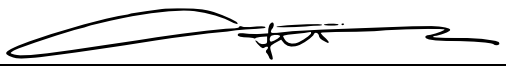
Thesis Consent Form

This thesis may be consulted for the purposes of research or private study provided that due acknowledgement is made where appropriate and that permission is obtained before any material from the thesis is published. Students who do not wish their work to be available for reasons such as pending patents, copyright agreements, or future publication should seek advice from the Graduate Centre as to restricted use or embargo.

Author of thesis	Jared Tobin
Title of thesis	Embedded Domain-Specific Languages for Bayesian Modelling and Inference
Name of degree	Doctor of Philosophy (Statistics)
Date Submitted	2017/05/01

Print Format (Tick the boxes that apply)	
☑	I agree that the University of Auckland Library may make a copy of this thesis available for the collection of another library on request from that library.
☑	I agree to this thesis being copied for supply to any person in accordance with the provisions of Section 56 of the Copyright Act 1994.
Digital Format - PhD theses	
I certify that a digital copy of my thesis deposited with the University is the same as the final print version of my thesis. Except in the circumstances set out below, no emendation of content has occurred and I recognise that minor variations in formatting may occur as a result of the conversion to digital format. Access to my thesis may be limited for a period of time specified by me at the time of deposit. I understand that if my thesis is available online for public access it can be used for criticism, review, news reporting, research and private study.	
Digital Format - Masters theses	
I certify that a digital copy of my thesis deposited with the University is the same as the final print version of my thesis. Except in the circumstances set out below, no emendation of content has occurred and I recognise that minor variations in formatting may occur as a result of the conversion to digital format. Access will normally only be available to authenticated members of the University of Auckland, but I may choose to allow public access under special circumstances. I understand that if my thesis is available online for public access it can be used for criticism, review, news reporting, research and private study.	
Copyright (Digital Format Theses) (Tick **ONE box only**)	
☑	I confirm that my thesis does not contain material for which the copyright belongs to a third party, ***(or)*** that the amounts copied fall within the limits permitted under the Copyright Act 1994.
☐	I confirm that for all third party copyright material in my thesis, I have obtained written permission to use the material and attach copies of each permission, ***(or)*** I have removed the material from the digital copy of the thesis, fully referenced the deleted materials and, where possible, provided links to electronic sources of the material.

Signature  Date 2018/01/10

Comments on access conditions

Faculty Student Centre / Graduate Centre only: Digital copy deposited □ Signature Date

Embedded Domain-Specific Languages for Bayesian Modelling and Inference

A dissertation submitted to the University of Auckland
in fulfillment of the requirements of the degree of
Doctor of Philosophy (Ph.D.) in Statistics

Jared Tobin

January 11, 2018

Abstract

This dissertation defends the thesis that novel and useful domain-specific languages for solving statistical problems can be embedded in statically-typed, purely-functional programming languages.

It presents techniques for representing probability distributions in embedded languages, deeply-embedding a type-safe probabilistic programming language in a way that is amenable to inference, and embedding a language for building composite Markov transition operators that can be used in MCMC.

Acknowledgements

I owe a debt of thanks to everyone and everything that encouraged me to enrol in a PhD programme and helped me see it through to the end.

Thanks to Josh Stella & the crew at Fugue, Tom Nielsen at Openbrain, Richard Dunne at Bdellium, and my old friends at T&W for being patient with me while I slogged away at this PhD. And a special thanks to Noel Cadigan and Gary Sneddon — my former supervisors — as well as Leslie Kennedy — my former high-school English teacher — each for having been a rather direct impetus for my success in some way.

Thanks go to Matt Might and Dan Roy — two people I've never met, but whose research and writing compelled me to learn more about computer science, probabilistic programming, and Haskell. And whoever it was who posted that notification on the Well-Typed blog about the 2012 Haskell summer school in France — apparently it was Eric Kow — do I ever owe you a drink or something.

Thanks to all the authors of the high-quality open-source software and open-access research that I use every day. You're an inspiration and I want to be just like you.

Thanks to my fantastic supervisor Russell, not only for your patience and ever-constructive advice, but also the freedom you gave me to explore literally life-changing avenues of statistics and computer science.

But most importantly: without my wonderful family and friends, this document would surely not exist. My partner Nadine, my parents Bob & Marilyn, my siblings Shawn & Rachel, and my friends around the world: the notion that I could express my gratitude through some mere poetry is comedy, so I will continue to demonstrate my appreciation to you for as long as I live.

Contents

List of Figures

Chapter 1

Thesis

> The point is: what are you trying to show? The point is: what is your point?
>
> Olin Shivers

Novel and useful domain-specific languages for solving problems in Bayesian statistics can be embedded in statically-typed, purely-functional programming languages.

This dissertation defends that thesis, developing an argument in support of it by way of three chapters. Each chapter presents some conceptual issue or problem in statistics and demonstrates how it can be solved by some feat of *language engineering*; namely, embedding a limited domain-specific language for solving the problem inside of a host language. The ideas in this dissertation are all implemented using Haskell, probably the most popular and best-supported statically-typed and purely-functional programming language.

This thesis is both inspired by and supports ongoing research in *probabilistic pro-*

gramming: the idea of using specialized programming languages, interpreters, compilers, hardware, and other similar tools to do Bayesian statistics [Mansinghka, 2009]. In particular, this dissertation contributes primarily to research in *embedded* probabilistic programming languages, which have the advantage of being able to hijack many desirable features (parser, compiler infrastructure, library and extension ecosystem, etc.) from their host language.

The primary contributions of this dissertation are:

- Novel **probabilistic interpretations** of the Giry monad's algebraic structure. Most significantly, **we characterize image measure by functorial structure** and **product measure by applicative structure**. The functorial structure is demonstrated to be useful for transforming a measure's support while preserving its density structure, and the applicative/product measure structure is demonstrated to be useful for encoding independence between measurable functions.

- A novel **characterization of the Giry monad as a restricted continuation monad**. We implement a shallowly-embedded DSL for integration by using a dual interpretation for probability measures, encoding them as self-contained integration procedures that one can 'query' by integrating measurable functions against. We note that this language is structurally equivalent to the 'expectation monad' of Ramsey and Pfeffer [2002] since both are continuation-based encodings of the Giry monad. We develop a number of queries — notably **measure convolution** and recovery of **moment/cumulant-generating and cumulative distribution functions** — over measures defined over varying supports.

