $\underline{\mathrm{M}}$ be a commutative monad over $\mathscr{C}$. If the morphism $! : \mathrm{M}\,\mathbb{0} \to \mathbb{1}$ is invertible, then both the Eilenberg-Moore category $\mathscr{C}^{\underline{\mathrm{M}}}$ and the Kleisli category $\mathscr{C}_{\underline{\mathrm{M}}}$ have zero objects. As a consequence, we have a canonical $\underline{\mathrm{M}}$-homomorphism $>\!\!>= \left[\left(\delta_{i,j}\right)_j\right]_i : \mathrm{M}\sum_{i\in\mathbb{N}} X_i \to \prod_{j\in\mathbb{N}} \mathrm{M}X_j$.

Definition 7. *A* measure category *is a pair $(\mathscr{C}, \underline{\mathrm{M}})$ consisting of a cartesian closed category $\mathscr{C}$ with countable products and coproducts, and equalisers; and a commutative monad $\underline{\mathrm{M}}$ over $\mathscr{C}$ such that the morphisms $! : \mathrm{M}\,\mathbb{0} \to \mathbb{1}$ and $>\!\!>= \left[\left(\delta_{i,j}\right)_j\right]_i : \mathrm{M}\sum_{i\in\mathbb{N}} X_i \to \prod_{j\in\mathbb{N}} \mathrm{M}X_j$ are invertible.*

We fix a measure category $(\mathscr{C}, \underline{\mathrm{M}})$ for the remainder of this section. The intuition is that $\mathrm{M}X$ is the object of distributions/measures over X. 'Kock' shows that, while short, the above definition has surprisingly many consequences.

Both the Eilenberg-Moore and the Kleisli categories have countable biproducts, and as a consequence, all $\underline{\mathrm{M}}$-algebras have a σ-monoid structure and all $\underline{\mathrm{M}}$-homomorphisms are σ-monoid homomorphisms with respect to it. Moreover, this structure on the free algebra on the terminal object $R \coloneqq \mathrm{M}\mathbb{1}$ extends to a σ-semiring structure by setting: $1 \coloneqq \mathbf{return}()$ and $r \cdot s \coloneqq \underline{\mathrm{M}}.\mathbf{do}\,\{r; s\}$. Kock calls this structure the *σ-semiring of scalars*. Each $\underline{\mathrm{M}}$-algebra A has an R-module structure

$$r : R, a : |A| \vdash r \odot a \coloneqq \underline{\mathrm{M}}.\mathbf{do}\,\{r; a\}$$

. As $\mathscr{C}$ has equalisers, for each object X, we may form the equaliser $\mathrm{P}X \overset{\mathrm{sub}_X}{\dashrightarrow} \mathrm{M}X \underset{\underline{1}}{\overset{\mathrm{M}!}{\rightrightarrows}} R$ because $R = \mathrm{M}\mathbb{1}$. Each sub_X is monic, the monadic structure factors through sub turning $\underline{\mathrm{P}}$ into a commutative monad $\underline{\mathrm{P}}$, and $\mathrm{sub} : \underline{\mathrm{P}} \rightarrowtail \underline{\mathrm{M}}$ into a strong monad monomorphism.

The morphism $\mathrm{M}! : \mathrm{M}X \to R$ is called the *total measure* morphism, and P is then the subobject of all the measures with total measure 1, and so we think of it as the object of *probability measures* over X. For example, every $\underline{\mathrm{P}}$-algebra is closed under *convex* linear combinations of scalars: if $r_\_ : \mathbb{N} \to R$ satisfies $\Sigma(r_i)_i = 1$ then $\underline{\mu}_\_ : (\mathrm{P}X)^{\mathbb{N}} \vdash \mathrm{M}!(\Sigma\left(r_i \odot \underline{\mu}_i\right)_i) = 1$.

4.2.2 Notation and Basic Properties

Kock's theory shines brightly when we adopt a measure-theoretic notation, as in Fig. 4.1, by thinking of $\mathrm{M}X$ as the object of measures over X, and R as the object of scalars these measures take values in. The functorial action of the monad allows us to push measures along morphisms, and pushing all the measure into the terminal object gives a scalar we think of as the total measure of an object. The monadic return acts as a Dirac distribution. The main advantage is the *Kock integral*, synonymous to the monadic $\gg\!=$. The main difference between the Kock integral $\oint$ and the usual Lebesgue integral $\int$ from measure theory is that the Kock integral evaluates to a *measure*, and not a scalar. Calculating with the Kock integral is analogous to using Lebesgue integrals with respect to a generic test function, and proceeding by algebraic manipulation. The scalar rescaling $\odot$ allows us to rescale a distribution by an arbitrary weight function. A *kernel* is a morphism $k : X \to \mathrm{M}Y$, and we use the usual notation for integration against a kernel and iterated integration. We define the product measure by iterated integration. Finally, the $\gg\!=$ operation of an $\underline{\mathrm{M}}$-algebra A gives rise to an expectation operation. Here we will only make use of the scalars' algebra structure, which generalises the usual Lebesgue integral.

	Notation	Meaning	Terminology
	R	$:= \mathrm{M}\mathbb{1}$	Scalars
$f : Y^X, \underline{\mu} : \mathrm{M}X$	$\vdash f_* \underline{\mu}$	$:= (\mathrm{M}f)(\underline{\mu})$	Push-forward
$\underline{\mu} : \mathrm{M}X$	$\vdash \underline{\mu}(X)$	$:= !_* \underline{\mu}$	The total measure
$x : X$	$\vdash \underline{\delta}_x$	$:= \mathbf{return}(x)$	Dirac distribution
$\underline{\mu} : \mathrm{M}X, f : (\mathrm{M}Y)^X$	$\vdash \int_X f(x)\underline{\mu}(\mathrm{d}x)$	$:= \underline{\mu} >\!\!>\!= f$	Kock integral
$w : R^X, \underline{\mu} : \mathrm{M}X$	$\vdash w \odot \underline{\mu}$	$:= \int_X (w(x) \odot \underline{\delta}_x)\underline{\mu}(\mathrm{d}x)$	Rescaling
$\begin{bmatrix} f : (TZ)^{X\times Y}, \\ x : X, k : (TY)^X \end{bmatrix}$	$\vdash \int_Y f(x,y)k(x,\mathrm{d}y)$	$:= \int_Y f(x,y)k(x)(\mathrm{d}y)$	Kernel integration
$\begin{bmatrix} f : (\mathrm{M}X)^{X\times Y}, \\ \underline{\mu} \in \mathrm{M}(X\times Y) \end{bmatrix}$	$\vdash \iint_{X\times Y} f(x,y)\underline{\mu}(\mathrm{d}x,\mathrm{d}y)$	$:= \int_{X\times Y} f(z)\underline{\mu}(\mathrm{d}z)$	Iterated integrals
$\underline{\mu} : \mathrm{M}X, \underline{\nu} : \mathrm{M}Y$	$\vdash \underline{\mu} \otimes \underline{\nu}$	$:= \int_X \left(\int_Y \underline{\delta}_{(x,y)} \underline{\nu}(\mathrm{d}y) \right) \underline{\mu}(\mathrm{d}x)$	Product measure
$\underline{\mu} : \mathrm{M}X, f : \lvert A \rvert^X$	$\vdash \mathbb{E}^A_{x\sim\underline{\mu}}[f(x)]$	$:= \underline{\mu} >\!\!>\!= f$	Expectation
$f : R^X, \underline{\mu} : \mathrm{M}X$	$\vdash \int_X f(x)\underline{\mu}(\mathrm{d}x)$	$:= \mathbb{E}^R_{x\sim\underline{\mu}}[f(x)]$	Lebesgue integral

Figure 4.1: Synthetic measure theory notation

The justification for this notation is that it obeys the expected properties, which we now survey. The commutativity of the monad lets us change the order of integration:

Theorem 8 (Fubini-Tonelli)**.** *For every pair of objects X, Y in a measure category $(\mathscr{C}, \underline{\mathrm{M}})$:*

$$\iint_{X\times Y} f(x,y)(\underline{\mu} \otimes \underline{\nu})(\mathrm{d}x,\mathrm{d}y) \;=\; \int_X \underline{\mu}(\mathrm{d}x) \int_Y \underline{\nu}(\mathrm{d}y) f(x,y) \;=\; \int_Y \underline{\nu}(\mathrm{d}y) \int_X \underline{\mu}(\mathrm{d}x) f(x,y).$$

Moreover, for every M*-algebra A:*

$$\underline{\mu} : \mathrm{M}X, \underline{\nu} : \mathrm{M}Y, f : \lvert A \rvert^{X\times Y} \vdash \mathop{\mathbb{E}^A}_{\substack{x\sim\underline{\mu} \\ y\sim\underline{\nu}}}[f(x,y)] = \mathop{\mathbb{E}^A}_{x\sim\underline{\mu}}[\mathop{\mathbb{E}^A}_{y\sim\underline{\nu}}[f(x,y)]] = \mathop{\mathbb{E}^A}_{y\sim\underline{\nu}}[\mathop{\mathbb{E}^A}_{x\sim\underline{\mu}}[f(x,y)]].$$

As usual, we allow placing the binder $\underline{\mu}(\mathrm{d}x)$ on either side of the integrand $f(x)$.

The push-forward operation interacts with rescaling in the following way:

Theorem 9 (Frobenius reciprocity)**.** *For all objects X, Y in a measure category $(\mathscr{C}, \underline{\mathrm{M}})$:*

$$w : R^X, \underline{\mu} : \mathrm{M}X, f : Y^X \vdash w \odot \left(f_* \underline{\mu} \right) = f_* \left((w \circ f) \odot \underline{\mu} \right).$$

When calculating in this notation, we use the equations in Figure 4.2 which contains a toolbox that includes most of the equations we come to expect from standard measure theory,

$$\begin{array}{lll}
\underline{\mu} : \mathrm{M}X, a : Y^X \vdash \oint_X \underline{\delta}_{a(x)} \underline{\mu}(\mathrm{d}x) = a_* \underline{\mu} & & f : Y^X \vdash f_* \underline{\delta}_x = \underline{\delta}_{f(x)} \\
x_0 : X, f : (\mathrm{M}Y)^X \vdash \oint_X f(x) \underline{\delta}_{x_0}(\mathrm{d}x) = f(x_0) & & \underline{\mu} : \mathrm{M}X \vdash \oint_X \underline{\delta}_x \underline{\mu}(\mathrm{d}x) = \underline{\mu} \\
x_0 : X, f : |A|^X \vdash \mathbb{E}_{x \sim \underline{\delta}_{x_0}}[f(x)] = f(x_0) & & \underline{\mu} : \mathrm{M}X \vdash \underline{\mu}(X) = \oint_X 1 \underline{\mu}(\mathrm{d}x) \\
r : \mathrm{M}\mathbb{1}, \underline{\mu} : \mathrm{M}X \vdash \oint_{\mathbb{1}} \underline{\mu}\, r(\mathrm{d}\xi) = r \odot \underline{\mu} & &
\end{array}$$

$$\begin{array}{ll}
\underline{\mu} : \mathrm{M}X, a : Y^X, f : (\mathrm{M}Z)^Y \vdash \oint_X (f \circ a)(x) \underline{\mu}(\mathrm{d}x) & = \oint_Y f(y)(a_* \underline{\mu})(\mathrm{d}y) \\
x_0 : X, w : R^X, f : (TY)^X \vdash \oint_X f(x)(w \odot \underline{\delta}_{x_0})(\mathrm{d}x) & = w(x_0) \odot f(x_0) \\
\underline{\mu} : \mathrm{M}X, w : R^X, f : (TY)^X \vdash \oint_X f(x)(w \odot \underline{\mu})(\mathrm{d}x) & = \oint_X w(x) \odot f(x) \underline{\mu}(\mathrm{d}x) \\
\underline{\mu} : \mathrm{M}X, f : (\mathrm{M}Y)^X \vdash \mathbb{E}_{x \sim \underline{\mu}}^{(KY, >\!>=)}[f(x)] & = \oint_X f(x) \underline{\mu}(\mathrm{d}x) \\
\underline{\mu} : \mathrm{M}X, a : Y^X, f : |A|^Y \vdash \mathbb{E}_{y \sim a_* \underline{\mu}}[f(y)] & = \mathbb{E}_{x \sim \underline{\mu}}[(f \circ a)(x)]
\end{array}$$

$$\begin{array}{ll}
\underline{\mu} : \mathrm{M}(X_1 \times X_2), f : (\mathrm{M}Y)^{X_i} \vdash \oiint_{X_1 \times X_2} f(x) \underline{\mu}(\mathrm{d}x, \mathrm{d}y) & = \oint_X f(x)((\pi_i)_* \underline{\mu})(\mathrm{d}x) \\
\underline{\mu} : \mathrm{M}X, f : (\mathrm{M}Y)^X, g : (\mathrm{M}Z)^Y \vdash \oint_X \underline{\mu}(\mathrm{d}x) \oint_Y g(y) f(x)(\mathrm{d}y) & = \oint_Y g(y) \left(\oint_X f(x) \underline{\mu}(\mathrm{d}x) \right) (\mathrm{d}y) \\
\underline{\mu} : \mathrm{M}X, \underline{\nu} : \mathrm{M}Y, g : (\mathrm{M}Z)^Y \vdash \oint_Y (\underline{\mu} \otimes g(y)) \underline{\nu}(\mathrm{d}y) & = \underline{\mu} \otimes \oint_Y g(y) \underline{\nu}(\mathrm{d}y)
\end{array}$$

$\boxed{\Gamma \coloneqq \underline{\mu}_1 : X_1, \underline{\mu}_2 : X_2}, i = 1, 2 :$

$$\begin{array}{ll}
\Gamma \vdash (\pi_i)_*(\underline{\mu}_1 \otimes \underline{\mu}_2) = \underline{\mu}_{3-i}(X_{3-i}) \odot \underline{\mu}_i & \oint_X \underline{\mu}_1\, \underline{\mu}_2(\mathrm{d}x) = \underline{\mu}_2(X_2) \odot \underline{\mu}_1 \\
\Gamma, f : (\mathrm{M}Y)^{X_i} \vdash \oiint_{X_1 \times X_2} f(x_i)(\underline{\mu}_1 \otimes \underline{\mu}_2)(\mathrm{d}x_1, \mathrm{d}x_2) & = \underline{\mu}_{3-i}(X_{3-i}) \odot \oint_{X_i} f(x_i) \underline{\mu}_i(\mathrm{d}x_i) \\
\Gamma, f^1 : Y_1^{X_1}, f^2 : Y_2^{X_2} \vdash (f^1 \times f^2)_*(\underline{\mu}_1 \otimes \underline{\mu}_2) & = (f^1_* \underline{\mu}_1) \otimes (f^2_* \underline{\mu}_2) \\
\oint_{X_1 \times X_2} \left(f^1(x_1) \otimes f^2(x_2) \right) (\underline{\mu}_1 \otimes \underline{\mu}_2)(\mathrm{d}x_1, \mathrm{d}x_2) & = \left(\oint_{X_1} f^1(x_1) \underline{\mu}_1(\mathrm{d}x_1) \right) \otimes \left(\oint_{X_2} f^2(x_2) \underline{\mu}_2(\mathrm{d}x_2) \right)
\end{array}$$

Figure 4.2: Toolbox for synthetic measure theory

like the change of variables law. To validate them, inline the definitions and proceed using the usual category-theoretic properties.

The following two sections contain relevant extensions to Kock's theory.

4.2.3 Radon-Nikodym Derivatives

The Radon-Nikodym Theorem is a powerful tool in measure theory, and we now phrase a synthetic counterpart. As usual in the synthetic setting, we set the definitions up such that the theorem will be true. Doing so highlights the difference between three measure-theoretic concepts that coincide in measure theory, but may differ in the synthetic setting.

Let $\underline{\mu}, \underline{\nu} \in \mathrm{M}X$ be measures. We say that $\underline{\nu}$ is *absolutely continuous* with respect to $\underline{\mu}$, and write $\underline{\nu} \ll \underline{\mu}$, when there exists a morphism $w : X \to R$ such that $\underline{\nu} = w \odot \underline{\mu}$. Given two

morphisms $w, v : X \to R$ and a measure $\underline{\mu} \in \mathrm{M}X$, we say that w and v are *equal* $\underline{\mu}$*-almost everywhere* ($\underline{\mu}$-a.e.) when $w \odot \underline{\mu} = v \odot \underline{\mu}$. A *measurable property* over X is a morphism $P : X \to \mathsf{bool}$. Given a measure $\underline{\mu} \in \mathrm{M}X$ a measurable property P over X *holds* $\underline{\mu}$*-a.e.*, when the morphism $[P] \coloneqq \lambda x.\ \mathbf{if}\ P\ x\ \mathbf{then}\ 1\ \mathbf{else}\ 0$ is equal $\underline{\mu}$-a.e. to 1.

Theorem 10 (Radon-Nikodym)**.** *Let* $(\mathscr{C}, \mathrm{M})$ *be a well-pointed measure category. For every* $\underline{\nu} \ll \underline{\mu}$ *in* $\mathrm{M}X$*, there exists a* $\underline{\mu}$*-a.e. unique morphism* $\frac{\mathrm{d}\underline{\nu}}{\mathrm{d}\underline{\mu}} : X \to R$ *satisfying* $\frac{\mathrm{d}\underline{\nu}}{\mathrm{d}\underline{\mu}} \odot \underline{\mu} = \underline{\nu}$.