- A novel technique for **embedding a statically-typed probabilistic programming language in a purely functional language**. We use the *free monad* of a probabilistic base functor in order to define our embedded language, giving us the same syntax as the language based on the Giry or

sampling monads, but with considerably more flexibility when it comes to interpretation.

- A novel **characterization of execution traces as cofree comonads**. We demonstrate that probabilistic programs encoded using the free monad have a dual representation as execution traces under the *cofree comonad*, which allows us to 'move about' in trace space and perturb a model's internal parameters. We then implement a novel **comonadic Markov Chain Monte Carlo (MCMC) algorithm** that makes use of this characterization.
- A novel technique for **statically encoding conditional independence** of terms in the embedded language. We use the *free applicative functor* in order to capture applicative expressions in a structure-preserving way.
- A novel technique for **building custom transition operators for use in Markov Chain Monte Carlo**. Markov transition operators can be denoted by a particular instance of the *state monad* such that familiar monadic combinators can be used to build composite transition operators from a set of base, 'known-good' primitives.

Each of the above techniques has been implemented in one or more Haskell libraries which are liberally licensed and available on Github.

The rest of this chapter describes how the dissertation is structured. Chapter 2 is a background chapter that takes pains to provide the minimum necessary background required to understand the computer science concepts, Haskell terminology, and programs introduced throughout the dissertation; it can likely be used as a reference and flipped to when needed. Chapters 3, 4, and 5 present the primary contributions.

1.1 Representing Probability Distributions

Chapter 3 explores much of the existing ground and discusses some fundamentals of representing probability distributions in typed functional languages. It presents the *monadic structure* of probability distributions, developed originally in mathematics by Lawvere [1962] and Giry [1981] and then discussed seminally with respect to functional programming by Ramsey and Pfeffer [2002] before being importantly extended on by Park et al. [2008] and Ścibior et al. [2015].

We develop the Giry monad from first principles, using it to characterize important probabilistic semantics common to embedded monadic probabilistic programming languages. We then implement the Giry monad using a restricted continuation monad through which the thorny details of preserving measurability can be abstracted away. This representation proves to be interesting and accurately captures important probabilistic semantics, but its prohibitive computational complexity and some issues around implementing integration limits its use in practice.

1.2 Representing Structured Probabilistic Models

We start by describing the well-known *sampling monad* and then note that it shares the same monadic structure as the Giry monad from Chapter 3. These constructions, however, are each 'lossy' in some sense; interpreting a distribution and producing some output destroys the internal structure of the distribution being interpreted by collapsing it to a point.

Chapter 4 presents a novel way to keep the same lightweight syntax used in both the measure and sampling function-based DSLs, but also preserve the model's internal structure. In particular, the structure-preserving concept of algebraic

freeness is exploited to reify a distribution as a data structure that can be traversed and analyzed to support almost arbitrary interpretation. The result is a deeply-embedded language for working with structured probabilistic *models* that are more amenable to inference. Numerous useful interpreters are described for working with these models. Sharing the same monadic structure captured by the *free monad*, each of the measure and sampling function-based languages can almost trivially be re-used and grafted onto the structure-preserving embedded language.

1.3 Declarative Markov Chains

In similar fashion to the previous chapters, Chapter 5 demonstrates that the Markov transition operators used in MCMC have a simple monadic structure. Additionally they can be combined in a way that preserves existing properties that are important for MCMC — Markovness, stationarity, and reversibility.

These properties are used to implement transition operators as monadic *state transitions* in a simple shallowly-embedded language for building composite transition operators. The language — implemented as a library called *declarative* [Tobin, 2013a] — can be used to build transition operators for any 'single-particle' Markov chain.

The *declarative* language includes transition operators corresponding to the Metropolis-Hastings, slice sampling [Neal, 2003], Hamiltonian Monte Carlo (HMC) [Neal, 2011], Metropolis-adjusted Langevin Diffusion (MALA), and No U-Turn Sampler (NUTS) [Hoffman and Gelman, 2011] algorithms. These can be fruitfully assembled to design compound transition operators that balance exploratory power with computational expense, for example by interleaving a large number of cheap Metropolis transitions with the occasional computationally expensive, gradient-based HMC transition. Examples are provided over a number of target

functions.

1.4 Scope of Formality

The DSLs developed in support of the thesis make a number of guarantees about the behaviour of programs written in them. What follows is a brief summary of the scope of formality aimed at and achieved in the languages presented in Chapters 3-5.

The embedded languages developed in this dissertation are in general constructed around:

- a particular abstract data type or types,
- a characterization of structure of those types, by way of e.g. functors or monads, and
- a number of evaluation or query functions.

The scope of formality is thus to demonstrate that the data type or types we employ accurately capture some fundamental probabilistic construct, that the categorical structure of the probabilistic construct is characterized accurately, and that the corresponding evaluation or query functions are well-motivated and correct.

Each chapter presents some statistical construct that is first formalized mathematically, and then distilled into some concise data type and functions for evaluating values of it. We then use the categorical structure of the type to characterize some corresponding probabilistic structure, the result of which is a set of combinators for manipulating values of the type in a law-abiding and semantics-preserving fashion. These constitute our embedded DSLs.

The mathematical formalizations used are concrete to a degree of rigour that is likely satisfactory to the academic or practicing statistician. When translating to code, we either directly prove, or cite some existing proof, that any claimed categorical structure holds as described and required. The embedded languages presented here are thus demonstrated to be correct at the level of formality described above. Other degrees of formality, whatever they might be, are deemed out of scope.

Various well-motivated supplementary functions — e.g. probability density function implementations, primitive Markov transition functions, pure pseudorandom number generators — are not treated with the same formality. The rationale is that even if there were an error in the implementation of any of these accessories, it would not invalidate the correctness of the embedded languages at the level of formality described above. We do however employ a variety of common-sense and easily-verified sanity tests throughout the dissertation, in order to both illustrate the languages and also provide additional assurance that the implementations are correct.

1.5 Wrapping up

Chapter 6 concludes the dissertation by providing a unifying summary, with attention to arguments made in support of the thesis. Appendix A includes code for one of the constructions developed in Chapter 4, and Appendix B includes code for the primitive transition operators described in Chapter 5.

Libraries and code developed in support of this thesis are summarized alphabetically below, along with URLs to their respective Github repositories:

- *deanie*: An embedded probabilistic programming language.
  `http://github.com/jtobin/deanie`

- *declarative*: DIY Markov chains.
  http://github.com/jtobin/declarative

- *flat-mcmc*: Painless general-purpose sampling.
  http://github.com/jtobin/flat-mcmc

- *hasty-hamiltonian*: Speedy traversal through parameter space.
  http://github.com/jtobin/hasty-hamiltonian

- *hnuts*: Automatic gradient-based sampling.
  http://github.com/jtobin/hnuts

- *lazy-langevin*: Gradient-based diffusion.
  http://github.com/jtobin/lazy-langevin

- *mcmc-types*: Common types for implementing MCMC algorithms.
  https://github.com/jtobin/mcmc-types

- *measurable*: A shallowly-embedded DSL for basic measure wrangling.
  http://github.com/jtobin/measurable

- *mighty-metropolis*: The Metropolis sampling algorithm.
  http://github.com/jtobin/mighty-metropolis

- *mwc-probability*: Sampling function-based probability distributions.
  http://github.com/jtobin/mwc-probability

- *speedy-slice*: Speedy slice sampling.
  http://github.com/jtobin/speedy-slice

Chapter 2

Language Engineering in Haskell

> A monad is just a monoid in the category of endofunctors, what's the problem?
>
> Philip Wadler (attributed)

2.1 Abstract and Contributions

This chapter provides background on important concepts in typed functional programming, as well as an introduction to Haskell and its syntax.

Domain-specific languages and typed functional programming are undoubtedly esoteric topics amongst statisticians. This chapter focuses on the concepts of *algebraic data types*, important *typeclasses*, and how to use these ideas to define *embedded domain specific languages* (EDSLs) that can be used within a host language like Haskell. Special attention is paid to the *monad* typeclass — a class of data structures with particular characteristics that can be used to denote and

enforce useful probabilistic semantics. Monads are demonstrated to be a useful foundational choice for representing distributions in embedded languages, with emphasis on the useful *composition* and guarantee of strong *type safety* they provide.

Along the way, we build up a toy shallowly-embedded domain specific language for representing and manipulating discrete probability distributions.

2.2 Motivation

The term 'probability distribution' is in practice somewhat ambiguous. A probability distribution is an *abstract concept* that can be distinctly characterized by a variety of concrete representations. Canonically probability and distributions are typically defined in terms of measures, but random variables, probability density functions, characteristic functions, cumulative distribution functions, etc. are used to represent distributions, depending on the problem under consideration.

Probably the most common representation in applied work is the probability density or mass function: a measurable function f from some support Ω to $\mathbb{R}$ such that $\int_\Omega f d\mu = 1$ for an appropriate measure μ. This is a useful representation for many common statistical tasks, namely for calculating probabilities and expectations by integrating over it.

When it comes to representing distributions on computers we're typically interested in doing all sorts of things with them, and the density function representation can be unsatisfying for at least two reasons.

First, frequently one doesn't actually *wish* to calculate probabilities and expectations explicitly. Instead, another characterization — like *sampling functions* — may be more useful, either for simulation or other approximate work.

But second — and more importantly — it is not necessarily easy to encode the *laws* of this representation in any consistent way. For example, one might wish to ensure that given two density functions f and g, the two can be combined or *composed* in some sense to form a function h that is also a valid density function for some distribution. It's not clear how to enforce this for all valid density functions one might want to consider.

In particular, **composition of distributions is integral to applied Bayesian statistics**. Typically we are interested in forming some *probabilistic model* for some phenomenon by combining distributions together in some hierarchical structure. Indeed, a hierarchical Bayesian model is nothing more than a collection of probability distributions composed together in a particular, consistent manner. To express general problems of interest to Bayesian statistics it is thus desirable to have not only a well-defined representation for probability distributions, but also a well-defined method for manipulating them or otherwise putting them together.

To get around the first problem there is the 'kitchen sink' approach to representation, in which one represents a probability distribution by a unified bag or object of individual characterizations: sampling functions, density functions, cumulative density functions, and so on. This approach is effective for some applications — note the R package *distr* [Ruckdeschel et al., 2006], for example — but it still suffers from the problem of enforcing well-defined composition of distributions. An object representation in the 'object-oriented' sense does not immediately expose any opportunities for combining or composing general distributions together such that they are guaranteed to obey certain desirable laws. Not only that, but any definition of composition must be implemented individually for each characterization, and there is typically no guarantee that the composition is consistent across representations in any sense.

So what is desirable is to go a step further — to encode probability distributions by way of some consistent data structure (as in the kitchen sink approach, perhaps),

but also define rigorous and exclusive ways by which they can be manipulated and composed together in a well-defined way. The ideal case would be to specify a *general* framework for encoding varied representations of probability distributions, such that any given representation is guaranteed to obey certain important laws.

This chapter develops such a representation.

2.2.1 Domain Specific Languages and Haskell

The combination of data structure and facilities for working with it constitute the basics of a *domain-specific language* (DSL) — a collection of object representations, combinators for working with them, and functionality for interacting with or querying them. This dissertation discusses building, manipulating, and composing distributions at the language level, such that probability distributions (and, later, related concepts) are treated as fundamental 'first-class citizens' with supporting frameworks built around them. It demonstrates that such languages can be particularly useful for formulating and solving problems in Bayesian statistics.

DSLs are useful tools in that they grant a user significant power to express problems within some domain, while also significantly limiting the expressiveness of the language *precisely* to that domain. This limited scope makes the language easier to use, implement, and interpret: a domain-specific language such as SQL[1], for example, has no need to implement general facilities for making exotic network calls (or indeed, implementing a language for probability distributions). It focuses entirely on the domain of building structured query expressions.