4.2.4 Kernels

We say that a kernel $k : X \to \mathrm{M}Y$ is *Markov* when, for all x, $k(x, Y) = 1$, i.e., when k factors through the object of probability measures via $\mathsf{sub} : \mathrm{P} \rightarrowtail \mathrm{M}$. We now restrict attention to kernels $k : X \to \mathrm{M}X$ over the same object X. We say that such a kernel *preserves* a measure $\underline{\mu}$ when $\underline{\mu} \gg\!= k = \underline{\mu}$. Recall the morphism $\mathsf{swap} \coloneqq \lambda(x, y).\,(y, x) : X \times Y \to Y \times X$. Given a measure $\underline{\mu} \in \mathrm{M}X$ and a kernel k, we define the *box product* by $\underline{\mu} \boxtimes k \coloneqq \iint_{X \times X} \underline{\delta}_{(x,y)} \underline{\mu}(\mathrm{d}x) k(x, \mathrm{d}y)$. A kernel k is *reversible* with respect to a measure $\underline{\mu} \in \mathrm{M}X$ when $\mathsf{swap}_*(\underline{\mu} \boxtimes k) = \underline{\mu} \boxtimes k$.

The following standard results on kernels transfer into the synthetic setting. If a *Markov* kernel k is reversible with respect to $\underline{\mu}$, then k preserves $\underline{\mu}$. Kernels obtained by rescaling the Dirac kernel, i.e., $\lambda x.\, w(x) \odot \underline{\delta}_x$ are reversible w.r.t. all measures. Finally, linear combinations $\lambda x.\, \sum_{n \in \mathbb{N}} r_n \odot k_n(x)$ of reversible kernels w.r.t. $\underline{\mu}$ are also reversible w.r.t. $\underline{\mu}$.

4.3 Quasi-Borel Spaces

It remains to show that there is a concrete model of synthetic measure theory that contains the classical measure theoretic ideas that are central to probability theory and inference. This is novel because Kock's work [48] is targeted at the geometric/topological setting, whereas probability theory is based around Borel sets rather than open sets. It is non-trivial because the traditional setting for measure theory does not support higher-order functions [5] and commutativity of integration is subtle in general. In this section we resolve these problems by combining some recent discoveries [38, 83], and exhibit a model of synthetic measure theory which contains classical measure theory, for instance:

- the σ-semiring over the morphisms $\mathbb{1} \to R$ is isomorphic to the usual σ-semiring over the extended non-negative reals, $\overline{\mathbb{R}}_+$;
- this isomorphism induces a bijective correspondence between the morphisms $R \to \mathbb{1} + \mathbb{1}$ and the Borel subsets of $\overline{\mathbb{R}}_+$, as characteristic functions, and also between the morphisms $R \to R$ and the measurable functions $\overline{\mathbb{R}}_+ \to \overline{\mathbb{R}}_+$;

- it also induces an injection of the morphisms $\mathbb{1} \to \mathrm{M}(R)$ into the set of Borel measures on $\overline{\mathbb{R}}_+$, whose image contains all the probability measures; the morphisms $R \to \mathrm{M}(R)$ include all the Borel probability kernels;
- the canonical morphism $R^R \times \mathrm{M}(R) \to R$, $(f, \underline{\mu}) \mapsto \int f(x)\, \underline{\mu}(\mathrm{d}x)$, corresponds to classical Lebesgue integration.

Moreover, each object X can be seen as a set $U(X) = \mathscr{C}(\mathbb{1}, X)$ with structure, because the category is well-pointed, in the sense that the morphisms $X \to Y$ are a subset of the functions $U(X) \to U(Y)$.

4.3.1 Rudiments of Classical Measure Theory

Measurable spaces are the cornerstone of conventional measure theory, supporting a notion of measure.

Recall that a σ-*algebra* on a set X is a set Σ_X of subsets of X that is closed under countable unions and complements. A *measurable space* is a set together with a σ-algebra. A *measure* is a σ-additive function $\Sigma_X \to \overline{\mathbb{R}}_+$. A function f between measurable spaces is *measurable* if the inverse image of every measurable set according to f is measurable.

For example, on a Euclidean space $\overline{\mathbb{R}}^n$ we can consider the Borel sets, which form the smallest σ-algebra containing the open cubes. There is a canonical measure on $\overline{\mathbb{R}}^n$, the *Lebesgue measure*, which assigns to each cube its volume, and thus to every measurable function $f\colon \overline{\mathbb{R}}^n \to \overline{\mathbb{R}}_+$ a Lebesgue integral $\int_{\overline{\mathbb{R}}^n} f \in \overline{\mathbb{R}}_+$. A slightly more general class of measures is the σ-*finite measures*, which include the Lebesgue measures and are closed under disjoint unions and product measures.

A measurable space that arises from the Borel sets of a Polish space is called a *standard Borel space*. In fact, every standard Borel space is either countable or isomorphic to $\overline{\mathbb{R}}$. Standard Borel spaces are closed under countable products and countable disjoint unions.

4.3.2 Quasi-Borel Spaces

In this section we fix an uncountable standard Borel space, $\mathfrak{R}$. For example, $\mathfrak{R} = \overline{\mathbb{R}}$. The basic idea of quasi-Borel spaces is that rather than focusing on measurable sets of a set X, as in classical measure theory, one should focus on the admissible random elements $\mathfrak{R} \to X$.

Definition 11 ([38])**.** *A* quasi-Borel space (QBS) *is a set X together with a set of functions $M_X \subseteq [\mathfrak{R}, X]$ such that (i) all the constant functions are in M_X, (ii) M_X is closed under precomposition with measurable functions on $\mathfrak{R}$, and (iii) M_X satisfies the piecewise condition:*

if $\Re = \biguplus_{i=1}^{\infty} U_i$*, where* U_i *is Borel measurable and* $\alpha_i \in M_X$ *for all* i*, then* $\biguplus_{i=1}^{\infty} \alpha_i \cap (U_i \times X)$ *is in* M_X*.*

A morphism $f : X \to Y$ *is a function that respects the structure, i.e. if* $\alpha \in M_X$ *then* $(f \circ \alpha) \in M_Y$*. Morphisms compose as functions, and we have a category* **QBS**.

A QBS X *is a* subspace *of a QBS* Y *if* $X \subseteq Y$ *and* $M_X = \{\alpha : \Re \to X \mid \alpha \in M_Y\}$.

A measurable space X can be turned into a QBS when given the set of measurable functions $\Re \to X$ as M_X. When X and Y are standard Borel spaces considered as QBSes this way, $\mathbf{QBS}(X,Y)$ comprises the measurable functions, so **QBS** can be thought of as a conservative extension of the universe of standard Borel spaces. The three conditions on quasi-Borel spaces ensure that coproducts and products of standard Borel spaces retain their universal properties in **QBS**. In fact, the category of **QBS**s has all limits and colimits. It is also cartesian closed; e.g., $\mathbb{R}^{\mathbb{R}} \coloneqq \mathbf{QBS}(\mathbb{R},\mathbb{R})$, and $M_{(\mathbb{R}^{\mathbb{R}})} = \left\{\alpha : \Re \to (\mathbb{R}^{\mathbb{R}}) \;\middle|\; \mathsf{uncurry}(\alpha) \in \mathbf{QBS}(\Re \times \mathbb{R} \to \mathbb{R})\right\}$. For any QBS X, $M_X = \mathbf{QBS}(\Re, X)$.

4.3.3 A Monad of Measures

The following development is novel.

Definition 12. *A* measure *on a quasi-Borel space is a triple* (Ω, α, μ) *where* Ω *is a standard Borel space,* $\alpha \in \mathbf{QBS}(\Omega, X)$*, and* μ *is a* σ*-finite measure on* Ω*.*

For example, Ω might be $\overline{\mathbb{R}}^n$ and μ might be the Lebesgue measure. A measure determines an integration operator: if $f \in \mathbf{QBS}(X, \overline{\mathbb{R}}_+)$ then define

$$\int f \,\mathrm{d}(\Omega, \alpha, \mu) \coloneqq \int_\Omega f(\alpha(x))\, \mu(\mathrm{d}x)$$

using Lebesgue integration according to μ. We say that two measures are equal, denoted $(\Omega, \alpha, \mu) \approx (\Omega', \alpha', \mu')$, if they determine the same integration operator. We write $[\Omega, \alpha, \mu]$ for an equivalence class of measures.

As an aside, we note that not every integration operator on $\overline{\mathbb{R}}$ in the classical sense is a measure in the sense of Def. 12, because we restrict to σ-finite μ. Technically, the only integration operators that arise in this way are those corresponding to s-finite measures. This is a class of measures that includes the probability measures, and which works well with iterated integration and probabilistic programming [83].

The measures up-to $\approx$ form a monad, as follows. First, the set of all measures $\mathrm{M}X$ forms a QBS by setting $M_{\mathrm{M}X} = \{\lambda r.\, [D_r, \alpha(r,-), \mu|_{D_r}] \mid \mu\ \sigma\text{-finite on } \Omega,\ D \subseteq \Re \times \Omega \text{measurable},\ \alpha \in \mathbf{QBS}(D,X)\}$, where $D_r = \{\omega \mid (r,\omega) \in D\}$. In consequence, when Ω' is a standard Borel space, for every morphism $f : \Omega' \to \mathrm{M}X$, there exist Ω, μ, $D \subseteq \Omega' \times \Omega$ and $\alpha \in \mathbf{QBS}(D,X)$ such that

$f(\omega') = [D_{\omega'}, \alpha(\omega', -), \mu|_{D_{\omega'}}]$. One intuition is that α is a partial function $\Omega' \times \Omega \to X$, with domain D.

The unit of the monad, return : $X \to \mathrm{M}X$, is $\mathrm{return}(x) \coloneqq [\mathbb{1}, \lambda\_.x, \delta_{()}]$ where $\delta_{()}$ is the Dirac measure on the one-point space $\mathbb{1}$. We often write $\underline{\delta}_x$ for $\mathrm{return}(x)$. The bind $\gg\!=$: $\mathrm{M}X \times \mathrm{M}Y^X \to \mathrm{M}Y$ is

$$[\Omega, \alpha, \mu] \gg\!= \varphi \;\coloneqq\; [D, \beta, (\mu \otimes \mu')|_D],$$

where $\varphi(\alpha(r)) = [D_r, \beta(r, -), \mu']$. Note that $(\varphi \circ \alpha) : \Omega \to \mathrm{M}X$ must be of this form because it is a morphism from a standard Borel space. The measure $\mu \otimes \mu'$ is the product measure, which exists because μ and μ' are σ-finite.

This structure satisfies the monad laws, it is commutative by the Fubini-Tonelli theorem, and it satisfies the biproduct axioms, and so it is a model of synthetic measure theory. Every measure on $\mathbb{1}$ is equivalent to one of the form $([0, r], !, \mu)$ where $r \in \overline{\mathbb{R}}_+$, $! : [0, r] \to \mathbb{1}$ is the unique such random element, and μ is the Lebesgue measure. Thus $\mathrm{M}\mathbb{1} \cong \overline{\mathbb{R}}_+$.

As another aside, we note that when Ω, Ω' are standard Borel spaces, the Kleisli morphisms $\Omega \to \mathrm{M}\Omega'$ correspond to s-finite kernels, which were shown in [83] to provide a fully complete model of first-order probabilistic programming.

Chapter 5

Continuous Inference Semantics

In this chapter we provide the main theoretical contribution of the dissertation, which is a denotational semantics for probabilistic programs with continuous distributions corresponding to popular Monte Carlo algorithms. The construction is analogous to the discrete one presented in Chapter 3 and it uses the same calculus, extended with a primitive for sampling from a continuous distribution. The formal developments in this chapter heavily rely on the mathematical tools introduced in Chapter 4.

The crucial advantage of our semantics is that it corresponds to approximate inference algorithms that are commonly run in practice rather than measures corresponding to exact but intractable results. However, by relating the approximate semantics to the exact one based on measures we can state and prove correctness of the inference algorithms obtained in this fashion. Specifically, our formal notion of correctness corresponds to the sampler in question being unbiased.

Our semantics is compositional in a way that allows various components of Monte Carlo algorithms to be easily combined. Specifically, we have three different components of intermediate representations, two used in Sequential Monte Carlo methods and one in Metropolis-Hastings, and a collection of transformations corresponding to each component. These can be combined in various ways, for example to obtain an interpretation corresponding to the Resample-Move SMC algorithm. The semantic construction guarantees that all samplers obtained by such combinations are unbiased. Such compositions are discussed more thoroughly in Chapter 7.

5.1 Inference representations

We now develop the continuous counterpart to Sec. 3.6. The semantic structure of the category of quasi-Borel spaces allows us to transport many of the definitions with little change. For

example, a monadic interface $\underline{T}$ consists of analogous data, but the assignments are indexed by quasi-Borel spaces, T assigns quasi-Borel spaces, and $\mathrm{return}^{\underline{T}}$ and $\gg\!=^{\underline{T}}$ assign quasi-Borel space morphisms.

Definition 13. *A* continuous representation $\underline{T}$ *is a tuple* $(T, \mathrm{return}^{\underline{T}}, \gg\!=^{\underline{T}}, m^{\underline{T}})$ *consisting of:*

- *a monadic interface* $(T, \mathrm{return}^{\underline{T}}, \gg\!=^{\underline{T}})$*;*
- *an assignment of a* meaning *morphism* $m^{\underline{T}}_X : TX \to \mathrm{M}X$ *for every space* X

such that $m^{\underline{T}}$ *preserves* $\mathrm{return}^{\underline{T}}$ *and* $\gg\!=^{\underline{T}}$*.*

A sampling *representation is a tuple* $(T, \mathrm{return}^{\underline{T}}, \gg\!=^{\underline{T}}, m^{\underline{T}}, \mathbf{sample}^{\underline{T}})$ *such that its first four components form a continuous representation, it has an additional* **Qbs***-morphism* $\mathbf{sample}^{\underline{T}} : \mathbb{1} \to T\,\mathbb{I}$*, and* $m^{\underline{T}}$ *maps* $\mathbf{sample}^{\underline{T}}()$ *to the uniform* **Qbs***-measure* $\mathbf{U} = [\mathbb{I}, \mathrm{id}, \mathsf{Uniform}]$ *on the unit interval* $\mathbb{I}$*, where* Uniform *is the usual uniform distribution on* $\mathbb{I}$*.*

A conditioning *representation* $\underline{T}$ *is similarly a tuple* $(T, \mathrm{return}^{\underline{T}}, \gg\!=^{\underline{T}}, \mathrm{score}^{\underline{T}}, m^{\underline{T}})$*, with a* **Qbs***-morphism* $\mathrm{score}^{\underline{T}} : \mathbb{R}_+ \to T\,\mathbb{1}$ *such that for each* r*,* $m^{\underline{T}}$ *maps* $\mathrm{score}^{\underline{T}}(r)$ *to the* r*-rescaled unit* **Qbs***-measure* $r \odot \underline{\delta}_{()} = [\mathbb{1}, \lambda\_.\,(), r \cdot \delta_{()}]$*.*

An inference *representation* $\underline{T}$ *is a tuple* $(T, \mathrm{return}^{\underline{T}}, \gg\!=^{\underline{T}}, \mathbf{sample}^{\underline{T}}, \mathrm{score}^{\underline{T}}, m^{\underline{T}})$ *with the appropriate components forming both a sampling representation and a conditioning representation.*

This definition refines Definition 4 with sampling and scoring representations, allowing us to talk about inference transformers that augment a representation of one kind into another.

Example 12 (Continuous sampler)**.** By analogy with Example 7, we define in Fig. 5.1a a sampling representation using the type $\mathsf{Sam}\,\alpha := \{\mathsf{Return}\,\alpha \mid \mathsf{Sample}\,(\mathbb{I} \to \mathsf{Sam}\,\alpha)\}$. Validating the preservation of sample and the monadic interface is straightforward. It also follows from more general principles: $\underline{\mathsf{Sam}}$ is the initial monad with an operation $\mathbf{sample} : T\,\mathbb{I}$. □

We define inference transformations between any two representations as in Definition 5. We have four kinds of representations, and when defining transformers we can augment a representation with additional capabilities:

Definition 14. *Let* k_1*,* k_2 *be a pair of kinds of representation. A* k_1 to k_2 transformer $\underline{F}$ *is a tuple* $(F, \mathsf{tmap}^{\underline{F}}, \mathsf{lift}^{\underline{F}})$ *consisting of an assignments of:*

- *a* k_2 *representation* $F\,\underline{T}$ *to every* k_1 *representation* $\underline{T}$*;*
- *an inference transformation* $\mathsf{tmap}^{\underline{F}}\,\underline{t} : \underline{FT} \to \underline{FS}$ *to every transformation* $\underline{t} : \underline{T} \to \underline{S}$*; and*
- *an inference transformation* $\mathsf{lift}_T : \underline{T} \to F\,\underline{T}$ *to every* k_1 *representation* $\underline{T}$*.*

$$
\begin{aligned}
&\textbf{instance}\ \textit{Sampling Monad}\,(\mathsf{Sam})\ \textbf{where}\\
&\quad \textbf{return}\, x = \mathsf{Return}\, x\\
&\quad a >\!\!>\!= f\ = \textbf{match}\, a\, \textbf{with}\, \{\\
&\qquad\qquad \mathsf{Return}\, x \to f(x)\\
&\qquad\qquad \mathsf{Sample}\, k \to\\
&\qquad\qquad\quad \mathsf{Sample}\,(\lambda r.\, k(r) >\!\!>\!= f)\}\\
&\quad \mathsf{sample}\ = \mathsf{Sample}\, \lambda r.\, (\mathsf{Return}\, r)\\
&\quad m\, a\ \ \ = \textbf{match}\, a\, \textbf{with}\, \{\\
&\qquad\qquad \mathsf{Return}\, x \to \underline{\delta}_x\\
&\qquad\qquad \mathsf{Sample}\, k \to \int\!\!\!\!\!\!\int_{\mathbb{I}} k(x)\mathbf{U}(\mathrm{d}x)\}
\end{aligned}
$$

(a) Continuous sampler representation

$$
\begin{aligned}
&\textbf{instance}\ \textit{Cond Trans}\,(\mathsf{W})\ \textbf{where}\\
&\quad \textbf{return}_{\mathsf{W}\underline{T}}\, x = \mathrm{return}^{\underline{T}}(1,x)\\
&\quad a >\!\!>\!=_{\mathsf{W}\underline{T}} f = \underline{T}.\textbf{do}\,\{(r,x) \leftarrow a;\\
&\qquad\qquad\qquad\qquad (s,y) \leftarrow f(x);\\
&\qquad\qquad\qquad\qquad \textbf{return}(r\cdot s, y)\}\\
&\quad (\textbf{tmap}\, \underline{t})_X\ \ = \underline{t}_{\mathbb{R}_+ * X}\\
&\quad \textbf{lift}_{\underline{T}}\, a\ \ \ = \underline{T}.\textbf{do}\,\{x \leftarrow a; \textbf{return}(1,x)\}\\
&\quad m_{\mathsf{W}\underline{T}}\, a\ \ \ = \lambda x.\ \int\!\!\!\!\!\!\int_{\mathbb{R}_+ \times X} r \odot \underline{\delta}_x m^{\underline{T}}(a)(\mathrm{d}r, \mathrm{d}x)\\
&\quad \textbf{score}_{\mathsf{W}\underline{T}}\, r\ \ = \mathrm{return}^{\underline{T}}(r, ())
\end{aligned}
$$

(b) Continuous weighting inference transformer

Figure 5.1: Continuous representations and tranformers

When the two kinds k_1, k_2 differ, we say that that the transformer is augmenting.