DSLs also tend to be *declarative*, in that the programmer typically specifies 'what something is' rather than 'how to do it'. Queries in SQL are declarative, for

[1] A family of popular database querying languages.

example:

```
SELECT columns FROM table WHERE predicate
```

The query describes exactly what is desired, rather than how the database's indexes should be resolved to extract the corresponding data. This declarative emphasis is part of what makes DSLs work; users of the language can express their *intents*, without needing to provide the gory details of how they are achieved. A well-designed DSL allows users to formulate and solve problems in their domain at a relatively high level of abstraction.

DSLs have already proven themselves useful in Bayesian statistics in particular. BUGS [Lunn et al., 2000], JAGS [Plummer, 2015a], and Stan [Stan, 2013] are three popular limited languages for building Bayesian models and performing inference on them. The recent increased research effort in probabilistic programming has also led to the development of various other experimental languages, including Hakaru [Hakaru, 2014] and Venture [Mansinghka et al., 2014].

A DSL can be implemented on its own, with a standalone compiler and toolchain. But it can also be implemented inside some other host language, a technique called *embedding*. By using a host language to embed the DSL, one reaps a number of benefits, namely that the existing compiler, toolchain, and library ecosystem of the host language can be used in tandem with the embedded language. Implementing an industrial quality standalone compiler can be a demanding task, so recycling an existing compiler infrastructure is often desirable. An embedded DSL, or EDSL, typically has a lower startup cost than implementing a compiler for a standalone language. There are downsides as well — error messages in EDSLs must rely heavily on the host language, for example — but they are particularly useful for smaller, minimalist-style implementations.

Haskell [Marlow (editor), 2010] is a particularly suitable host language for EDSLs [Gill, 2014]. It has a sophisticated and well-developed compiler infrastructure in

the *Glasgow Haskell Compiler*, or GHC, and a wealth of general-purpose libraries available in its ecosystem. There exist robust tools for: building, packaging, and distributing libraries; benchmarking and profiling runtime execution; network programming; parallel and concurrent programming, and so on.

Haskell is a *purely-functional* programming language. To tackle the last part of that definition first: *functional programming* is a programming paradigm in which the central computational element is the evaluation of functions or *expressions*. This can be contrasted to *imperative* programming, in which the central computational elements are *statements* that change a program's internal state. Common functional programming languages include Lisp, Erlang, and the ML family of languages, while common imperative languages include C, C++, and Java. Most languages contain at least limited support for both paradigms.

Imperative languages rely on *mutation* or *destructive updating* for control flow and many calculations. For example, a simple 'for' loop consists of a single piece of mutable data — a counter — that is repeatedly mutated to perform some overall calculation. Most existing functional languages, such as Lisp and the ML family, also permit (and make use of) destructive updates.

A function contains a *side effect* if it modifies some program state in addition to returning some value. Side effects can be common in programs written in C, Java, Python, R, and so on, where *global* variables — variables that are in the global lexical scope of a program — tend to find heavy use. For example, a common idiom in statistical work is to use a global 'number of accepts' variable for keeping track of proposals accepted while running an MCMC routine. Typically the same function that accepts a proposed move in the chain also increments the counter, performing a side effect.

A *purely functional* language is characterised by a lack of language support for both 'unmanaged' mutation and side effects. In a purely functional language, functions may only return a value, and all data is immutable. A *pure function*

— like a typical function one deals with in maths — has the property that it will always return the same value when given the same inputs. Effects and mutation are still possible in purely-functional languages, but they must be *managed* in a particular way. A program that performs effects must in some sense be explicit about what effects it performs (we'll see examples of this later in the chapter).

But most importantly for the topic at hand, Haskell supports a number of desirable language constructs for doing 'language engineering'; the ability to write *higher-order functions*, a strong system of *algebraic data types*, support for *parametric polymorphism*, and useful, generic *typeclasses* constitute a powerful set of tools for building compilers or constructing EDSLs.

The following sections outline these techniques in some detail, providing a brief introduction to concepts in programming languages, functional programming, and type systems, in addition to an overview of Haskell syntax. For a rigorous and comprehensive treatment of types and programming languages, see the excellent and seminal Pierce [2002]. For an introduction to programming in Haskell, Lipovača [2011] is a popular, readable, and enjoyable text.

2.3 Algebraic Data Types

2.3.1 Abstract Terms and Types

A programming language is typically defined by its *formal grammar*. The grammar specifies the *terms* (or *expressions*) of a language — the collection of legal syntactic constructs that can be expressed in the language proper. Consider a very simple language for adding integers, for example. It can be defined by the following grammar, denoted in *Backus-Naur Form*, or BNF [Grune and Jacobs, 2008]:

```
<addition> ::= lit <int>
             | add <addition> <addition>
```

This grammar (where integers are taken to be primitive) is a specification of the language — though it is not a *total* specification since it does not define semantics. Terms in this language are either *literals*, represented by 'lit <int>', or the *addition* of terms, represented by 'add <addition> <addition>'. A valid program in this language could be 'add (lit 1) (lit 1)', representing the addition of two literals, or even just something like the trivial 'lit 0'.

Like all nontrivial grammars, this one has a familiar *recursive* structure that is defined in terms of itself. The structure of 'addition' terms is such that we can write programs like:

```
add (add (lit 1) (add (lit 1) (lit 0)))
  (add (lit 0) (add (lit 1) (lit 2)))
```

in which 'add' expressions can contain other 'add' subexpressions.

A *type* is a metavariable for tagging terms in a programming language. Types T are typically assigned to terms t via a binary relation ':', so that we can denote 'term t has type T' by $t : T$. Types are important for the *static analysis* of programming languages; by analyzing types, we can rule out certain pathological programs without ever attempting to evaluate them [Pierce, 2002]. As an example, consider adding some literal Boolean terms to the toy language for addition from above:

```
<augmented> ::= lit <lit>
              | add <augmented> <augmented>

<lit> ::= true
```

```
        | false
        | <int>
```

In this language it is possible to write programs like 'add (lit true) (lit 1)' which may not have any meaningful semantics. By annotating terms with types, we can rule these programs out *statically*, i.e. without needing to actually evaluate them. We can define a set of types:

```
<T> ::= Boolean
      | Integer
```

as well as a set of typing relations:

$$\text{lit true} : \text{Boolean} \qquad \text{lit false} : \text{Boolean} \qquad \text{lit <int>} : \text{Integer}$$

$$\frac{t_0 \ : \ \text{Integer} \qquad t_1 \ : \ \text{Integer}}{\text{add}\, t_0\, t_1 \ : \ \text{Integer}}$$

where the bottom expression reads "if the terms t_0 and t_1 both have type Integer, then the term 'add t_0 t_1' also has type Integer". Any expression for which a type cannot be derived by these relations — for example 'add t_2 t_3' for t_2 : Boolean and t_3 : Int — has no meaningful semantics in the language. An algorithm that verifies a set of typing relations is called a *type checker*, and serves to rule out invalid programs without needing to evaluate them (typically when the program is *compiled*). A type checker will typically report a *type error* for an expression that is not well-typed.