When defining a k_1 to k_2 transformer, we adopt a Haskell-like type-class constraint notation $k_1 \implies k_2$ used for example in Fig. 5.4a.

Example 13. By analogy with Example 11, Fig. 5.1b presents the *continuous weighting* transformer structure on $\mathsf{W}\underline{T}\,X \coloneqq T(\mathbb{R}_+ * X)$. It augments any representation transformer with conditioning capabilities. Each conditioning operation is deferred to the return value, and so we can view this transformer as freely adding a conditioning operation that commutes with all other operations. When the starting representation had conditioning capabilities, we have an inference transformation $\mathrm{waggr} : \mathsf{W}\underline{T} \to \underline{T}$, given by $\mathrm{waggr}\, a \coloneqq \underline{T}.\textbf{do}\,\{(r,x) \leftarrow a; \mathrm{score}^{\underline{T}}\, r; \textbf{return}\, x\}$ which conditions based on the aggregated weight.

Its validity follows from a straightforward calculation using the meaning preservation of $\underline{T}$. □

In the continuous case, the output of the final inference transformation will always be $\mathsf{WSam}\, X$ or a similar $\mathsf{PopSam}\, X$ described in the next section. From this representation, we obtain the Monte Carlo approximation to the posterior by using a random number generator to supply the values required by Sam. Interpreting the program directly in $\mathsf{WSam}\, X$ and sampling from that would correspond to simple importance sampling from the prior, which usually needs a very large number of samples to give a good approximation to the posterior. Our goal in approximate Bayesian inference is therefore to find another representation for the program and a sequence of inference transformations that map it to $\mathsf{WSam}\, X$. While, in principle, this output represents the same posterior distribution, hopefully it uses a representation that requires

fewer samples to obtain a good approximation than a direct interpretation in $\mathsf{WSam}\,X$. We emphasise that approximation is only done in this final sampling step, while all the inference transformations that happen before it are exact.

5.2 Population

Given a representation $\underline{T}$, we define a representation structure over $\mathsf{Pop}\,\underline{T}\,X \coloneqq T(\mathsf{List}(\mathbb{R}_+ * X))$. We further deconstruct this representation transformer as the composition of two transformers: the continuous weighting transformer W from Example 13, and Haskell's notorious ListT transformer.

The negative reputation associated to the transformer $\mathsf{ListT}\,\underline{T}\,X \coloneqq T(\mathsf{List}\,X)$ stems from its failure to validate the monad laws when $\underline{T}$ is not commutative.[1] However, it is a perfectly valid representation transformer, described in Fig. 5.2a, since we do not require that representations satisfy monad laws.

To prove the meaning function preserves return, simply calculate. For $\gg\!=$ preservation, show that

$$a_s : \mathsf{List}(T\,X) \vdash m^{\mathsf{ListT}\,\underline{T}}(\mathrm{sequence}\,a_s) = \sum_{a \in a_s} m^{\underline{T}}(a)$$

and proceed via straightforward calculation using the linearity of the Kock integral and the commutative (σ-)monoid structure on measures.

By composing the two representation transformers, we obtain the representation to conditioning transformer Pop, given explicitly in Fig. 5.2b.

Fig. 5.2c presents a $\mathbb{N}_+$-indexed family of inference transformations. Fix any $n \in \mathbb{N}$. The spark function generates a population of particles with the unit value, and the same weight $\frac{1}{n}$. Thus, $\mathrm{spawn}\,(n,a)$ takes a distribution a over particle populations, sparks n equally weighted particles, and for each of them, samples a population based on a. A straightforward calculation confirms that the meaning of spark is 1, and so $\mathrm{spawn}\,(n,-) : \mathsf{Pop}\,\underline{T} \to \mathsf{Pop}\,\underline{T}$ is an inference transformation. In the version of SMC we consider below, we will only pass to spawn a distribution a over uniformly-weighted single-particle populations.

We use spawn to *resample* a new population. Thinking operationally, we have a population of weighted particles and we obtain a new population by sampling with replacement from the current one, where the probability of selecting a given particle is proportional to its weight. Doing so is equivalent to simulating a discrete weighted sample using a uniform one.

Lemma 15. *There is a* **Qbs**-*morphism* $\mathrm{dwrand} : \mathsf{List}(\mathbb{R}_+ * X) * \mathbb{I} \to \{\mathsf{Take}\,X \mid \mathsf{Fail}\}$ *such that:*

[1] For a list transformer "done right", see Jaskelioff's thesis [43], and its generalisations [72, 24].

Auxiliary functions:
$\text{sequence} : \mathsf{List}(TX) \to T(\mathsf{List}\,X)$
$\text{sequence} \coloneqq \mathsf{foldr}\,(\text{return}[\,])$
$\qquad (\lambda(a,r).\,\underline{T}.\mathbf{do}\,\{x \leftarrow a;$
$\qquad\qquad x_s \leftarrow r;$
$\qquad\qquad \mathbf{return}(x :: x_s)\}$
$\text{concat} : \mathsf{List}(\mathsf{List}\,X) \to \mathsf{List}\,X$
$\text{concat} \coloneqq \mathsf{foldr}\,[\,] +\!\!+$
$\sum_{x \in x_s} f(x) \coloneqq \mathsf{foldr}\,0\ \left(\lambda(x,s).\,f(x)+s\right)\ x_s$

instance *Rep Trans* (ListT) **where**
$\quad \mathbf{return}_{\mathsf{ListT}\,\underline{T}}\,x = \text{return}^{\underline{T}}[x]$
$\quad a \gg\!=_{\mathsf{ListT}\,\underline{T}}\,f\ = \underline{T}.\mathbf{do}\,\{x_s \leftarrow a;$
$\qquad\qquad\qquad\qquad \mathbf{let}\,b_s = \text{map}\,f\,x_s\,\mathbf{in}$
$\qquad\qquad\qquad\qquad y_{ss} \leftarrow \text{sequence}\,b_s;$
$\qquad\qquad\qquad\qquad \mathbf{return}(\text{concat}\,y_{ss})\}$
$\quad m_{\mathsf{ListT}\,\underline{T}\,a} \quad = \oint_{\mathsf{List}\,X} m^{\underline{T}}(a)(\mathrm{d}x_s) \sum_{x \in x_s} \underline{\delta}_x$
$\quad \mathbf{lift}_{\mathsf{ListT}\,\underline{T}\,a} \quad = \underline{T}.\mathbf{do}\,\{x \leftarrow a; \mathbf{return}\,[x]\}$
$\quad (\mathbf{tmap}\,\underline{t})_X \quad = t_{\mathsf{List}\,X}$

(a) The list transformer

instance *Cond Trans* (Pop) **where**
$\quad \mathbf{return}_{\mathsf{Pop}\,\underline{T}} = \mathbf{return}_{(\mathsf{W}\circ\mathsf{ListT})\underline{T}}$
$\quad \gg\!=_{\mathsf{Pop}\,\underline{T}} \quad = \gg\!=_{(\mathsf{W}\circ\mathsf{ListT})\underline{T}}$
$\quad \mathbf{lift}_{\mathsf{Pop}\,\underline{T}} \quad = \mathbf{lift}_{\mathsf{W}(\mathsf{ListT}\,\underline{T})} \circ \mathbf{lift}_{\mathsf{ListT}\,\underline{T}}$
$\quad \mathbf{tmap}_{\mathsf{Pop}\,\underline{T}} \quad = \mathbf{tmap}_{\mathsf{W}(\mathsf{ListT}\,\underline{T})} \circ \mathbf{tmap}_{\mathsf{ListT}\,\underline{T}}$
$\quad m_{\mathsf{Pop}\,\underline{T}} \quad = m_{(\mathsf{W}\circ\mathsf{ListT})\underline{T}}$
$\qquad\qquad = \lambda a.\ \oint_{\mathsf{List}(\mathbb{R}_+ \times X)} m^T(a)(\mathrm{d}x_s) \sum_{(r,x) \in x_s} r \odot \underline{\delta}_x$
$\quad \mathbf{score}_{\mathsf{Pop}\,\underline{T}} \quad = \mathbf{score}_{(\mathsf{W}\circ\mathsf{ListT})\underline{T}}$

(b) The population transformer

$\text{replicate} : \mathbb{N} * X \to \mathsf{List}\,X$
$\text{replicate}(\mathsf{Zero}, x) = [\,]$
$\text{replicate}(\mathsf{Succ}\,n, x) = x :: \text{replicate}(n,x)$
$\text{spark} : \mathbb{N}_+ \to \mathsf{Pop}\,\underline{T}\,\mathbf{1}$
$\text{spark} \coloneqq \text{return}^{\underline{T}}\left(\text{replicate}(n, (\tfrac{1}{n}, ()))\right)$
$\text{spawn} : \mathbb{N}_+ * \mathsf{Pop}\,\underline{T}\,X \to \mathsf{Pop}\,\underline{T}\,X$
$\text{spawn}(n,a) \coloneqq \mathsf{Pop}\,\underline{T}.\mathbf{do}\,\{\text{spark}\,n; a\}$

(c) Spawning new particles

Figure 5.2: Representing populations

```
dwrand(x_s, r) :=
  let w = Σ_{(r,_)∈x_s} r in
  if    w = 0
  then Fail
  else foldr (w · r, Fail)
            (λ((s,x),(fuel,result)).
            if   0 ≤ fuel < s
            then(−1, Take x)
            else −− potential underflow
                 (fuel − s, result))
            x_s
```

(a) A discrete weighted randomiser

```
dwsample^T(x_s) :=
  T.do {score(Σ_{(r,_)∈x_s} r);
        r ← sample;
        match dwrand (x_s, r) with {
              Fail   →fail
            | Take x→return x}}
```

(b) A discrete weighted sampler

```
resample : ℕ+ ∗ Pop T X → Pop T X
resample(n, a) :=
    T.do {x_s ← a;
          spawn(n, dwsample^{Pop T} x_s)}
```

(c) Resampling

Figure 5.3: The resampling transformation

- *For all x_s for which $\sum_{(r,\_)\in x_s} r = 0$, we have* $\mathrm{dwrand}(x_s,-)_*\mathbf{U} = \underline{\delta}_{\mathsf{Fail}}$.
- *For all x_s for which $w := \sum_{(r,\_)\in x_s} r > 0$, we have* $\mathrm{dwrand}(x_s,-)_*\mathbf{U} = \sum_{(r,x)\in x_s} \frac{r_i}{w} \odot \underline{\delta}_{\mathsf{Take}\,x}$.

Fig. 5.3a presents one such morphism, though its precise implementation does not matter to our development. As a consequence, for every sampling representation $\underline{T}$ for which we have an element $\mathrm{fail} : TX$ such that $m^{\underline{T}}(\mathrm{fail}) = 0$, we can define a discrete weighted sampler $\mathrm{dwsample}^{\underline{T}}(x_s) : \mathsf{List}(\mathbb{R}_+ X) \to TX$ in Fig. 5.3b which will then satisfy $m^{\underline{T}}(\mathrm{dwsample}^{\underline{T}}(x_s)) = \sum_{(r,x)\in x_s} r \odot \underline{\delta}_x$.

The resampling step in Fig. 5.3c operationally takes the current population, creates a computation/thunk that samples a single particle from this population, and then spawns n new particles that are initialised with this thunk. The morphism $\mathrm{resample}\,(n,-) : \mathsf{Pop}\,\underline{T} \to \mathsf{Pop}\,\underline{T}$ is an inference transformation because, as we know, $\mathrm{spawn}\,(n,-)$ is one and $\mathrm{dwsample}^{\mathsf{Pop}\,\underline{T}} : \mathsf{Pop}\,\underline{T} \to \mathsf{Pop}\,\underline{T}$ samples a population consisting of just a single unit weight particle with a probability proportional to its renormalised weight in the original population.

5.3 Sequential

The second transformer in the SMC algorithm allows us to suspend computation after each conditioning. The suspension transformer equips the standard *resumption* monad transformer

$$
\begin{aligned}
&\textbf{instance } \mathit{Cond} \implies \mathit{Cond}\ \mathit{Trans}\,(\mathsf{Sus})\ \textbf{where}\\
&\quad \textbf{return}_{\mathsf{Sus}\,\underline{T}}\,x \;= \mathrm{return}_{\underline{T}}(\mathsf{Return}\,x)\\
&\quad a \gg\!=_{\mathsf{Sus}\,\underline{T}}\, f \;= \textbf{fold}\,(\lambda b.\,\underline{T}.\textbf{do}\,\{\\
&\qquad\qquad t \leftarrow b;\\
&\qquad\qquad \textbf{match}\,t\ \textbf{with}\,\{\\
&\qquad\qquad\qquad \mathsf{Return}\,x \to f(x)\\
&\qquad\qquad\qquad \mid \mathsf{Yield}\,c \to \mathsf{Yield}\,c\})\}\\
&\qquad\qquad a\\
&\quad \textbf{lift}_{\mathsf{Sus}\,\underline{T}}\,a \;= \underline{T}.\textbf{do}\,\{x \leftarrow a; \mathrm{return}_{\mathsf{Sus}\,\underline{T}}\,x\}\\
&\quad (\textbf{tmap}_{\mathsf{Sus}\,\underline{T}}\,t)_X = \mathsf{Sus}\,\underline{T}X.\textbf{fold}\,(\lambda b.\, m_{\underline{S}}(b))\\
&\quad m_{\mathsf{Sus}\,\underline{T}}\,a \;= m_{\underline{T}}(\mathrm{finish}_{\mathsf{Sus}\,\underline{T}}(a))\\
&\quad \textbf{score}\,r \;= \mathrm{return}_{\underline{T}}(\mathsf{Yield}\ \mathrm{lift}_{\mathsf{Sus}\,\underline{T}}(\textbf{score}\,r))
\end{aligned}
$$

(a) The suspension transformer

$$
\begin{aligned}
&\mathrm{advance}_{\underline{T}} : \mathsf{Sus}\,\underline{T}X \to \mathsf{Sus}\,\underline{T}X\\
&\mathrm{advance}_{\underline{T}}\,a = \underline{T}.\textbf{do}\,\{\\
&\qquad t \leftarrow a;\\
&\qquad \textbf{match}\,t\ \textbf{with}\,\{\\
&\qquad\qquad \mathsf{Return}\,x \to \mathrm{return}_{\underline{T}}\,x\\
&\qquad\qquad \mid \mathsf{Yield}\,t \to t\}\}\\
&\mathrm{finish}_{\underline{T}} : \mathsf{Sus}\,\underline{T}X \to \underline{T}X\\
&\mathrm{finish}_{\underline{T}}\,a = \textbf{fold}\,\lambda b.\,\underline{T}.\textbf{do}\,\{\\
&\qquad t \leftarrow b;\\
&\qquad \textbf{match}\,t\ \textbf{with}\,\{\\
&\qquad\qquad \mathsf{Return}\,x \to \mathrm{return}\,x\\
&\qquad\qquad \mid \mathsf{Yield}\,b \to b\}\}
\end{aligned}
$$

(b) Suspension operations

Figure 5.4: The suspension transformation

$\mathsf{Sus}\,\underline{T}X \coloneqq T\{\mathsf{Return}\,X \mid \mathsf{Yield}\,(\mathsf{Sus}\,\underline{T}X)\}$, presented in Fig. 5.4a, with inference transformations.

The two transformations on suspended computations in Fig. 5.4b take one step, and complete the computation, accordingly. As the meaning function for the transformed representation returns the meaning the computation would have if it was allowed to run to completion, these two operations do not change the meaning and so form inference transformations.

We can now put all the components together:

Theorem 16. *Let $\underline{T}$ be a sampling representation. For every pair of natural numbers n, k, the following composite forms an inference transformation:*

$$
\begin{aligned}
\mathrm{smc}^{\underline{T}}_{n,k} \coloneqq (\mathsf{Sus}\circ\mathsf{Pop})\underline{T} &\xrightarrow{\textbf{tmap}_{\mathsf{Sus}}\,\mathrm{spawn}(n,-)} (\mathsf{Sus}\circ\mathsf{Pop})\underline{T}\\
&\xrightarrow{(\mathrm{advance}\circ\textbf{tmap}_{\mathsf{Sus}}\,\mathrm{resample}(n,-))^{\circ k}} (\mathsf{Sus}\circ\mathsf{Pop})\underline{T} \xrightarrow{\mathrm{finish}} \mathsf{Pop}\,\underline{T}.
\end{aligned}
$$

In the above $(-)^{\circ -} : X^X \times \mathbb{N} \to X^X$ denotes n-fold composition. The transformation $\mathrm{smc}^{\underline{T}}_{n,k}$ amounts to running the SMC algorithm with n particles for k steps. If the representation $\underline{T}$ is operational in nature, such as the continuous sampler Sam, we get a sequence of weighted values over the return type when we run the resulting representation. By construction, the distribution on the results, rescaled according to their final weights, would be identical to the desired posterior distribution.