Types are regularly (if informally) used in mathematical specifications as well, usually to denote the set that an element belongs to, or the sets that make up the domain and range (or *codomain*) of a function. Consider a one-dimensional Gaussian density f, for example; the statement $\forall x \in \mathbb{R}.\, f(x) \in \mathbb{R}$ ascribes a typing relation that can be interpreted as 'for any x with type $\mathbb{R}$, $f(x)$ has type $\mathbb{R}$', where the types denote the sets that the each term belongs to. Equivalently,

we could write $f : \mathbb{R} \to \mathbb{R}$ to denote that f takes an argument having type $\mathbb{R}$ and returns an value having the same type.

2.3.2 Terms and Types in Haskell

Haskell is a *statically typed* programming language: just like the previous abstract treatment, every term in Haskell has a type, and these types are checked statically — before any program is actually evaluated. To ascribe types to terms, Haskell expressions are typically annotated like so:

```
add :: Int -> Int -> Int
add a b = a + b
```

The top line is a *type signature*, indicating that the 'add' function has two arguments, each of type 'Int', and that it returns a value with type 'Int'. Haskell has a *Hindley-Milner* or *Damas-Milner* type system [Milner, 1978] that is capable of *inferring* types from arbitrary expressions. As a result, type signatures like the above are rarely required, but are considered good practice to add to top level definitions. We can ask for the inferred type of arbitrary Haskell expressions from within GHCi, the standard Haskell interpreter, using ':t':

```
> :t "hello"
"hello" :: String
```

An *algebraic data type* (ADT) is a type built from other types using the two logical algebraic operations 'and' and 'or'. ADTs are pervasive in Haskell, and prove to be useful for modelling generic composite types. An ADT is declared by the 'data' keyword; the following ADT declaration defines a type for the toy 'addition' language from Section 2.3.1:

```
data Addition =
    Lit Int
  | Add Addition Addition
```

Note that it reads almost exactly like the BNF for the grammar presented previously. This ADT is constructed using both a logical 'or' — denoted by the pipe operator '|' — as well as a several logical 'and' operations denoted by whitespace. It can be read as 'the Addition type is defined by **either** the *data constructor* Lit *and* the *type constructor* Int **or** the data constructor Add **and** the type constructor Addition **and** the type constructor Addition'. This ADT is also defined *recursively*, just as was the case in the formal grammars from Section 2.3.1.

Due to their algebraic properties, types formed from logical 'or' operators are often called *sum types*, while types formed from logical 'ands' are known as *product types*.

2.3.3 A Proto-Representation For Probability Distributions

As an intermission of sorts, we can demonstrate that we're able to define probability distributions on values having sum or product types. In the most trivial cases we can take a type like 'Double' or 'Int' and define a probability density or mass function over its values respectively, but the structure of more general algebraic types pose no difficulty either. Consider the following 'Group' sum type corresponding to a collection of groups, as well as a function corresponding to a categorical distribution over values with that type:

```
data Group = A | B | C

categorical :: Group -> Double
categorical A = 0.1
```

```
categorical B = 0.7
categorical C = 0.2
```

The 'categorical' mass function takes values of type 'Group' and assigns to them a probability, represented by the 'Double' numeric type that corresponds to double-precision floating point numbers. For a product type, we can consider:

```
data R3 = R3 Double Double Double

standardGaussian3d :: R3 -> Double
standardGaussian3d (R3 x y z) = gauss x * gauss y * gauss z where
  gauss a = 1 / sqrt (2 * pi) * exp (negate (a ^ 2) / 2)
```

Here 'R3' corresponds to $\mathbb{R}^3$, and 'standardGaussian3d' assigns a standard normal density to values with type 'R3'. Product types correspond to Cartesian products of types, which have a natural mapping to values over n-dimensional spaces.

We can also define more complex distributions over general ADTs that consist of both sums and products. Consider the following type that assigns probability density over values in a union space:

```
data Union =
    A R3
  | B R3

density :: Union -> Double
density (A r3) = 0.9 * standardGaussian3d r3
density (B r3) = 0.1 * standardGaussian3d r3
```

Notice that the algebraic properties of 'Union' mean that 'density' transcribes the laws of probability verbatim; for $R_3 \in \mathbb{R}^3$ we have that $P(A \cap R_3) =$

$P(A)P(R_3|A)$, $P(B \cap R_3) = P(B)P(R_3|B)$, and $P(\text{Union}) = P(A) + P(B)$, since the intersection of A and B is empty.

Again, these examples fall back on the method of representing distributions via densities. We can ascribe distributions to values of various types using densities, but we have not gone much further towards building a language for working with them more generally.

2.4 Parametric Polymorphism and Typeclasses

All the algebraic data types we've seen thus far have been built out of *interpreted base types* [Pierce, 2002]; primitive, concrete types like 'Int' or 'Bool' put together as abstract sums and products. We can also define more general *parameterized types* which make use of uninterpreted *type variables*.

For example, we could create the following ADT that 'holds' two other general types denoted by type variables:

```
data Pair a b = Pair a b
```

Here the type variables a and b are used to denote arbitrary other types. Note that they appear on either side of the 'equals' operator in the data declaration, which can be read as 'for every a and b the type Pair a b is defined by the data constructor Pair, the type constructor a, and the type constructor b'.