When the representation $\underline{T}$ is not a commutative monad, like the continuous sampler Sam, the resulting representation $\mathsf{Pop}\,\underline{T}$ is not a monad: the monad laws do not hold. Therefore, to encompass representations of $\mathsf{Pop}\,\underline{T}$ one must generalise beyond monads.

5.4 Traced

Markov Chain Monte Carlo (MCMC) algorithms operate by repeatedly using a transition kernel to generate a new sample from a current one. Thus they can be thought of as performing a random walk around the space they are exploring. If the transition kernel is well-behaved, they are guaranteed to preserve the distribution. A popular MCMC algorithm used for Bayesian inference is Metropolis-Hastings (MH), where the transition kernel consists of a proposal kernel followed by a decision to either accept the proposed sample or keep the old one. The accept or reject step is used to correct for bias introduced by the proposal kernel, thus producing a valid MCMC algorithm for a rich family of proposal kernels.

MH is a general inference method, but it requires specialised knowledge about the space on which they operate. In the context of a probabilistic programming language, the *Trace MH* algorithm replaces the unknown target space with the space of program *traces*, which are shared by all probabilistic programs. Thus, Trace MH allows probabilistic programming language designers to devise general-purpose kernels to effectively explore traces.

We analyse the Trace MH as follows. First, we prove a quasi-Borel space counterpart of the *Metropolis-Hastings-Green (MHG) Theorem*, that forms the theoretical foundation for the correctness of MH. We then present the *tracing* representation and show its validity. We present the Trace MH algorithm, parameterised by a proposal kernel for traces, and give sufficient conditions on this kernel for the resulting transformation to be valid. We then give a concrete proposal kernel and show that it satisfies these general conditions.

5.4.1 Abstract Metropolis-Hastings-Green

In the abstract, the key ingredient in MH is the *Metropolis-Hastings-Green (MHG)* morphism η presented in Fig. 5.5a, formulated in terms of an arbitrary inference representation $\underline{T}$. This transformation is usually known as the *update step* of the MH algorithm. It is parameterised by a (representation of a) *proposal* kernel $\psi : X \to T\,X$, and by a chosen (representation of a) Radon-Nikodym derivative $\rho : X \times X \to \overline{\mathbb{R}}_+$.

To use η in an inference transformation, we need to provide well-behaved parameters ψ, ρ, and their behaviour may depend on the representation of the input distribution a. In particular, the parameter ρ should represent a well-behaved appropriate Radon-Nikodym derivative. To

$$\psi : (TX)^X, \rho : \overline{\mathbb{R}}_+^{X\times X}, a : TX \vdash$$
$$\underline{T}.\mathbf{do}\,\{x \leftarrow a;$$
$$y \leftarrow \psi(x);$$
$$r \leftarrow \text{sample};$$
$$\mathbf{if}\ r < \min(1, \rho(x,y))$$
$$\mathbf{then\ return}\ y$$
$$\mathbf{else\ return}\ x\}$$

(a) The Metropolis-Hastings-Green morphism $\eta_{\psi,\rho}(a) : TX$

$$p \in t =$$
$$\mathbf{match}\,(p,t)\,\mathbf{with}\,\{([\,], \mathsf{Return}\,x) \to \mathsf{True}$$
$$(r :: r_s, \mathsf{Sample}\,f) \to [r_s \in f(r)]$$
$$-- \text{ any other case:}$$
$$(\_, \_) \to \mathsf{False}\}$$
$$w_- : \textstyle\sum_{t\in \mathsf{WSam}X} \mathsf{Paths}\,t \to \mathbb{R}_+ \qquad v_- : \sum_{t\in \mathsf{WSam}X} \mathsf{Paths}\,t \to X$$
$$w_{\mathsf{Return}\,(r,x)}([\,]) = r \qquad v_{\mathsf{Return}\,(r,x)}([\,]) = x$$
$$w_{\mathsf{Sample}\,t_-}(s :: r_s) = w_{t_s}(r_s) \qquad v_{\mathsf{Sample}\,t_-}(s :: r_s) = v_{t_s}(r_s)$$

(b) Traces through a probabilistic program

Figure 5.5: Basic notions in Trace MH

simplify our proofs, we also require that the proposal kernel ψ is Markov, which suffices for our application.

Theorem 17 (Metropolis-Hastings-Green)**.** *Let X be a QBS, $a \in TX$ a distribution, $\psi : X \to TX$ a kernel, and $\rho : X \times X \to \overline{\mathbb{R}}_+$ a* **Qbs***-morphism. Set $k := m^{\underline{T}} \circ \psi$ and $\underline{\mu} := [\rho \neq 0] \odot (m^{\underline{T}}(a) \boxtimes k)$.*

Assume that: 1. k is Markov; 2. $[1 = (\rho \circ \mathsf{swap}) \cdot \rho]$ holds $\underline{\mu}$-a.e.; 3. ρ is a Radon-Nikodym derivative of $\mathsf{swap}_*\underline{\mu}$ with respect to $\underline{\mu}$; and 4. $\rho(x,y) = 0 \iff \rho(y,x) = 0$ for all $x, y \in X$.

Then $(m^{\underline{T}} \circ \eta_{\psi,\rho})(a) = m^{\underline{T}}(a)$.

Using Kock's synthetic measure theory, we were able to follow closely standard measure-theoretic proofs of MHG [28]. The synthetic setting highlights the different roles each of the three abstractions: a.e.-equality, a.e.-properties, and Radon-Nykodim derivatives play in the proof that our formulation exposes (cf. § 4.2.3).

5.4.2 Tracing Representation

A sampling *trace* is a sequence of samples that occur during the execution of a probabilistic program. We represent such programs as elements of the continuous weighted sampler WSam from (cf. Fig. 5.1). Consequently, the collection of traces through a program $t \in \mathsf{WSam}X$ is a subset of $\mathsf{List}\,\mathbb{I}$. Fig. 5.5b defines a measurable predicate $[\in] : \mathsf{WSam}X \times \mathsf{List}\,\mathbb{I} \to \mathsf{bool}$ that tests whether a given sequence p of probabilistic choice forms a complete trace in the program t. Consequently, we can define the set of *paths* through a given program t by $\mathsf{Paths}\,t := \{p \in \mathsf{List}\,\mathbb{I} \mid p \in t\} \subseteq \mathsf{List}\,\mathbb{I}$, and equip it with the subspace structure it inherits from $\mathsf{List}\,\mathbb{I}$. We can therefore define the set

$$\sum_{t\in \mathsf{WSam}X} \mathsf{Paths}\,t := \left\{(t,p) \in \mathsf{WSam}X \times \mathsf{List}\,\mathbb{I} \,\middle|\, p \in t\right\} \subseteq \mathsf{WSam}X \times \mathsf{List}\,\mathbb{I},$$

$$
\begin{aligned}
&\textbf{instance } \mathit{Inf} \implies \mathit{Inf}\ \mathit{Monad}\,(\mathrm{Tr}\,\underline{T})\ \textbf{where}\\
&\quad \textbf{return}\,x \qquad\qquad = (\mathrm{return}_{\mathsf{WSam}}\,x, \mathrm{return}_{\underline{T}}[\,])\\
&\quad (t,a) >\!\!>\!= (f,g) =\\
&\qquad (t >\!\!>\!=_{\mathsf{WSam}}\ f, \underline{T}.\textbf{do}\,\{p \leftarrow a;\\
&\qquad\qquad\qquad\qquad q \leftarrow g \circ v_t(p);\\
&\qquad\qquad\qquad\qquad \textbf{return}(p +\!\!+ q)\})\\
&\quad m(t,a) \qquad\qquad = m_{\mathsf{WSam}}(t) = \oint_{\mathsf{List}\,\mathbb{I}} \underline{\delta}_{v_t(p)}\, m_{\underline{T}}(a)(\mathrm{d}p)\\
&\quad \textbf{tmap}\,\underline{t} \qquad\qquad = \mathrm{id} \times \underline{t}_{\mathsf{List}\,\mathbb{I}}\\
&\quad \mathrm{sample} \qquad\qquad = (\textbf{sample}_{\mathsf{WSam}},\\
&\qquad\qquad\qquad \underline{T}.\textbf{do}\,\{r \leftarrow \mathrm{sample}; \textbf{return}[r]\})\\
&\quad \textbf{score}\,r \qquad\qquad = (\textbf{score}_{\mathsf{WSam}},\\
&\qquad\qquad\qquad \underline{T}.\textbf{do}\,\{\textbf{score}\,r; \textbf{return}[\,]\})
\end{aligned}
$$

(a) The tracing inference

$$
\begin{aligned}
&\eta^{\mathrm{Tr}\,T}_{\psi,\rho} : \mathrm{Tr}\,T\,X \to \mathrm{Tr}\,T\,X\\
&\eta^{\mathrm{Tr}\,T}_{\psi,\rho}\,(t,a) \coloneqq \big(t, \eta_{\psi_t,\rho_t}(a)\big)
\end{aligned}
$$

(b) Trace MH update-step

$$
\begin{aligned}
&\mathrm{pri}_{\underline{T}} : \mathsf{WSam}\,X \to T(\mathsf{List}(\mathbb{I}))\\
&\mathrm{pri}_{\underline{T}}(t) \coloneqq \textbf{fold}\\
&\lambda\{\mathsf{Return}\,(r,x) \to \mathrm{return}_{\underline{T}}[\,]\\
&\quad \big|\,\mathsf{Sample}\,k \quad \to \underline{T}.\textbf{do}\,\{\\
&\qquad\qquad\qquad r \leftarrow \textbf{sample}_{\underline{T}};\\
&\qquad\qquad\qquad k(r)\}\}
\end{aligned}
$$

(c) Prior representation

Figure 5.6: Building blocks of Trace MH

which we can also equip with a subspace structure. We can now define the *weight* w_- and *valuation* v_- morphisms in Fig. 5.5b that retrieve the likelihood and value at the end of a trace.

We can now define the tracing inference representation. It is parameterised by an inference representation $\underline{T}$ and given for X as the following subspace of $\mathsf{WSam}\,X \times T(\mathsf{List}\,\mathbb{I})$:

$$
\mathrm{Tr}\,\underline{T}\,X \coloneqq \left\{ (t,a) \in \mathsf{WSam}\,X \times T(\mathsf{List}\,\mathbb{I}) \;\middle|\; \begin{array}{l} \oint_{\mathsf{List}\,\mathbb{I}} \underline{\delta}_{x \in t}\, m_{\underline{T}}(a)(\mathrm{d}x) = \underline{\delta}_{\mathsf{True}} \\ m_{\mathsf{WSam}}(t) = \oint_{\mathsf{List}\,\mathbb{I}} \underline{\delta}_{v_t(p)}\, m_{\underline{T}}(a)(\mathrm{d}p) \end{array} \right\}.
$$

Thus, a representation consists of a program representation t, together with a distribution a on all lists, but maintaining two invariants. First, the lists are $m_{\underline{T}}(a)$-almost-everywhere paths through t, and so we can indeed think of a as a representation of a distribution over traces. Second, if we calculate the posterior of the paths through t according to $m_{\underline{T}}(a)$, it should have the same meaning as the original program.

We stress that an implementation need *not* compute the meaning of the program. But this representation guarantees that the meaning will be preserved by the inference operations.

Note that the integrand in the definition of $(t,a) \in \mathrm{Tr}\,T\,X$ is only partially defined. This partiality is not an issue because the first condition guarantees it is $m^T(a)$-a.e. defined. We can then choose the constantly 0 distribution when $p \notin t$.

Fig. 5.6a presents the inference representation structure of $\mathrm{Tr}\,\underline{T}$. Most of the proof revolves around preserving the invariant, i.e., that these definitions define set-theoretic functions.

The inference transformation $\text{marginal}_{\underline{T}} : \mathrm{Tr}\,\underline{T}X \to \underline{T}X$ *marginalises* the trace transformer once it is no longer useful. It first samples a path and then uses it to run the program discarding the weight: $\text{marginal}\,(t,a) = \mathbf{do}\,\{x \leftarrow a; \mathbf{return}\ v_t(x)\}$. Its correctness is precisely the invariant.

5.4.3 Inference with MHG

The transition from $\underline{T}$ to $\mathrm{Tr}\,\underline{T}$ still requires a proposal kernel and a representation of the appropriate derivative, but these can now be given in terms of concrete traces.

Given an inference representation $\underline{T}$, a *trace proposal kernel* is a transformation representing a kernel $\psi : (\sum_{t \in \mathsf{WSam}\,X} \mathrm{Paths}\,t) \to T(\mathsf{List}\,\mathbb{I})$. A *trace derivative* is a transformation representing the derivative $\rho : (\sum_{t \in \mathsf{WSam}\,X} \mathrm{Paths}\,t \times \mathrm{Paths}\,t) \to \overline{\mathbb{R}}_+$. Given a trace proposal kernel ψ and a trace derivative ρ, Fig. 5.6b presents the trace MHG update transformation using the corresponding MHG update on $T(\mathsf{List}\,\mathbb{I})$.

The Trace MH update step requires some assumptions to form an inference transformation:

Theorem 18 (Trace Metropolis-Hastings-Green)**.** *Let T be an inference representation, ψ a trace proposal kernel, and ρ a trace derivative. Assume that, for every $(t,a) \in \mathrm{Tr}\,T\,X$, letting $k := m^T \circ \psi_t$ and $\underline{\mu} := [\rho_t \neq 0] \odot (m^T(a) \boxtimes k)$:*

1. *k is Markov;*
2. *$[1 = \rho_t \cdot (\rho_t \circ \mathsf{swap})]$ holds $\underline{\mu}$-a.e.;*
3. *ρ_t is a Radon-Nikodym derivative of $\mathsf{swap}_* \underline{\mu}$ with respect to $\underline{\mu}$; and*
4. *$\rho_t(p,q) = 0 \iff \rho_t(q,p) = 0$ for all $p,q \in \mathsf{List}(\mathbb{I})$.*

Then $\eta^{\mathrm{Tr}\,T}_{\psi,\rho} : \mathrm{Tr}\,T \to \mathrm{Tr}\,T$ is a valid inference transformation.

We will now demonstrate such a simple and generic trace proposal kernel and trace derivative that implement a MHG update step of a popular lightweight Metropolis-Hastings algorithm in several probabilistic programming language systems [31, 40, 92, 32].

For any inference representation $\underline{T}$, Fig. 5.6c defines the morphism $\text{pri}_{\underline{T}}$ that maps a representation $t \in \mathsf{WSam}\,X$ to its prior distribution on paths over t. Let $\mathrm{U_D}(n) \in \mathrm{M}(\mathbb{N})$ be the measure for the uniform discrete distribution with support $\{0,1,\ldots,n\}$. Intuitively, it assigns a probability $\frac{1}{n+1}$ to every element in the support. It can be easily defined from $\mathbf{sample}_{\mathrm{M}}$, which denotes the uniform distribution on $\mathbb{I}$, as in Lemma 15.

We now define our concrete proposal ψ_t and derivative, a.k.a. ratio, ρ_t:

$$\begin{aligned}
&\psi_t : \mathsf{List}(\mathbb{I}) \to T(\mathsf{List}(\mathbb{I})) \\
&\psi_t(p) \coloneqq \underline{T}.\mathbf{do}\,\{i \leftarrow \mathrm{U_D}^{\underline{T}}(|p|) \\
&\qquad\qquad q \leftarrow \mathrm{pri}^{T}(\mathrm{sub}(t, \mathrm{take}(i,p))) \\
&\qquad\qquad \mathbf{return}(\mathrm{take}(i,p) + q)\}
\end{aligned}
\qquad\qquad
\begin{aligned}
&\rho_t : \mathsf{List}(\mathbb{I}) \times \mathsf{List}(\mathbb{I}) \to \overline{\mathbb{R}}_+ \\
&\rho_t(p,q) \coloneqq \tfrac{w_t(q)\cdot(|p|+1)}{w_t(p)\cdot(|q|+1)}
\end{aligned}$$

where $\mathrm{sub}(t,x)$ selects a subterm of a given term by following the list x and $\mathrm{take}(i,p)$ retrieves the i-th prefix of p. This proposal and derivative/ratio satisfy the condition in the Trace MH.

Our approach lets us combine MH updates with other inference building blocks. For example, recall the SMC algorithm from Section 5.3. Each time it performs resampling, multiple particles are given the same values, which results in inadequate coverage of the space, a phenomenon known as *degeneracy*. One way to ameliorate this problem is to apply multiple MH transitions to each particle after resampling in order to spread them across the space, resulting in an algorithm known as resample-move SMC [19].

The implementation of resample-move SMC is very similar to that of SMC from Section 5.3, except we introduce an additional layer Tr between Sus and Pop:

Theorem 19. *Let $\underline{T}$ be a sampling representation. For every pair of natural numbers n, k, ℓ the following composite forms an inference transformation:*

$$\begin{aligned}
\mathrm{rmsmc}^{\underline{T}}_{n,k,\ell} \coloneqq (\mathsf{Sus}\circ\mathsf{Tr}\circ\mathsf{Pop})\underline{T} &\xrightarrow{\mathbf{tmap}_{\mathsf{Sus}}\,\mathbf{tmap}_{\mathrm{Tr}}\,\mathrm{spawn}(n,-)} (\mathsf{Sus}\circ\mathsf{Tr}\circ\mathsf{Pop})\underline{T} \\
&\xrightarrow{(\mathrm{advance}\circ\ \mathbf{tmap}_{\mathsf{Sus}}\,\eta^{\circ\ell}\ \circ\mathbf{tmap}_{\mathsf{Sus}}\,\mathbf{tmap}_{\mathrm{Tr}}\,\mathrm{resample}(n,-))^{\circ k}} (\mathsf{Sus}\circ\mathsf{Tr}\circ\mathsf{Pop})\underline{T} \xrightarrow{\mathrm{marginal}\circ\mathrm{finish}} \mathsf{Pop}\,\underline{T}.
\end{aligned}$$

In the above we apply ℓ MH transitions after each resampling. Our compositional correctness criterion corresponds to a known result that resample-move SMC is an unbiased importance sampler.

Chapter 6

Implementation of Inference Building Blocks

We now describe an implementation of a Haskell library for probabilistic programming based on the semantic construction presented in the chapters above. In this chapter we show an implementation of the basic building blocks while in the subsequent one we show how to combine them to obtain compound inference algorithms. The work presented in these two chapters has been published as [79] and is a result of collaborations with the other authors of that paper. In order to make these chapters more accessible to readers less interested in formal semantics we describe the relevant concepts avoiding references to chapters 3 and 5 whenever possible. We apologise to readers who read the previous chapters and find the presentation here overly repetitive.

Our library provides a monadic typeclass with probabilistic effects that can be used to construct probabilistic programs using arbitrary pure Haskell code. The modelling language is thus expressive, constituting what is sometimes called a *universal* probabilistic programming language [31]. The main novelty of our library is its compositional approach to specifying Bayesian inference algorithms. The library is called MonadBayes and it is freely available online[1].

We prefix our development with two remarks for Bayesian inference cognoscenti. First, we focus on the most general family of inference algorithms, namely samplers that do not use *gradient* information. We ignore: gradient-based techniques such as Hamiltonian Monte Carlo, that assume the likelihood function is differentiable; optimisation based methods, including variational inference; and enumeration based methods such as belief propagation. While these

[1] https://github.com/adscib/monad-bayes

```
type R = Double
class Monad m ⇒ MonadSample m where
  random :: m R
  bernoulli :: R → m Bool
  bernoulli p = fmap (< p) random
  -- and other default distributions:
  -- normal, gamma, beta, geometric,
  -- poisson, dirichlet
class Monad m ⇒ MonadCond m where
  score :: Log R → m ()
class (MonadSample m, MonadCond m) ⇒ MonadInfer m
```

Figure 6.1: Inference representations using Haskell type-classes. `Log R` is a numeric type representing non-negative real numbers using their logarithms.

classes of algorithms are less generic than sampling, they are still important and we plan to develop them in future work.

Second, our library implements basic versions of advanced sampling algorithms. However, their successful application in practice requires incorporating established heuristics, such as: adaptive proposal distributions, controlling resampling with effective sample size, tuning rejuvenation kernels based on population in SMC^2, and so on. We believe these are largely orthogonal to the core design, so excluding them makes for a clearer and more accessible presentation of the main ideas.

6.1 Basic building blocks

We express the building blocks for inference algorithms in Haskell. Unless stated otherwise, the structures presented in this section follow the mathematical formulation from Chapter 5. We distinguish three types of building blocks:

1. **Inference representations**
   Inference representations are data structures representing distributions. Concretely, they are instances/implementations of the monad type-class/interface, but they need *not* satisfy the monad laws. We use them as the intermediate representation in the inference/compilation process.

   In Haskell, we express these abstract interfaces as type-classes, as in Figure 6.1. There are three separate type-classes: the *sampling* representation `MonadSample`, and the *conditioning* representation `MonadCond`. A representation that is both a sampling and conditioning representation is called an *inference* representation `MonadInfer`. Together with the `Monad`

interface, the interfaces in the figure can express our probabilistic programs of interest. Our library includes default implementations for common probability distributions in terms of `random`, which is a uniform distribution on the unit interval $[0,1]$. Specific representations can overwrite these default implementations for better efficiency.

To reason about correctness of algorithms involving a representation `m`, we define a semantics map μ `:: m a` $\to$ `D a` assigning to each representation `c :: m a` a distribution over its return type μ_c `:: D a`. This "type" of distributions and the semantics map are mathematical objects impossible to implement. They are pure reasoning abstractions, and to ensure correctness, we require the semantics map preserves the monadic structure. Since `D` is itself a monad, both in terms of an interface and satisfying the monad laws, we require that the following equations hold.

$$\mu \texttt{ . return = return} \qquad\qquad \mu \texttt{ (c >>= f) = } \mu\texttt{(c) >>= (}\mu\texttt{. f)}$$

Furthermore, we require that μ `(random)` is the uniform distribution over the unit interval $[0,1]$, and that μ `(score r)` is the (unique) distribution over the singleton with total measure `r`. The details are presented in Chapter 5.

2. **Inference transformations**
   Inference transformations are mappings `t :: m a` $\to$ `m' a` between inference representations `m`, `m'`. They can be thought of as passes in a compilation process. For reasoning purposes, such an inference transformation is *correct* if it preserves the semantics map of the transformed inference representation: μ `. t` $= \mu$. It follows that a composition of correct inference transformations is a correct inference transformation, so algorithms given by composition of correct inference transformations are correct by construction. Unless stated otherwise, all inference transformations are correct based on the reasoning in Chapter 5. As a consequence, the weighted samplers our inference algorithms produce are *unbiased* by construction. Being unbiased means that the weighted expectation of the sampler is identical to the expectation of the model.

3. **Inference transformers**
   Inference transformers are compositional building blocks of inference representations, much like monad transformers are building blocks of monads. By analogy with monad stacks we call a sequence of inference transformers an *inference stack* and implicitly identify a stack `[`$\texttt{mT}_1$`, `$\texttt{mT}_2$`, ..., `$\texttt{mT}_n$`]` with the composed transformer $\texttt{mT}_1$ `.` $\texttt{mT}_2$ `. ... .` $\texttt{mT}_n$. Our inference transformations often apply to a specific transformer in the stack, and are polymorphic in the remainder of the stack. Like monad transformers, inference transformers specify a function to lift a transformation through them, which we call `hoist`

since `lift` is already used in Haskell. These abstractions allow us to apply inference transformations to particular layers in the inference stack.

This chapter uses these abstractions to define basic inference building blocks. The following Chapter 7 shows how to compose them to obtain advanced inference algorithms.

6.1.1 Models

The following example program implements a simple random walk using the abstractions introduced above. It models a particle travelling in one dimension in discrete time. At each time step the particle moves randomly according to a Gaussian distribution. The model takes as argument a data-set containing noisy observations about the particle's location. The model captures our hypothesis about the particle's movement, subject to some unknown parameters such as the rate in which it is moving. The goal of inference is to fit this model to the observed data, updating these parameters accordingly.

```
1  random_walk :: MonadInfer m ⇒ [R] → m [R]
2  random_walk ys = do
3    s ← gamma 1 1
4    let expand xs [] = return xs
5        expand (x:xs) (y:ys) = do
6          x' ← normal x s
7          obs x' y
8          expand (x':x:xs) ys
9        where
10         obs x y = score (normalPdf x 1 y)
11   xs ← expand [0] ys
12   return (reverse xs)
```

The *parameter* `s` controls the deviation of the particle from its current position. We emphasise that the parameter in this case is the sampling operation in the dynamic call to `gamma` on line 3, rather than the static program variable `s`. The model starts at position `0`, and iteratively constructs the desired list of locations in reverse. At each step, we sample the next location `x'` from a normal distribution around `x` with a standard deviation of `s` (line 6). This sample is then fitted to the observed data point `y`, using the helper function `obs` from line 10. It uses the probability density function (pdf) for the normal distribution

$$\texttt{normalPdf}\ \mu\ \sigma\ z := \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(z-\mu)^2}{2\sigma^2}}$$

centered around `x` with standard deviation `1`, representing our assumption that `y` is a noisy observation of `x`, where the noise distributes normally with standard deviation `1`. The call

`obs x' y` on line 7 thus lowers the score of samples `x'` that are far from the observed location `y`. In standard statistical modelling terminology the calls to `normal x s` on line 6 are called *latent variables*. The output of the model is the predicted sequence of true positions of the particle based on provided noisy observations. It is reversed only because Haskell lists are more easily extended at the front than at the back so `expand` constructs this list in the reverse order.

The type of `random_walk` is abstract with respect to the inference representation `m`. Thus we express models as computations constructed in terms of an abstract inference representation interface. This architecture lends itself to a shallowly embedded probabilistic programming DSL, with the usual benefits of DSLs, e.g., the ability to call standard library functions such as `reverse`. We could alternatively use a stand-alone language for expressing models that a suitable front-end would convert to a desired inference representation.

6.1.2 Basic Samplers

To interpret the program above we need to construct a concrete inference representation. The simplest one is a sampler that draws concrete values for random variables from the prior. Such a sampler type can be constructed as a state monad that references a global pseudo-random number generator. Since the generator is mutable, in Haskell we need to use the `ST` monad.

```
1 newtype Sampler a =
2   Sampler (forall s. ReaderT (GenST s) (ST s) a)
3 instance MonadSample Sampler where
4   random = Sampler $ do
5     gen ← ask
6     lift (MWC.uniform gen)
```

Line 5 retrieves the reference to the random seed, line 6 uses the standard library `MWC` for random number generation, and lifts it to the underlying reader monad transformer.

We can directly execute computations of type `Sampler` to obtain concrete samples, for example:

```
runSampler :: Sampler a → a
runSampler (Sampler s) = runST $ do
  gen ← MWC.create
  runReaderT s gen
```

The represenation `Sampler` is a sampling representation, i.e., an instance of `MonadSample`, but not a conditioning representation, i.e., an instance of `MonadCond`: `Sampler` does not support conditioning. To obtain a working inference algorithm we need to add an interpretation of `score`, by applying a suitable inference transformer.

We use the *weighting* inference transformer `W`. Theoretically it is the writer monad transformer for the multiplicative monoid structure on `Log R` given by multiplication, but in our implementation we are using the state monad transformer instead since we found it to be much faster. In either case, `W m a` is an `m`-computation returning pairs `(a, Log R)` of the result type together with the accumulated log-likelihood.

```
newtype W m a = W (StateT (Log R) m a)
runW :: Monad m ⇒ W m a → m (a, Log R)
runW (W x) = runStateT x 1
instance MonadSample m ⇒ MonadSample (W m) where
  random = lift . random
instance Monad m ⇒ MonadCond (W m) where
  score w = W (modify (* w))
hoistW :: (forall x. m x → n x) → W m a → W n a
hoistW f (W m) = W (mapStateT f m)
```

Weighting a representation `m` equips it with conditioning operation making `W m` a conditioning representation. If `m` was a sampling representation, its weighted version is also a sampling representation by lifting the sampling operation. Finally, the function `hoist` lifts inference transformations applicable to `m` and turns them into inference transformations applicable to `T m`. When `m` is already a conditioning representation, we may use the conditioning available through the weighting transformer, or hoist `score r` to use the ambient conditioning of `m`.

We construct a simple inference algorithm by interpreting the model in `W Sampler a` and unwrapping type constructors to obtain a *weighted sampler* of type `Sampler (a, Log R)`. When `a` is a numeric type, we can approximate the expectation of the model by repeatedly running the sampler and calculating the weighted average. Unfortunately such an algorithm usually has low statistical efficiency and it needs to generate impractically many samples to obtain good predictions. Therefore, advanced inference algorithms first apply multiple inference transformations between more advanced inference representations before arriving at the final weighted sampler representation.

6.1.3 Population

The `Pop` inference transformer turns a single sample into a collection of weighted samples called the *population*. It is the weighted list transformer, i.e., the composition of `W` with the `ListT` transformer.

```
newtype Pop m a = Pop (W (ListT m) a)
  deriving(Monad,MonadSample,MonadCond,MonadInfer)
runPopulation :: Monad m ⇒ Pop m a → m [a, Log R]
runPopulation (Pop p) = runListT (runW p)
hoistP :: (forall x. m x → n x) → Pop m a → Pop n a
hoistP f (Pop m) = Pop (hoistW (mapListT f) m)
```

The samples in the population are usually referred to as *particles*. The usual problem with treating `ListT` as a monad transformer, namely that `ListT`-transformed monads do not always satisfy the monad laws, does not apply to inference representation: we do not require that our representations satisfy the monad laws.

In this dissertation we use three inference transformations associated with `Pop`:

```
spawn    :: Monad        m ⇒ Int     → Pop m ()
resample :: MonadSample m ⇒ Pop m a → Pop m a
pushEvidence :: MonadInfer m ⇒ Pop m a → Pop m a
```

One is `(spawn n >>)`, which increases the population size `n` times adjusting the weights accordingly. Next comes `resample`, which draws a new population with uniform weights from the current population. Resampling's purpose is to remedy situations when a single sample has a large weight compared to the other particles in the population and dominates the result making the other particles irrelevant. Finally, we have `pushEvidence` which normalises the weights in the population, while at the same time incorporating the sum of the weights as a `score` in `m`.

The meaning function μ is defined in terms of the weighted average over the population. We can state it concisely by making use of `pushEvidence` and a categorical distribution.

$$\mu^{\texttt{Pop m}}(\texttt{p}) = \mu^{\texttt{m}}(\texttt{runPopulation (pushEvidence p) >>= categorical})$$

In the above `categorical` is a distribution that draws a sample from a weighted list with probabilities proportional to weights.

Since all of the presented transformations preserve the weighted average μ, they are correct inference transformations. The details of `resample` may vary since there are multiple good choices. Our library uses systematic resampling [19] due to its good computational efficiency.

By itself `Pop` is similar to `W`. To appreciate its utility we need to combine it with different inference transformers, where it abstracts away the maintenance of the particle population.

6.2 Advanced building blocks

Sequential Monte Carlo (SMC) and Markov Chain Monte Carlo (MCMC) are two of the most general inference algorithms for probabilistic programs. When we express them as inference

representations and reuse the basic building blocks of the previous section, we obtain the components that underlie many advanced inference algorithms.

6.2.1 Sequential

Many models exhibit a sequential structure where observations are interleaved with sampling. In those models a possible inference strategy is to consider a program up to a certain point, do inference on the partial posterior it defines, then run the program a little more, do more inference, and so on. To implement such algorithms we introduce the sequential transformer `Seq` which introduces suspensions after each `score` in the program. `Seq` is the standard coroutine transformer [8]. In our library we use the implementation of the coroutine transformer available in the monad-coroutine library[2] but the snippet below shows how to implement it from scratch.

```
data Seq m a = Seq {runSeq :: m (Either a (Seq m a))}

instance Monad m ⇒ Monad (Seq m) where
  return x = Seq (return (Left x))
  Seq c >>= f = Seq $ do
    t ← c
    case t of
      Left x → runSeq (f x)
      Right m → return (Right (m >>= f))
instance MonadTrans Seq where
  lift = Seq . fmap Left

suspend :: Monad m ⇒ Seq m ()
suspend = Seq (return (Right (return ())))

instance MonadSample m ⇒ MonadSample (Seq m) where
  random  = lift random
instance MonadCond   m ⇒ MonadCond   (Seq m) where
  score w = lift (score w) >> suspend
```

We have two inference transformations associated with `Seq`:

```
advance :: Monad m ⇒ Seq m a → Seq m a
advance (Seq m) = Seq (m >>= either (return . Left) runSeq)
finish :: Monad m ⇒ Seq m a → m a
finish  (Seq m) = Seq (m >>= either return finish)
```

[2] http://hackage.haskell.org/package/monad-coroutine

The `advance` transformation runs the program to the next suspension point. The `finish` transformation runs the program to the end. When reasoning about `Seq`, the meaning function is:

$$\mu^{\texttt{Seq m}}(\texttt{c}) := \mu^{\texttt{m}}(\texttt{finish c})$$

Finally, `hoistS` applies the inference transformation only to the part of the program executed so far.

```
hoistS :: (forall x. m x → m x) → Seq m a → Seq m a
hoistS tau (Seq m) = Seq (tau m)
```

Combining `Seq` with `Pop`, we obtain a Sequential Monte Carlo variant known as the *particle filter* [19] that we refer to simply as SMC. Within the context of SMC, recall that a sample in a population is called a *particle*. The algorithm starts by initialising a population of size `n`, then repeatedly runs the program to the next `score`, resamples the population, runs to the next `score` and so on. We implement it by composing the inference transformations we have introduced so far.

```
smc :: MonadSample m ⇒ Int → Int → Seq (Pop m) a → Pop m a
smc k n = finish .
  compose k (advance . hoistS resample) . hoistS (spawn n >>)
```

The argument `k` is the number of time steps in SMC, `n` is the number of particles used, and

```
compose :: Int → (a → a) → a → a
```

is `k`-fold function composition. To execute the sampler, we use the instance where `m` is `Sampler`.

6.2.2 Traced

The final transformer we present supports a class of algorithms known as *Trace Markov Chain Monte Carlo (MCMC)*. This transformer provides the necessary machinery to perform Metropolis-Hastings (MH) updates described in Section 2.1.2, in particular by enabling calculation of the required densities. As these updates are performed on traces of programs, rather than just the outputs, this transformer also includes a tracing mechanism.