The type constructor 'Pair' defines an infinite family of types 'Pair a b'. For example, the following values x and y have different types constructed by 'Pair':

```
> let x = Pair (1 :: Int) (”hello” :: String)
> :t x
```

```
x :: Pair Int String
> let y = Pair x True
> :t y
y :: Pair (Pair Int String) Bool
```

The 'Pair a b' type is said to be *polymorphic*, and by virtue of having type parameters a and b it is an example of *parametric polymorphism*. A type system featuring parametric polymorphism is very useful to have in a statically-typed programming language: it enables *generic programming*, in that a function written once can work over many types. For example, consider the following function that plucks the first element out of a 'Pair':

```
pluck :: Pair a b -> a
pluck (Pair x _) = x
```

The 'pluck' function can be used for any value having type 'Pair a b', no matter what a and b are. There is no need to write one 'pluck' function that works for the 'Pair Int Double' type, another for the 'Pair Bool String' type, and so on. Parametricity enables various 'free theorems' that must always hold for any implementation of a given function [Wadler, 1989]. Consider again the type signature for 'pluck' on its own, for example:

```
pluck :: Pair a b -> a
```

The general nature of the polymorphic type constrains the possible legal implementations of the function so much so that it can only do a single thing: return the first value of the 'Pair'. There are *no other* legal implementations that will pass a type checker.[2]

[2] This disregards so-called 'bottom' values whose details are unnecessary here.

It is useful to be able to restrict the type parameters of 'Pair a b' in a function like 'pluck' to belong only to certain collections of types, called *typeclasses*. By restricting the allowable types of a or b, for example, we expand the number of legal implementations of 'pluck', typically by quite a large amount.

A typeclass is a collection of types that must all implement some particular functions. A simple example is the 'Eq' typeclass, which allows values of a type in that class to be compared for equality:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

The above typeclass definition states that for any type a that is a member of the 'Eq' typeclass, values having that type can be compared for equality. Note that this was not possible for values having type 'Group' defined in Section 2.3.3. If we try to compare equality of the values 'A' and 'B' in GHCi, for example, we receive the following error:

```
> A == B
    No instance for (Eq Group) arising from a use of '=='
    In the expression: A == B
```

We can redefine 'Group' and automatically *derive* an instance of the 'Eq' typeclass for it as follows:

```
data Group = A | B | C
  deriving Eq
```

This lets us compare values for equality as one would expect:

```
> A == B
False
```

There are many important standard typeclasses in Haskell, with 'Eq' just being a single example. Other important typeclasses include 'Ord', for types of values that have a well-defined ordering, or 'Show', for types of values that have some sort of textual representation.

The following subsections describe three foundational typeclasses that we will throughout the dissertation: the 'Functor', 'Applicative', and 'Monad' typeclasses. We will derive and demonstrate some properties of these structures more formally in Chapter 3, but here we just present a more gentle introduction.

2.4.1 The Functor Typeclass

The 'Functor' typeclass represents a class of types that can be 'mapped over' in some sense. Instances of 'Functor' (called *functors*) must implement a single function — called 'fmap' — which is defined as follows:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

'fmap' is an example of a *higher-order function*. It has two arguments: a function of type $a \to b$ and a value of type fa. Intuitively, the functor f can be pictured as a container of sorts that holds elements of type a. 'fmap' applies its first argument — the function of type $a \to b$ — to the elements of the container, resulting in a value of type fb. The 'shape' of the container is unchanged, but it now holds elements of type b instead of a. Instances of the Functor typeclass must satisfy the following two laws (the 'functor laws'), which can't be automatically verified by the compiler:

```
fmap id      = id                -- identity
fmap (g . h) = fmap g . fmap h   -- homomorphism
```

As an example, consider one of the most fundamental data types in Haskell, the *list*. It can be defined abstractly as

```
data List a =
    Prepend a (List a)
  | Empty
  deriving (Eq, Show)
```

A list is a polymorphic type that is either empty or is the prepending of a value of type a onto another list. Thus the following examples are lists of type '$\forall a$ List a', 'List Int', and 'List Char' respectively:

```
> let foo = Empty
> let bar = Prepend 1 Empty
> let baz = Prepend 'h' (Prepend 'i' (Prepend '!' Empty))
```

The lists are containers that each hold values of a certain type. We can define a 'Functor' instance for generic lists as follows:

```
instance Functor List where
  fmap _ Empty         = Empty
  fmap f (Prepend x l) = Prepend (f x) (fmap f l)
```

Intuitively, 'fmap' applies the function 'f' to each element of the list. It does so by applying 'f' directly to the head of the list and recursively calling itself on the list's tail.

Since the functor laws can't be verified by the compiler, we can use *equational reasoning* to verify them ourselves. For the above instance, for example, we can establish that:

```
fmap id Empty
  = Empty                          -- functor instance

fmap id (Prepend x l)
  = Prepend (id x) (fmap id l)
  = Prepend x (fmap id l)          -- identity function
  = Prepend x l                    -- inductive hypothesis
```

so that 'fmap id = id', verifying the first functor law. To verify the homomorphism law, we have:

```
fmap (f . g) Empty
  = Empty

(fmap f . fmap g) (Prepend x l)
  = fmap f (fmap g (Prepend x l))
  = fmap f (Prepend (g x) (fmap g l))          -- functor instance
  = Prepend (f (g x)) (fmap f (fmap g l))      -- functor instance
  = Prepend ((f . g) x) ((fmap f . fmap g) l)  -- defn. of composition
  = Prepend ((f . g) x) (fmap (f . g) l)       -- inductive hypothesis
  = fmap (f . g) (Prepend x l)                 -- functor instance
```

so that 'fmap f . fmap g = fmap (f . g)', as required. We won't verify any other laws in this chapter, but make use of the equational reasoning technique in the sequel.

The functor instance enables us to use 'fmap' to change the contents of a given list without changing the overall structure of the list itself. We can transform the 'bar' and 'baz' examples from the previous block as follows:

```
> fmap (+ 1) bar
Prepend 2 Empty
```

```
> let f a | a == 'h' = 1 | a == 'i' = 2 | otherwise = 3
> fmap f baz
Prepend 1 (Prepend 2 (Prepend 3 Empty))
```

In practice, lists are of such importance in Haskell that they have a special built-in syntax denoted by square brackets, '["like", "this"]'. Additionally, Haskell's built-in String type is defined in terms of lists of characters, such that we can write the following in place of the previous definition for 'bar':

```
> let quux = "hi!"
> fmap f quux
[1, 2, 3]
```

2.4.2 Another Proto-Representation

That a functor's 'contents' can be changed while leaving its overall structure invariant is a very useful property.