The idea behind MH is to represent a distribution over a space `a` as a simulation of a random walk through the space `a` according to some predefined *proposal kernel* `k :: a →m a`. If the simulation is currently at value `x`, the kernel `k` determines a distribution `k x` over the proposed values. The second fundamental part of MCMC is the *acceptance rate* ρ. This rate is a non-negative function ρ `: a →a →Log R`. At each step, given some `x :: a` sampled from a distribution `c :: m a`, we sample a new proposal `y∼k x` and, with probability ρ `x y`, we *accept*

the new proposal `y`, taking it as the new point, or *reject* it and remain with `x`. We can summarise this process in the following code:

```
abstractMH :: MonadSample m ⇒ m a → m a
abstractMH c = do
  x ← c
  y ← k x
  b ← bernoulli (ρ x y)
  if b then return y
       else return x
```

If the kernel `k` and the rejection rate ρ are correctly chosen, then `abstractMH` is a bona fide inference transformation. The mathematical justification for this fact is the Metropolis-Hastings-Green theorem stated in Section 5.4.

We might try to run MH directly on the space of the program outputs. Unfortunately this is not feasible since computing the acceptance ratio in that case requires an intractable marginalisation over all the random variables in the program. Instead, the Trace MCMC family of algorithms operates on traces through the probabilistic program. For a concrete example, consider the sprinkler model from the introduction. The trace would contain the values sampled for the variables `rain` and `sprinkler`. The Trace MCMC algorithm maintains a distribution over such traces, using a suitable proposal kernel to generate a new trace through the model.

Here, we take traces to be lists of real numbers `[R]` from the unit interval $[0, 1]$, each corresponding to one invocation of `random` in the program. For example, in the sprinkler model a trace `[0.15, 0.5]` would correspond to `rain = True` and `sprinkler = False`, while `[0.3, 0.05]` would correspond to `rain = False` and `sprinkler = True`.

If the variable names in the program are globally unique, we could have a record where each field corresponds to the random variable with the same name. However, in a general model, samples may be nested in complicated control flow. Wingate et al. [91] devised a popular method for tagging random choices in the program based on the context in which they are executed. On top of that, the random choices stored in the trace are often augmented with additional information such as the distribution they were drawn from or some control flow information extracted from the program [54]. These sophisticated representations of traces can improve the statistical efficiency of MCMC algorithms. They are compatible with the design we present in this dissertation but we refrain from using them for simplicity.

A traced inference representation consists of two components, one being the trace and the other a representation that can run the full program with a modified trace. There are multiple possible ways to combine these two components, which trade off computational efficiency for flexibility. Below we present a sequence of `Tr` datatypes, each more expressive than the previous in a sense of allowing additional inference transformations, but less computationally efficient

in cases where the additional flexibility is not required. The first one was is the construction presented in Chapter 5, while the subsequent ones are extensions of it. We expect the additional inference transformations associated with these extensions to be correct, although we have not proven that.

All of these variations share a common definition of a `Trace` data structure. Specifically, a `Trace` consists of a list of values for the latent variables `[R]`, the output value `a`, and the density `Log R`. It is equipped with methods for constructing basic traces and combining them within a monad.

```
data Trace a =
  Trace {
    variables :: [R],
    output :: a,
    density :: Log R
  }

pure :: a → Trace a
pure x = Trace {variables = [], output = x, density = 1}

singleton :: Double → Trace Double
singleton u = Trace {variables = [u], output = u, density = 1}

scored :: Log Double → Trace ()
scored w = Trace {variables = [], output = (), density = w}

bind :: Monad m ⇒ m (Trace a) → (a → m (Trace b)) → m (Trace b)
bind dx f = do
  t1 ← dx
  t2 ← f (output t1)
  return $ t2 {variables = variables t1 ++ variables t2,
               density = density t1 * density t2}
```

Full Tracing of the Whole Program

We begin with the most straightforward construction that allows our basic modular implementation [80] of the Trace Metropolis-Hastings algorithm. It consists of a weighted free monad over `random` and a computation generating a trace in the transformed inference representation. For efficiency we use the Church-encoded version of the free monad from the package free[3], that is `F f = forall r. (a → r) → (f r → r) → r`.

[3] http://hackage.haskell.org/package/free

```
-- sampling functor
newtype SamF a = Random (R → a)

data Tr m a = Tr (W (F SamF) a) (m (Trace a))
traceDist (Tr m d) = d

instance Monad m ⇒ Monad (Tr m) where
  return x = Tr (return x) (return (pure x))
  (Tr mx dx) >>= f = Tr my dy where
    my = mx >>= model . f
    dy = dx `bind` (traceDist . f)

instance MonadSample m ⇒ MonadSample (Tr m) where
  random = Tr random (fmap singleton random)

instance MonadCond m ⇒ MonadCond (Tr m) where
  score w = Tr (score w) (score w >> return (scored w))

hoistT :: (forall x. m x → m x) → Tr m a → Tr m a
hoistT f (Tr m d) = Tr m (f d)

marginal :: Monad m ⇒ Tr m a → m a
marginal (Tr m d) = fmap output d

mhStep :: MonadSample m ⇒ Tr m a → Tr m a
mh :: MonadSample m ⇒ Int → Tr m a → m [a]
```

The implementation of `mhStep` follows the structure of `abstractMH` above, suitably instantiated for traces. We emphasise that `Tr` is not an instance of `MonadTrans` since it does not allow for computation in `m` to be lifted to `Tr m`. For reasoning, the semantic function of `Tr` is defined in terms of `marginal`, which marginalises the trace and the model, leaving only the return value:

$$\mu^{\texttt{Tr m}}(\texttt{c}) := \mu^{\texttt{m}}(\texttt{marginal c})$$

The inference transformation `mhStep` performs a single step of the Trace MH algorithm updating the trace but leaving the program unchanged. Specifically, a new trace is proposed by taking the old trace and randomly modifying one of the random variables in it, selected again at random. Since the number of random variables used in the program can vary dynamically, the length of the new trace is adjusted to match the length required by the program. If the trace is too long it is truncated, if it is too short it is extended with freshly sampled values. This adjustment requires a pass through the program so at the same time we compute the likelihood

associated with the adjusted trace. Finally, based on the ratio of likelihoods, as well as some correcting factors [91], we compute the probability of accepting the new trace. With that probability we retain the proposed trace, otherwise we keep the old one. The details of this construction are given in Section 5.4.

Repeating this procedure multiple times defines a Markov process on the space of execution traces, which constitutes the Trace MH algorithm. It is available in our library as the `mh` inference transformation. However, the basic building block is `mhStep`, forming a component of larger inference algorithms such as the resample-move SMC in Section 7.1.

The efficiency of MH crucially depends on the choice of the proposal kernel. Our library uses as a default the single-site kernel sampling from the prior as proposed by Wingate et al. [91].

Partial Tracing of the Whole Program

The construction presented above is suitable if all random variables in the program are subject to the Trace MH updates. This is not the case in the family of inference algorithms known as pseudo-marginal MH, where only a subset of variables is updated using MH and the remaining ones are marginalised using another inference algorithm. This marginalisation is usually performed approximately using importance sampling. This is the case for all the algorithms we present in this dissertation, but the marginalisation could also be done with enumeration or a different inference algorithm.

To enable pseudo-marginal MH methods we extend the construction above by replacing the free monad with a free monad transformer applied to `m`. This transformer enables us to lift computations in `m` into `Tr m`, which is not possible with the previous construction. In Haskell this change means `Tr` becomes an instance of the `MonadTrans` class. The variables in computations lifted from `m` are then marginalised by `m` as far as Trace MH is concerned while the ones created in `Tr m` are subject to Trace MH updates. Since `Trace` caches the output and the density of the pseudo-marginal computation, we realise what Andrieu and Roberts [4] call the grouped independence MH which is known to be asymptotically correct. In Section 7.2 we present a concrete pseudo-marginal inference algorithm constructed in this fashion.

```
data Tr m a = Tr (W (FT SamF m) a) (m (Trace a))
instance MonadTrans Tr where
  lift m = Tr (lift $ lift m) (fmap pure m)
```

Note that we define `lift` in such a way that the lifted random variables are resampled at every `mhStep` proposal. If we only included `m` in the second component of `Tr` and not the first, they would be fixed throughout MH updates. While that is also potentially useful, this is not

what pseudo-marginal MH requires so we do not pursue this possibility here. We do not show code for the remaining instances and transformations since it is exactly the same as for the version above.

Partial Tracing of Program Fragments

In certain situations it is desirable to freeze the values of random variables using the contents of the current trace. This is useful when we know we will not update them any more but still want to keep the `Tr` structure for the random variables that come later. We present a concrete use case for this operation in Section 7.1.

```
data Tr m a = Tr (m (W (FT SamF m) a, (Trace a)))
runTr (Tr c) = c

instance Monad m ⇒ Monad (Tr m) where
  return x = Tr (return (return x, pure x))
  (Tr cx) >>= f = Tr $ do
    (mx, tx) ← cx
    let m = mx >>= pushM . fmap fst . runTraced . f
    t ← return tx `bind` (fmap snd . runTraced . f)
    return (m, t)

instance MonadTrans Tr where
  lift m = Tr $ fmap ((,) (lift $ lift m) . pure) m

instance MonadSample m ⇒ MonadSample (Tr m) where
  random = Tr $ fmap ((,) random . singleton) random

hoistT :: (forall x. m x → m x) → Tr m a → Tr m a
hoistT f (Tr c) = Tr (f c)

instance MonadCond m ⇒ MonadCond (Tr m) where
  score w = Tr $ fmap ((,) (score w)) (score w >> return (scored w))

marginal :: Monad m ⇒ Tr m a → m a
marginal = fmap (output . snd) . runTr

freeze :: Monad m ⇒ Tr m a → Tr m a
freeze (Tr c) = Tr $ do
  (_, t) ← c
  let x = output t
  return (return x, pure x)
```

The only difference from the previous construction is that we pushed the weighted free monad into `m`, which lets us implement the `freeze` inference transformation that commits to values stored in the current trace. This is useful if we later extend the program and do not want to update values for some variables anymore. We use it in Section 7.1 to obtain an efficient variant of resample-move SMC.

This implementation is strictly more expressive than the previous two so in principle it could be used instead of them. It is also significantly less computationally efficient due to the additional abstraction layers. We therefore prefer the previous constructions of `Tr` whenever possible.

6.3 Evaluation

To evaluate our architecture, we compare with state-of-the-art probabilistic programming systems Anglican [92] and WebPPL [32] since they implement similar inference algorithms and their front-end languages are extensions of popular programming languages, Clojure and Javascript respectively.

Benchmarks. To check if there is a significant overhead associated with the abstractions we compare execution times on a set of popular benchmarks [86]. The models we use are logistic regression (LR), hidden Markov model (HMM), and a latent Dirichlet allocation (LDA). Each of these models has a parameter that controls the size of the dataset, namely the number of labelled examples for LR, sequence length for the HMM, and document length for LDA. We run the inference algorithms SMC, MH, and RM-SMC from Section 6.1 and Chapter 7 on these models comparing execution times. For RM-SMC we use the `rmsmcLocal` and compare it with the corresponding WebPPL implementation. We do not compare Anglican here since it currently does not implement RM-SMC.

6.3.1 Quantitative Evaluation

Figure 6.2 shows how execution time scales with the size of the dataset. It shows that the cost of both MH and SMC increases linearly with model size in each implementation as expected. The plots show that Anglican and WebPPL have a noticeable starting overhead compared to our library. We expect this overhead to stem from just-in-time compilation in the Java and NodeJS virtual machines.

The slopes of each line are a measure of the time needed to incorporate an additional data point, with steeper slopes corresponding to higher cost. The slopes for different systems are

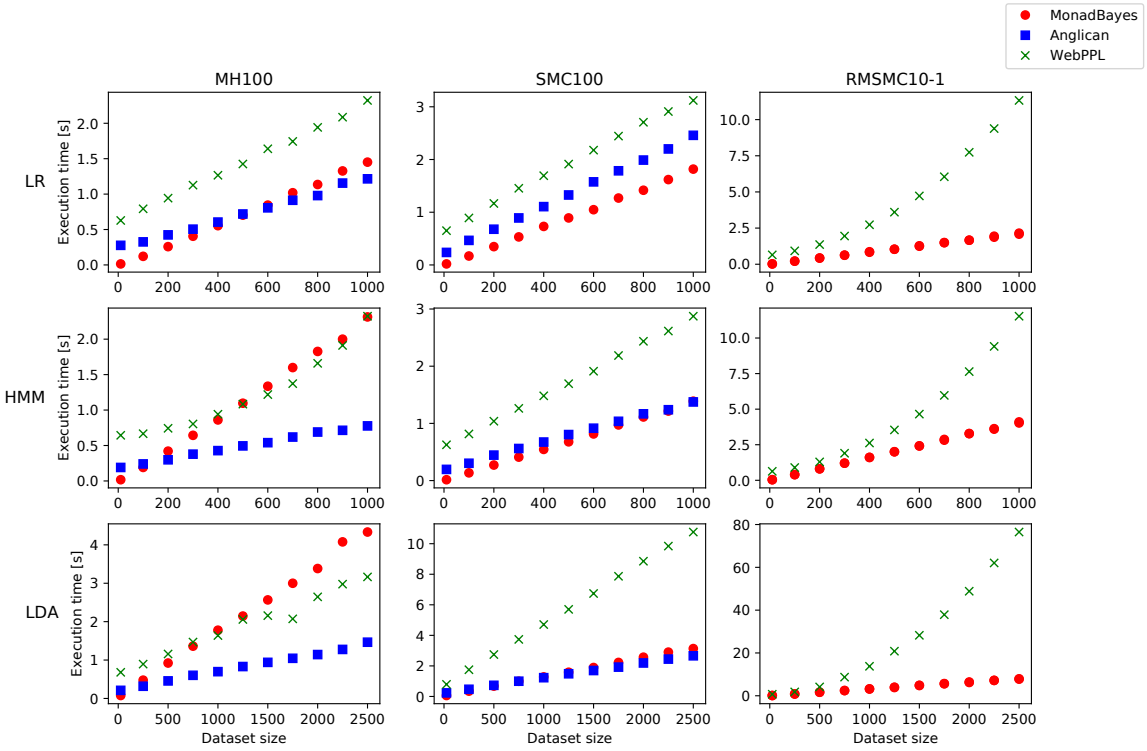


Figure 6.2: Execution times of inference algorithms with varying dataset size. The numbers in the algorithm description indicate the parameters used. For MH we used 100 transitions, for SMC 100 particles, and for RM-SMC 10 particles and 1 rejuvenation step per particle per resampling step. The dataset size is the number of observations in LR and HMM and the total number of words in all documents in LDA.

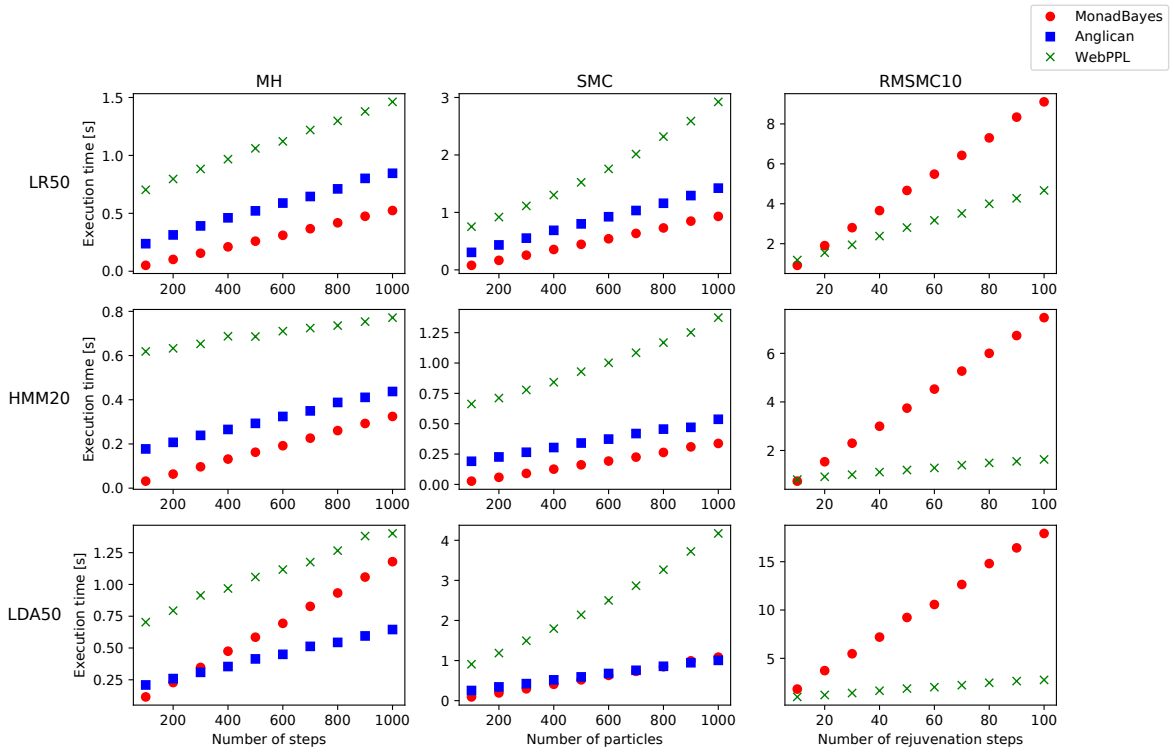


Figure 6.3: Execution times of inference algorithms with varying sample size. The numbers after model names are the sizes of the datasets used. The number 10 in RMSMC10 indicates that we used 10 particles and only varied the number of rejuvenation steps. The x axis for RMSMC shows the number of rejuvenation steps applied per particle after each resampling operation.

model-dependent and we attribute these differences to a variety of factors, such as differences in: the data structures used in the model; the pseudo-random number generators; and performance characteristics of the host language.

For RM-SMC our implementation scales linearly with dataset size while WebPPL appears to scale quadratically. We suspect this is due to WebPPL implementation traversing the whole trace at MH updates, even though only a fixed number of variables at the end are candidates for updates. We avoid this issue using the `freeze` transformation and achieve linear scaling.

Figure 6.3 shows how execution times scale with the number of samples produced or the number of MH transitions performed. In all cases the scaling is linear as expected. For MH and SMC the slopes of all lines for the three systems are similar which indicates that the cost of an additional sample is similar, although Anglican and WebPPL again suffer from some initial overhead. For RM-SMC the slope of the line associated with our library is significantly higher than for WebPPL. We speculate the cause to be that each `mhStep` in our library goes through the

Table 6.1: Net addition of Lines of Code (LoC), excluding comments and import statements. The entry "N/A" means that an algorithm is not available in a given language.

	MH	SMC	RM-SMC	PMMH	SMC^2
MonadBayes	67	70	11	4	20
Anglican	100	87	N/A	N/A	N/A
WebPPL	314	334	0	N/A	N/A

`Pop` layer while WebPPL only does it once per rejuvenation sequence. We leave reducing that overhead for future work.

6.3.2 Qualitative Evaluation

To estimate the implementation effort involved in writing the inference algorithms, Table 6.1 lists the number of lines of code (LoC) used for this purpose in the three systems. Although LoC are not a reliable metric, particularly comparing across languages, they do offer some estimation of the implementation effort. WebPPL requires no additional lines to implement RM-SMC because it implements SMC as a special case of RM-SMC with zero rejuvenation steps. The figure shows that our modular implementation of MH and SMC is actually shorter than monolithic implementations in Anglican and WebPPL. Neither of these systems implements PMMH or SMC^2 which would involve substantial effort while they are short snippets of code in our library. We can expect this difference to become more and more pronounced as we continue to build complex inference algorithms from smaller and reusable building blocks.

Apart from the reduced number of LoC, our architecture makes the code more reliable, maintainable, and malleable for reasoning as it is close to our semantic validation of inference. In the next section, we also argue that our architecture enables modular testing. We expect these features to reduce the number of bugs in implementations and ease refactoring.

Finally, we profiled our implementation on these benchmarks to investigate the bottlenecks and suggest optimisations. The insights from profiling has lead us to two optimisations. First, we replaced `WriterT` with `StateT` in `W`, then we switched to a Church-encoded version of the free monad in `Tr`. We report the benchmark results after applying those optimisations. Profiling the final code reveals that a substantial amount of time, up to 90% depending on the benchmark, is spent on the overhead associated with inference representations and not on doing essential numeric work. The biggest offenders seem to be the `Pop` and `W` transformers. We hope that this overhead could be further reduced by suitable techniques but we do not see a clear way to do it.

6.4 Testing

An important benefit of the modular implementation is enhanced testing capabilities. Every inference transformation can be tested independently and the correctness of their composition follows from correctness of individual components. Furthermore, in certain circumstances it is possible to replace statistical tests with deterministic ones.

A standard approach to test an algorithm implementation is to compare with a reference implementation, less efficient but clearly correct, on a set of small examples. In the context of Bayesian inference we can use exact enumeration of a small discrete model like the sprinkler model from the introduction. Unfortunately, probabilistic algorithms only give approximate answers that can be arbitrarily bad with non-zero probability. We can therefore never be sure if the answers they provide are correct, and any statistical test is bound to produce both false positive and false negative results.

In our library we can perform deterministic tests of Monte Carlo methods by replacing the bottom monad `Sampler` with `Exact` that computes exact answers for discrete models. It is defined as follows, omitting conversions between `R` and `Log R`:

```
newtype Exact a = Exact {run :: [(a, Log R)]}
instance Monad Exact where
  return x = Exact [(x,1)]
  m >>= f = Exact
    [(y, p*q) | (x,p) ← run m, (y,q) ← run (f x)]
instance MonadSample Exact where
  random = error "Not available"
  bernoulli p = Exact [(True, p), (False, 1-p)]
instance MonadCond Exact where
  score w = Exact [((),w)]
normalForm :: Ord a ⇒ Exact a → [(a, Log R)]
-- sort, aggregate, and remove zeros
```

The function `normalForm` sorts the list according to the first components, aggregates weights of equal elements, and removes elements with zero weight. It allows us to compare distributions represented by lists for equality.

Any correct inference transformation should not alter the result of `Exact`. For example, if `~==` is an acceptable approximate floating-point equality, then we can write a deterministic test for `smc` as follows:

```
(normalForm . (>>= Exact) . runPopulation . smc 2 2)
  sprinkler ~== normalForm sprinkler
```

Our implementation of traces as `[R]` is fundamentally continuous so it does not work with `Exact`. However, a more elaborate trace type that distinguishes between continuous and discrete variables would enable us to write similar tests for `mhStep`.

Deterministic tests of the kind described above are limited in that they can only be applied to small discrete models and only verify certain aspects of correctness. In particular for SMC it only checks that the result is unbiased but not that it is consistent. Similarly a test for MH would only check that it preserves the posterior distribution but not that it converges to it. Nonetheless, we found those tests to be invaluable in practice. For example, if we forgot to preserve the total weight in `resample`, such a bug would quickly be caught by the test shown above.

Chapter 7

Compositions of Inference Algorithms

In this chapter we demonstrate how to build sophisticated inference algorithms by combining MH and SMC in different ways. We implement these algorithms by composing specific inference transformations, each of which is defined for a particular transformer. We adhere fully to the inference representation interface, and only compose the basic inference transformations and their hoistings, maintaining the inference representation abstraction. Our library is the first probabilistic programming system that supports this level of compositionality.

7.1 Resample-Move SMC

A common problem with particle filters is that of particle degeneracy, where after resampling many particles are the same, effectively reducing the sample size. One way to ameliorate this problem is to introduce rejuvenation moves, where after each resampling we apply a number of MCMC transitions to each particle independently, thus spreading them around the space. If we use an MCMC kernel that preserves the target distribution at a given step, the resulting algorithm is correct. This algorithm is known as the resample-move SMC (RM-SMC) and was originally introduced by Gilks and Berzuini [29]. Algorithm 1 presents pseudocode for a classical formulation of this algorithm, highlighting its constituent components.

To implement RM-SMC we use the stack `Seq Tr Pop`. Inlining the types, a program is interpreted as a population of traced coroutines. It allows us to apply MH transitions to partially executed coroutines, which is exactly what we require for the rejuvenation steps. The implementation of resample-move SMC is similar to that of SMC, with the introduction of `mhStep`.

Algorithm 1 Resample-Move Sequential Monte Carlo in its classical formualtion. Braces indicate the conceptual components of the algorithm that we directly reflect in the implementation.

for $i = 1 : N$ **do**
  $W_i = \frac{1}{N}$
end for } Spawn particles
for $t = 1 : T$ **do**
  $W \leftarrow \frac{1}{N} \sum_i W_i$
  for $i = 1 : N$ **do**
    $A_i \sim Categorical(\{W_j\}_{j=1}^N)$
    $\tilde{X}_i \leftarrow X_{A_i}$
    $W_i \leftarrow W$
  end for } Resample
  for $i = 1 : N$ **do**
    for $j = 1 : K$ **do**
      $\tilde{X}_i^t \sim K(\tilde{X}_i^t, \cdot)$
    end for } MH steps
  end for
  for $i = 1 : N$ **do**
    $X_i^t \sim p(x^t | \tilde{X}_i^{t-1})$
    $W_i^t = W_i^{t-1} \frac{p(dx^t, y^t | \tilde{X}_i^{t-1})}{p(dx^t | \tilde{X}_i^{t-1})}(X_i^t)$
  end for } Advance
end for

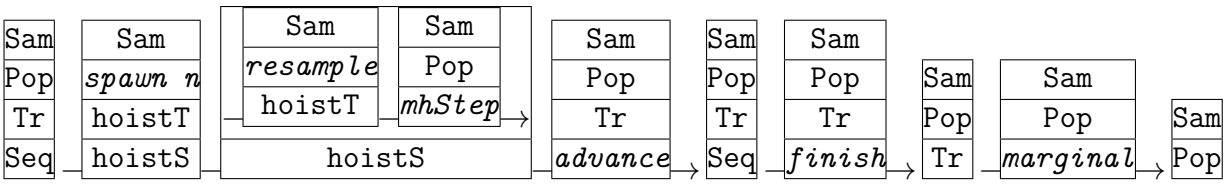


Figure 7.1: Graphical depiction of RM-SMC as a sequence of transformations between intermediate representations.

```
rmsmc :: MonadSample m
      ⇒ Int                  -- k : number of time steps
      → Int                  -- n : number of particles
      → Int                  -- t : number of MH steps
      → Seq (Tr (Pop m)) a   -- model
      → Pop m a              --result
rmsmc k n t = marginal . finish .
    compose k (advance . hoistS (
        compose t mhStep . hoistT resample)) .
    (hoistS . hoistT) (spawn n >>)
```

In the above `t` is the number of MH transitions to be applied after each resampling step. The inference stacks and transformations constituting `rmsmc` are depicted visually in Figure 7.1.

The version of RM-SMC presented above is computationally intensive. In some models it is better to restrict the rejuvenation transitions to the subset of random variables introduced since the last resampling. We can accomplish that using the `freeze` transformation from Section 6.2.2.

```
rmsmcLocal :: MonadSample m ⇒ Int → Int → Int →
              Seq (Tr (Pop m)) a → Pop m a
rmsmcLocal k n t = marginal . finish .
    compose k (advance . hoistS ( freeze .
        compose t mhStep . hoistT resample)) .
    (hoistS . hoistT) (spawn n >>)
```

7.2 Particle Marginal MH

RM-SMC uses the MH update inside SMC. An alternative composition is to use SMC inside an MH update. A particular instance is the algorithm called Particle Marginal Metropolis-Hastings (PMMH) [3], a pseudo-marginal MH algorithm that uses SMC to approximately integrate over the latent variables in the model. It is primarily used for parameter estimation in time series models.

PMMH is only applicable to models with a specific structure, namely the probabilistic program needs to decompose to a prior over the global parameters `m param` and the rest of the model `param → m a`. Combining these using `>>=` would yield the complete model of type `m a`. For example, the random walk model from Section 6.1.1 would be decomposed as follows:

```
s :: MonadSample m ⇒ m R
s = gamma 1 1
random_walk' :: MonadInfer m ⇒ [R] → R → m [R]
random_walk' ys s = do
  let obs x y = score (normalPdf x 1 y)
  let expand xs [] = return xs
      expand (x:xs) (y:ys) = do
        x' ← normal x s
        obs x' y
        expand (x':x:xs) ys
  xs ← expand [0] ys
  return (reverse xs)
```

The idea is to do MH on the parameters of the model. Recall that for MH we need to compute the likelihood for the particular values of parameters but that involves integrating over the remaining random variables in the model which is intractable. Fortunately to obtain valid MH it is sufficient to have an unbiased estimator for the likelihood which is produced by a single sample from `W`. MH with such an estimator is referred to as pseudo-marginal MH. If instead of taking a single weight from `W` we take the sum of weights from `Pop` we obtain an unbiased estimator with lower variance. In particular if such a `Pop` is a result of `smc` the resulting algorithm is known as PMMH.

The full implementation of PMMH is then as follows:

```
pmmh :: MonadInfer m
     ⇒ Int                    -- t: number of MH steps
     → Int                    -- k: number of time steps
     → Int                    -- n: number of particles
     → Tr m b                 -- param: model parameters prior
     → (b → Seq (Pop m) a)    -- model
     → m [[(a, Log R)]]       -- result
pmmh t k n param model =
  mh t (param >>= runPopulation . pushEvidence .
    hoistP lift . smc k n . model)
```

The code above can be read as follows. First it applies SMC to the model and lifts the entire SMC computation through `Tr`. Then it `score`s the sum of weights in `Tr` and keeps the population as the output. This process is depicted visually in Figure 7.2. The preprocessed computation is combined with the prior on parameters. Running `mh` then produces exactly the desired algorithm, since the only `score` in `Tr` is the sum of weights from the population.

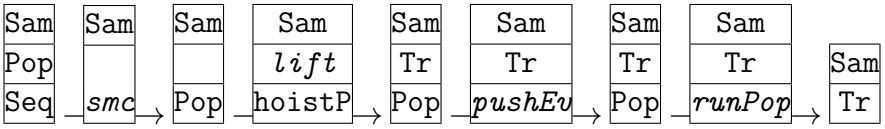


Figure 7.2: Graphical depiction of the preprocessing step involved in PMMH. Once the model is transformed in this fashion, it is combined with the parameters prior and a standard MH algorithm is executed on the resulting program.

7.3 SMC^2

The final inference algorithm we discuss, proposed by Chopin et al. [15], can be regarded as a hybrid of resample-move SMC and PMMH approaches and is used for joint estimation of the posterior over parameters and latent variables in state-space models. It features an outer population of particles, much like RM-SMC, each of which holds different values for the parameters. These particles are filtered through observations using resampling and MH-based rejuvenation where appropriate. However, like in PMMH, these MH transitions do not use exact densities but rather estimators obtained from an inner particle filter over the latent variables. The two particle filters are synchronised in the sense that they step through the same observations simultaneously.

Like PMMH, SMC^2 is only applicable to programs separable into the prior over parameters and the rest of the model. Furthermore, for SMC^2 we need a variant of `smc` that performs `pushEvidence` after each step. We call it `smcPush` and its implementation is almost identical to `smc`.

```
smcPush :: MonadInfer m ⇒
  Int → Int → Seq (Pop m) a → Pop m a
smcPush k n =
  finish . compose k (advance .
           hoistS (pushEvidence . resample)) .
  hoistS (spawn n >>)
```

We want to instantiate `m` to `Seq (Tr (Pop Sampler))` and run `rmsmc` on it. Unfortunately doing that naively has the unintended consequence that the random variables from `model` end up being traced which is not what we want. To remedy this situation we introduce a type synonym that performs the necessary lifting.

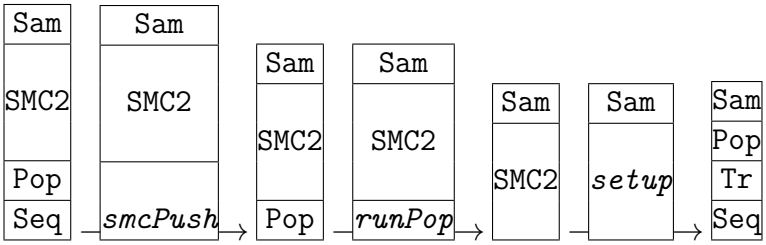


Figure 7.3: Graphical depiction of the preprocessing step involved in the implementation of SMC^2.

```
newtype SMC2 m a = SMC2 (Seq (Tr (Pop m)) a)
  deriving(Monad)
setup (SMC2 m) = m
instance MonadTrans SMC2 where
  lift = SMC2 . lift . lift . lift
instance MonadSample m ⇒ MonadSample (SMC2 m) where
  random = lift random
instance MonadCond m ⇒ MonadCond (SMC2 m) where
  score = SMC2 . score
```

The `SMC2` synonym thus ensures that the random variables bypass the transformers that need not be concerned with them. We can then complete the SMC2 implementation as follows.

```
smc2 :: MonadSample m
     ⇒ Int                              -- k: number of time steps
     → Int                              -- n: number of inner particles
     → Int                              -- p: number of outer particles
     → Int                              -- t: number of MH transitions
     → Seq (Tr (Pop m)) b               -- param: model parameters
     → ( b → Seq (Pop (SMC2 m)) a)      -- model
     → Pop m [(a, Log R)]
smc2 k n p t param model =
  rmsmc k p t (param >>= setup . runPopulation .
               smcPush k n . model)
```

Much like in the PMMH case, the implementation consists of a preprocessing step on the model, followed by combining the model with the parameters prior, followed by running an existing algorithm, in this case RMSMC. The preprocessing step is depicted visually in Figure 7.3.

Our framework allows more complicated compositions of similar kind, for example using RM-SMC within SMC^2 or introducing two types of parameters with varying scope. Their implementation would be analogous to the examples presented above.

Chapter 8

Conclusion

Design of Bayesian inference algorithms for probabilistic programs is a never-ending task where new variations of existing methods are proposed in literature every year. Implementation of these algorithms is subject to all the usual pitfalls of building software, as well as many problems that do not manifest themselves when writing deterministic programs. Even though conceptually many algorithmic developments are simple, incorporating them into existing implementations often involves making major changes and risks introducing bugs along the way. This directly limits the adoption of new inference algorithms and therefore restricts the set of models for which practitioners can easily perform inference.

In this dissertation we presented a modular construction for building inference algorithms for probabilistic programs, based on monad transformers, and showed that it can be used to decompose complicated inference algorithms into a small collection of easy-to-understand building blocks which we subsequently put together in various ways. Specifically, we focused on Monte Carlo algorithms and developed building blocks corresponding to the ideas of resampling, sequential sampling, and Metropolis-Hastings updates. We used them to construct algorithms such as Sequential Monte Carlo, Resample-Move Sequential Monte Carlo, Particle Marginal Metropolis-Hastings, and Sequential Monte Carlo squared. Once the building blocks are defined, each of these otherwise complicated algorithms can be implemented in a just few lines of code. Moreover, individual operations in this code correspond to high-level logical components of these algorithms.

We have developed a theoretical framework around a specific variant of the lambda calculus, which allows formal reasoning about inference algorithms and proving their correctness, and a Haskell implementation that can be used in practice to write probabilistic programs and perform inference in them. The calculus and the library are closely related, sharing the fundamental aspect of their design, but there remains a gap between them. Thus we do not have formal semantics for the programs actually run and we can not presently verify correctness of our

inference algorithm implementations using formal methods, although we expect this gap to be bridged in the near future. The design presented in this dissertation could then form a foundation for a formally verified probabilistic programming library. In the meantime we have shown how the proposed modular implementation can be used to enable a new class of deterministic testing methods for randomized inference algorithms.

The use of Haskell was convenient, as we could reuse the existing support for monads and their transformers for inference representations. However, the same design ports to other modern functional languages that contain higher-order functions and inductive types. In the absence of a type-class mechanism, one can use ML-style modules: users construct models abstractly with respect to a module signature containing `random` and `score`. Each inference representation is a module implementing this signature, and inference transformers are functors. To a certain extent our design can also be ported to imperative languages, although the lack of a powerful type system would likely make it more difficult to ensure correctness of implementations.

We have achieved performance comparable with existing probabilistic programming libraries available in general-purpose languages, although profiling shows there remains a significant overhead associated with the abstractions we use. While the convenience of a high-level language may in many cases already be worth reduced performance, additional work on minimising this overhead would make our approach even more practical.

Apart from optimising the code to increase computational efficiency, there are certain additions to the MonadBayes library that could increase the statistical efficiency. In particular these include the tools that would enable users to specify proposal distributions and transition kernels in a model-specifc way. In our implementation that would come mostly in the form of a more sophisticated `Tr` representation that could refer to specific program variables in a user-friendly manner.

An alternative approach to embedding probabilistic programs in existing languages is to use a stand-alone DSL for constructing models, such as used in Stan [12] and LibBi [61]. Such representations avoid the overheads discussed above and as a result can be used to generate computationally efficient inference code. It would be interesting to investigate if the approach we described here could be used to build modular compilers for such languages without sacrificing their efficiency. The idea to use monad transformers to build compilers in a modular fashion has been previously explored by Liang and Hudak [51].

Our semantic account was provided for a relatively expressive lambda calculus, although it lacked a mechanism to specify general recursion. This precludes some kinds of probabilistic programs, in particular those corresponding to Bayesian nonparametric models [78]. While a probabilistic program that diverges with non-zero probability is not very useful, there exists a class of programs where infinite runs are possible but the program terminates with probability

one. Constructing a domain theory for QBS would allow us to extend our semantic construction to such programs, where the inference algorithms provided should still work.

Many modern algorithms for Bayesian inference, such as Hamiltonian Monte Carlo [65] and black-box variational inference [75] rely on gradient information that can be obtained by methods of automatic differentiation. Inclusion of these methods can make a probabilistic programming system dramatically more practical as demonstrated by the success of Stan. We could incorporate these algorithms into our framework, the only obstacle being the lack of availability of suitable automatic differentiation tools. On the semantics side the problem is that no denotational account of automatic differentiation exists as of writing this paragraph, although it is being worked on by programming languages researchers. On the implementation side we could not find a suitable automatic differentiation library in Haskell. We have experimented with the ad library[1] but the resulting types were so complicated that we did not find the results satisfactory, although it did work. We hope that a more clever implementation hiding type-level complexities or an alternative automatic differentiation package will unlock the power of gradient-based inference algorithms for our library.

Finally, modern machine learning, especially in the context of big data, increasingly makes use of dedicated numerical libraries for efficiently performing matrix operations, including automatic differentiation, on GPUs. Currently the most popular choices are Tensorflow [1] and PyTorch[2]. Probabilistic programming systems such as Edward [88] and Pyro[3] exploit these libraries to deliver massive performance boosts. For example, Tran et al. [88] report that Hamiltonian Monte Carlo in Edward is more than an order of magnitude faster than in Stan. Taking advantage of such libraries could therefore greatly improve computational efficiency of inference.

Currently, PyTorch is only available in Python and existing Tensorflow bindings for Haskell and OCaml are not supported officially. There are several alternatives written directly in functional programming languages that cover all the core features required although they presently do not easily combine. For example, the Haskell library Accelerate [13] supports efficient matrix computation on CPUs and GPUs, but there is no automatic differentiation library built on top of it. On the other hand a recently released Owl library [90] for OCaml offers efficient matrix operations and automatic differentiation, but currently only emits CPU code. Since all the components are already there, it is simply a matter of investing sufficient developer time to create a natively functional numerical library for modern machine learning. We believe that in the near future such libraries, whether implemented from scratch in functional languages or as convenient bindings to imperative libraries, will make functional programming a serious

[1] http://hackage.haskell.org/package/ad
[2] https://github.com/pytorch
[3] http://pyro.ai/

player in the area of machine learning in general and probabilistic programming in particular. We hope that the constructions presented in this dissertation will help bring the modularity and reliability often associated with functional programming into the realm of state-of-the-art machine learning applications.

An alternative to developing numerical libraries for functional programming languages is to transfer our developments to imperative languages where these tools already exist. Unfortunately the design of these languages makes it difficult to use sophisticated compositional constructions of the kind presented in this dissertation. Nonetheless, attempting to bring them over to mainstream machine learning languages is a worthwhile direction for future work.

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems.

[2] Ackerman, N. L., Freer, C. E., and Roy, D. M. (2011). Noncomputable conditional distributions. In *LiCS*.

[3] Andrieu, C., Doucet, A., and Holenstein, R. (2010). Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society*, 72:269–342.

[4] Andrieu, C. and Roberts, G. O. (2009). The pseudo-marginal approach for efficient Monte Carlo computations. *The Annals of Statistics*, 37:697–725.

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

[6] Barber, D. (2012). *Bayesian Reasoning and Machine Learning*. Cambridge University Press.

[7] Bishop, C. (2006). *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer-Verlag New York.

[8] Blažević, M. (2011). Coroutine pipelines. *The Monad Reader*, pages 29–50.

[9] Borgström, J., Dal Lago, U., Gordon, A. D., and Szymczak, M. (2016). A lambda-calculus foundation for universal probabilistic programming. In *ICFP*.

[10] Borgström, J., Gordon, A. D., Greenberg, M., Margetson, J., and Van Gael, J. (2011). Measure transformer semantics for Bayesian machine learning. In *ESOP*.

[11] Brooks, S., Gelman, A., Jones, G. I., and Meng, X. (2011). *Handbook of Markov chain Monte Carlo*. Chapman and Hall.

[12] Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of Statistical Software*, 76.

[13] Chakravarty, M. M. T., Keller, G., Lee, S., McDonell, T. L., and Grover, V. (2011). Accelerating haskell array codes with multicore gpus. In *DAMP*.

[14] Chang, J. T. and Pollard, D. (1997). Conditioning as disintegration. *Statistica Neerlandica*, 51:287–317.

[15] Chopin, N., Jacob, P. E., and Papaspiliopoulos, O. (2013). SMC2: an efficient algorithm for sequential analysis of state space models. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 75:397–426.

[16] Church, A. (1941). *The calculi of lambda-conversion*. Princeton University Press.

[17] Del Moral, P. (1996). Non linear filtering: Interacting particle solution. *Markov Processes and Related Fields*, 2:555–580.

[18] Douc, R., Cappé, O., and Moulines, E. (2005). Comparison of resampling schemes for particle filtering. arXiv:cs/0507025.

[19] Doucet, A. and Johansen, A. M. (2011). A tutorial on particle filtering and smoothing: Fifteen years later. In Crisan, D. and Rozovskii, B., editors, *The Oxford Handbook of Nonlinear Filtering*, chapter 8. Oxford University Press.

[20] Duane, S., Kennedy, A., Pendleton, B. J., and Roweth, D. (1987). Hybrid Monte Carlo. *Physics Letters B*, 195:216–222.

[21] Ehrhard, T., Pagani, M., and Tasson, C. (2018). Measurable cones and stable, measurable functions. *Proceedings of the ACM on Programming Languages*, POPL.

[22] Erwig, M. and Kollmansberger, S. (2006). Probabilistic functional programming in Haskell. *Journal of Functional Programming*, 16:21–34.

[23] Felleisen, M. (1991). On the expressive power of programming languages. *Sci. Comput. Program.*, 17(1-3):35–75.

[24] Fiore, M. and Saville, P. (2017). List objects with algebraic structure. In *2st International Conference on Formal Structures for Computation and Deduction, FSCD 2017*.

[25] Ge, H., Xu, K., and Ghahramani, Z. (2018). Turing: a language for flexible probabilistic inference. In *AISTATS*.

[26] Geman, S. and Geman, D. (1984). Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions of Pattern Analysis and Machine Intelligence*, PAMI-6:721–741.

[27] Geuvers, H. and Poll, E. (2007). Iteration and primitive recursion in categorical terms. In *Reflections on Type Theory, Lambda Calculus, and the Mind, Essays Dedicated to Henk Barendregt on the Occasion of his 60th Birthday*, pages 101–114, Nijmegen. Radboud Universiteit.

[28] Geyer, C. (2011). Introduction to MCMC. In *Handbook of Markov chain Monte Carlo*. Chapman and Hall.

[29] Gilks, W. and Berzuini, C. (2001). Following a moving target - Monte Carlo inference for dynamic Bayesian models. *Journal of the Royal Statistical Society*, 63:127–146.

[30] Gilks, W. R., Thomas, A., and Spiegelhalter, D. J. (1994). A language and program for complex Bayesian modelling. *Journal of the Royal Statistical Society. Series D*, 43.

[31] Goodman, N., Mansinghka, V., Roy, D., Bonawitz, K., and Tenenbaum, J. (2008). Church: a language for generative models. In *UAI*.

[32] Goodman, N. and Stuhlmüller, A. (2014). Design and implementation of probabilistic programming languages. http://dippl.org.

[33] Gordon, A. D., Henziger, T. A., Nori, A. V., and Rajamani, S. K. (2014). Probabilistic programming. In *FOSE*.

[34] Gordon, N. J., Salmond, D. J., and Smith, A. F. M. (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEE Proceedings F - Radar and Signal Processing*, 140:107–113.

[35] Green, P. J. (1995). Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika*, 82:711–732.

[36] Haghverdi, E. and Scott, P. (2006). A categorical model for the geometry of interaction. *Theoretical Computer Science*, 350(2):252 – 274. Automata, Languages and Programming: Logic and Semantics (ICALP-B 2004).

[37] Hastings, W. K. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57:97–109.

[38] Heunen, C., Kammar, O., Staton, S., and Yang, H. (2017). A convenient category for higher-order probability theory. In *LiCS*.

[39] Hoffman, M. D. and Gelman, A. (2014). The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15:1593–1623.

[40] Hur, C.-K., Nori, A., Rajamani, S. K., and Samuel, S. (2015). A provably correct sampler for probabilistic programs. In *FSTTCS*.

[41] Hutton, G. (1999). A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9(4):355–372.

[42] Jacobs, B. (2017). From probability monads to commutative effectuses. *Journ. of Logical and Algebraic Methods in Programming*. To appear.

[43] Jaskelioff, M. (2009). *Lifting of Operations in Modular Monadic Semantics*. PhD thesis, University of Nottingham.

[44] Jones, C. and Plotkin, G. (1989). A probabilistic powerdomain of evaluations. In *LiCS*.

[45] Kelly, G. M. (1980). A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves and so on. *Bull. Austral. Math. Soc.*, 22:1–83.

[46] Kiselyov, O. and Shan, C. (2009). Embedded probabilistic programming. In *IFIP Working Conference on Domain-Specific Languages*.

[47] Kock, A. (1972). Strong functors and monoidal monads. *Archiv der Mathematik*, 23(1):113–120.

[48] Kock, A. (2012). Commutative monads as a theory of distributions. *Theory and Applications of Categories*, 26(4):97–131.

[49] Kozen, D. (1981). Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22:328–350.

[50] Landin, P. J. (1966). The next 700 programming languages. *Communications of the ACM*, 9:157–166.

[51] Liang, S. and Hudak, P. (1996). Modular denotational semantics for compiler construction. In *ESOP*.

[52] Liu, J. S. and Chen, R. (1996). Sequential Monte Carlo methods for dynamic systems. *Journal of the American Statistical Association*, 93:1032–1044.

[53] MacKay, D. J. C. (2003). *Information Theory, Inference and Learning Algorithms*. Cambridge University Press.

[54] Mansinghka, V., Selsam, D., and Perov, Y. (2014). Venture: a higher-order probabilistic programming platform with programmable inference. arXiv:1404.0099.

[55] Marlow, S. (2010). Haskell 2010 language report. https://www.haskell.org/onlinereport/haskell2010/.

[56] Marmolejo, F. and Wood, R. J. (2010). Monads as extension systems — no iteration is necessary. *Theory and Applications of Categories*, 24(4):84–113.

[57] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., and Teller, A. H. (1953). Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21.

[58] Minka, T., Winn, J., Guiver, J., Webster, S., Zaykov, Y., Yangel, B., Spengler, A., and Bronskill, J. (2014). Infer.NET 2.6. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[59] Moggi, E. (1989). Notions of computation and monads. In *LiCS*.

[60] Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning. MIT Press.

[61] Murray, L. M. (2013). Bayesian state-space modelling on high-performance hardware using libbi. arXiv:1306.3277.

[62] Narayanan, P., Carette, J., Romano, W., Shan, C., and Zinkov, R. (2016). Probabilistic inference by program transformation in Hakaru (system description). In *FLOPS*.

[63] Neal, R. M. (1993). Probabilistic inference using Markov chain Monte Carlo methods. Technical Report CRG-TR-93-1, Department of Computer Science, University of Toronto.

[64] Neal, R. M. (1998). Annealed importance sampling. Technical Report 9805, Department of Statistics and Department of Computer Science, University of Toronto.

[65] Neal, R. M. (2010). MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*. Chapman and Hall.

[66] Nori, A. V., Hur, C.-K., Rajamani, S. K., and Samuel, S. (2014). R2: An efficient MCMC sampler for probabilistic programs. In *AAAI*.

[67] Park, S., Pfenning, F., and Thrun, S. (2008). A probabilistic programming language based upon sampling functions. *ACM Transactions on Programming Languages and Systems*, 31.

[68] Patil, A., Huard, D., and Fonnesbeck, C. J. (2010). PyMC: Bayesian stochastic modelling in Python. *Journal of Statistical Software*, 35.

[69] Pearl, J. (1982). Reverend Bayes on inference engines: A distributed hierarchical approach. In *AAAI*.

[70] Pfeffer, A. (2001). IBAL: A probabilistic rational programming language. In *IJCAI*.

[71] Pfeffer, A. (2015). *Practical Probabilistic Programming*. Manning.

[72] Piróg, M. (2016). Eilenberg-Moore monoids and backtracking monad transformers. In Atkey, R. and Krishnaswami, N. R., editors, *Proceedings 6th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2016, Eindhoven, Netherlands, 8th April 2016.*, volume 207 of *EPTCS*, pages 23–56.

[73] Raiffa, H. and Schlaifer, R. (1961). *Applied Statistical Decision Theory*. Division of Research, Graduate School of Business Adminitration, Harvard University.

[74] Ramsey, N. and Pfeffer, A. (2002). Stochastic lambda calculus and monads of probability distributions. In *POPL*.

[75] Ranganath, R., Gerrish, S., and Blei, D. (2014). Black-box variational inference. In *AISTATS*.

[76] Ritchie, D., Mildenhall, B., Goodman, N. D., and Hanrahan, P. (2015). Controlling procedural modeling programs with stochastically-ordered sequential Monte Carlo. *ACM Transactions on Graphics*, 34.

[77] Roth, D. (1996). On the hardness of approximate reasoning. *Artificial Intelligence*, 82:279–302.

[78] Roy, D., Mansinghka, V., Goodman, N., and Tenenbaum, J. (2008). A stochastic programming perspective on nonparametric Bayes. ICML workshop on nonparametric Bayesian methods.

[79] Ścibior, A., Kammar, O., and Ghahramani, Z. (2018a). Functional programming for modular Bayesian inference. *Proceedings of the ACM on Programming Languages*, 2.

[80] Ścibior, A., Kammar, O., Vákár, M., Staton, S., Yang, H., Cai, Y., Ostermann, K., Moss, S. K., Heunen, C., and Ghahramani, Z. (2018b). Denotational validation of higher-order Bayesian inference. *Proceedings of the ACM on Programming Languages*, 2.

[81] Scott, D. and Strahey, C. (1971). Towards a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*.

[82] Shan, C. and Ramsey, N. (2017). Exact Bayesian inference by symbolic disintegration. In *POPL*.

[83] Staton, S. (2017). Commutative semantics for probabilistic programming. In *ESOP*.

[84] Staton, S., Yang, H., Heunen, C., Kammar, O., and Wood, F. (2016). Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *LiCS*.

[85] Strachey, C. (2000). Fundamental concepts in programming languages. 13:11–49.

[86] Tolpin, D., van de Meent, J., and Wood, F. (2015). Probabilistic programming in Anglican. In *Machine Learning and Knowledge Discovery in Databases*. Springer.

[87] Toronto, N., McCarthy, J., and Van Horn, D. (2015). Running probabilistic programs backwards. In *ESOP*.

[88] Tran, D., Hoffman, M. D., Saurous, R. A., Brevdo, E., Murphy, K., and Blei, D. M. (2017). Deep probabilistic programming. In *ICLR*.

[89] van de Meent, J.-W., Yang, H., Mansinghka, V., and Wood, F. (2015). Particle Gibbs with ancestor sampling for probabilistic programs. In *AISTATS*.

[90] Wang, L. (2017). Owl: A general-purpose numerical library in OCaml. arXiv:1707.09616.

[91] Wingate, D., Stuhlmüller, A., and Goodman, N. (2011). Lightweight implementations of probabilistic programming languages via transformational compilation. In *AISTATS*. The published version contains a serious bug in the algorithm description, which was fixed in Revision 3 available from the authors page.

[92] Wood, F., van de Meent, J.-W., and Mansinghka, V. (2014). A new approach to probabilistic programming inference. In *AISTATS*.

[93] Zhang, N. L. and Poole, D. (1994). A simple approach to Bayesian network computations. In *CCAI*.

[94] Zinkov, R. and Shan, C. (2016). Composing inference algorithms as program transformations. In *UAI*.