As an example closer to the topic at hand, consider the following container type that can be used to represent a discrete probability distribution:

```
data Distribution a = Distribution [(a, Rational)]
  deriving Show
```

The 'Distribution a' type is simply a wrapper around a list of pairs containing a value of type a and a rational number. We can use it to implement arbitrary categorical distributions. One of the simplest examples is the following Bernoulli$(2/3)$ distribution:

```
> let bernoulli = Distribution [(0, 1 / 3), (1, 2 / 3)]
```

The type of 'bernoulli' here is 'Distribution Int', indicating that it is a probability distribution over the integers. The following typeclass instance demonstrates that 'Distribution' is a functor:

```
instance Functor Distribution where
  fmap f (Distribution vs) = Distribution (fmap g vs) where
    g (v, p) = (f v, p)
```

We can exploit the fact that 'Distribution' is a functor in order to transform its support while leaving its mass structure invariant. For example, we can transform the support from $\{0, 1\}$ to the Boolean domain $\{False, True\}$:

```
> let convert x | x == 0 = False | otherwise = True
> fmap convert bernoulli
Distribution [(False, 1 % 3), (True, 2 % 3)]
> :t fmap convert bernoulli
Distribution Bool
```

The invariance property doesn't just hold for probability distributions of type 'Distribution Int': parametricity and the Functor instance ensure the free theorem that it holds for *any* type a. That is, we can use 'fmap' to transform the support of any valid probability distribution represented by this type and be guaranteed that we receive a valid probability distribution in return — completely independent of whatever that support may be.

The functor instance will ensure that the correct density structure is preserved even under 'collapsing' transformations to the support. Consider the distribution

```
example = Distribution [(0, 1 / 3), (1, 1 / 3), (2, 1 / 3)]
```

for example, and then consider using 'fmap' to apply the following function to it:

```
collapse x = if x <= 1 then 1 else 2
```

Clearly 'collapse' will decrease the size of the support by one element. However the corresponding mass structure is preserved as expected:

```
> fmap collapse example
Distribution [(1, 1 % 3), (1, 1 % 3), (2, 1 % 3)]
```

2.4.3 The Applicative Typeclass

Closely related to a functor is an *applicative functor* (or just *applicative*), which is what instances of the 'Applicative' typeclass are called. The typeclass itself can be defined as follows:

```
class Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Instances of 'Applicative' implement two functions: 'pure', and the infix operator '(<*>)', which we will alias as 'apply' in this dissertation. The 'pure' function takes a value of type a and simply puts it in the applicative, yielding a value of type fa. The 'apply' function is much like 'fmap' from the Functor typeclass; it takes as arguments function from $a \to b$ *itself* wrapped in an applicative f, plus a value of type fa, and returns a value of type fb. Every applicative functor is itself a functor by construction, as 'fmap f u' is equivalent to 'pure f <*> u' [McBride and Paterson, 2008]. Instances of the Applicative typeclass must satisfy the following laws, called the 'applicative laws':

```
pure id <*> x            =  x                    -- identity
pure f <*> pure x        =  pure (f x)           -- homomorphism
```

```
f <*> pure x                =  pure (\g -> g x) <*> f  -- interchange
pure (.) <*> f <*> g <*> x  =  f <*> (g <*> x)        -- composition
```

To illustrate applicative functors, consider Haskell's 'Maybe' data type, defined as follows:

```
data Maybe a =
    Just a
  | Nothing
```

'Maybe' has the following Applicative instance:

```
instance Applicative Maybe where
  pure x             = Just x
  Just f <*> Just x  = Just (f x)
  _ <*> _            = Nothing
```

We can use it to apply functions wrapped in 'Just' constructors to other values wrapped in 'Just' constructors. The following demonstrates this using Haskell's built-in 'succ' function, which returns the successor of a provider value:

```
> let f = Just succ
> f <*> Just 0
Just 1
> f <*> Nothing
Nothing
```

Instances of Applicative also have a function 'liftA2' defined for them by default, where 'liftA2' is a little like 'apply' but for functions with two arguments:

```
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = fmap f <*> a <*> b
```

Applicative functors can be used to implement a *different* typeclass instance: namely, the 'Num' typeclass, which provides the ring operations of addition, subtraction, and multiplication. The minimal definition of 'Num' is as follows:

```
instance Num a where
  (+)         :: a -> a -> a
  (-)         :: a -> a -> a
  (*)         :: a -> a -> a
  abs         :: a -> a
  signum      :: a -> a
  fromInteger :: Integer -> a
```

Any applicative functor over a type a, where a itself is an instance of 'Num' can automatically be made an instance of 'Num', as follows:

```
instance (Num a, Applicative f) => Num (f a) where
  (+)         = liftA2 (+)
  (-)         = liftA2 (-)
  (*)         = liftA2 (*)
  abs         = fmap abs
  signum      = fmap signum
  fromInteger = pure . fromInteger
```

Now, return to our example probability distribution type defined in Section 2.4.1. It is an instance of 'Applicative', though we will delay providing its implementation until the next section. By virtue of being an applicative functor, the 'Distribution' type can automatically be made an instance of 'Num' whenever its support is itself a numeric type.

With a 'Num' instance in play, we can 'add' any two such probability distributions together, like so:

```
> let bernoulli = Distribution [(0, 1 / 3), (1, 2 / 3)]
> let discrete  = Distribution [(0, 1 / 5), (1, 2 / 5), (2, 2 / 5)]
> let convolved = bernoulli + discrete
> convolved
Distribution [
    (0, 1 % 15), (1, 2 % 15)
  , (2, 2 % 15), (1, 2 % 15)
  , (2, 4 % 15), (3, 4 % 15)
  ]
```

As we'll see in Chapter 3, abstract addition in this context corresponds to a *convolution* of probability distributions. The resulting distribution's support is expanded beyond that of 'bernoulli' or 'discrete' to cover an additional point, 3, and its mass structure is reweighted appropriately. The resulting distribution has the majority of its mass ($6/15$) at the point 2.

Note that while 'fmap' always leaves a Functor's structure invariant, 'apply' is able to alter it. We used 'fmap' in Section 2.4.1 to change a distribution's support without touching its mass structure, but 'apply' can alter *both* the support and the mass structure. This phenomenon is known as an *effect* [McBride and Paterson, 2008].

Like the case of the Functor instance, values of any probability distribution type defined so as to implement an Applicative instance can be convolved with other distributions of the same type for free. What's more, abstract notions of subtraction and multiplication are *also* provided by the 'Num' typeclass, where the behaviour of these operators correspond to their equivalents on random variables. For example, if $X \sim f$ and $Y \sim g$ are random variables, then $X - Y \sim f - g$ and $XY \sim fg$.

Like the corresponding case for 'fmap', this property is guaranteed to hold for *any* probability distribution that is also an applicative functor; convolution (and

friends) of well-defined probability distributions is guaranteed to return other well-defined probability distributions.

2.4.4 The Monad Typeclass

Haskell is well-known for its 'Monad' typeclass, which proves to be notoriously difficult for many newcomers to understand. Instances of 'Monad' (called *monads*) are in some sense 'restricted' applicative functors; every monad is an applicative functor, but the converse is not necessarily true. The typeclass is defined as follows:

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

Monads must implement two functions: 'return', which functions identically to Applicative's 'pure', and the operator $\gg=$, called *bind*. Like the Functor and Applicative typeclasses, instances of Monad must satisfy the following monad laws:

```
return x >>= f  = f x                      -- left-identity
m >>= return    = m                        -- right-identity
(m >>= f) >>= g = m >>= \x -> (f x >>= g)  -- associativity
```

The 'bind' operator is what distinguishes a monad from an applicative functor. In the case of an applicative, 'apply' sequences effectful computations together in a *context-free* manner. The monadic bind operator on the other hand permits effectful computations to be combined in a *context-sensitive* manner, such that the result of one effect can determine whether or not to apply another. The bind operator gets its namesake as it 'binds' the result of an effectful or structure-altering computation to a name that can be referred to later.

The 'Maybe' type introduced in the previous section is also a monad, and it has the following typeclass instance:

```
instance Monad Maybe where
  return        = Just
  Just x  >>= f = f x
  Nothing >>= _ = Nothing
```

The 'Maybe' type and its monad instance describe computations that can fail. For example, consider the following three (contrived) functions:

```
positive :: Int -> Maybe Int
positive x = if x > 0 then Just x else Nothing

divisibleByThree :: Int -> Maybe Int
divisibleByThree x = if x `mod` 3 == 0 then Just x else Nothing

divisibleByFive :: Int -> Maybe Int
divisibleByFive x = if x `mod` 5 == 0 then Just x else Nothing
```

We can use these functions to validate a number as being positive, divisible by three, and divisible by five by sequencing them together using the monad instance:

```
validate :: Int -> Maybe Int
validate x =
  positive x >>= \foo ->
  divisibleByThree foo >>= \bar ->
  divisibleByFive bar >>= \baz ->
  return baz
```

Notice how we bind the result of any particular combination to a name; the result of applying 'positive' to an integer is bound to the name 'foo', which is then used as input to the function 'divisibleByThree', and so on. The function returns the wrapped integer if it is positive and divisible by both three and five, and 'Nothing' if any of the validations fail, demonstrated as follows:

```
> validate 0
Nothing
> validate 3
Nothing
> validate 5
Nothing
> validate 15
Just 15
```

Monads have a special place in Haskell, and are used to implement I/O, error-handling, concurrency, and more. Due to their special stature, Haskell includes a custom syntax for writing monadic functions, called *do-notation*. We can write the 'validate' function like so:

```
validate :: Int -> Maybe Int
validate x = do
  foo <- positive x
  bar <- divisibleByThree foo
  baz <- divisibleByFive bar
  return baz
```

In a so-called do-block, monadic binds are denoted by an arrow rather than the usual equals sign. The resulting program has a familiar-looking *imperative* style, proceeding sequentially from top to bottom.

Although we won't dwell on this point in much detail, monads can also be stacked together using *monad transformers*. A monad transformer stack is just a monad with multiple 'layers', where each layer can be accessed by using one or more 'lift' functions. The 'Maybe' type can be stacked over the 'IO' type, as in the following example (where the validation functions have been adjusted to work with the new type):

```
validateAndPrint :: Int -> MaybeT IO Int
validateAndPrint x = do
  foo <- positiveT x
  bar <- divisibleByThreeT foo
  baz <- divisibleByFiveT bar
  lift (print baz)
  return baz
```

Here we use the 'Maybe' monad to do validation and the 'IO' monad to print the value of 'baz' to stdout. The interesting thing is that if 'baz' ever fails to validate, the result won't be printed.

Monad transformers are used throughout this dissertation but a deep understanding of them is not required. They are essentially a way to build composite monads from primitive parts, and that abstract intuition is sufficient.

Along with its Functor and Applicative instances, the discrete probability distribution type from Section 2.4.1 is also a monad:

```
instance Monad Distribution where
  return x = Distribution [(x, 1)]
  Distribution xs >>= f = normalize
    (Distribution [
        (y, p * q)
      | (x, p) <- xs
```

```
      , (y, q) <- values (f x)
      ])
```

where the 'normalize' and 'values' functions are defined as follows:

```
normalize :: Distribution a -> Distribution a
normalize d = Distribution (fmap (second (/ total)) vals)
  where vals  = values d
        total = foldr ((+) . snd) 0 vals

values :: Distribution a -> [(a, Rational)]
values (Distribution vs) = vs
```

The 'normalize' function ensures that a distribution's probability mass sums to one, while the 'values' function simply extracts the distribution's support and mass structure. Note also that the 'normalize' function also makes use of the 'second' helper function, which can be imported via the 'Control.Arrow' module included in the Haskell standard library.

The two monadic functions 'return' and 'bind' have familiar interpretations in the context of probability theory. 'return' takes a value x of type a and returns a *Dirac distribution* over a, with the entirety of its probability concentrated at x. 'bind' on the other hand is a *marginalizing* operator; it takes a probability distribution P over type a and a function that uses a value of type a to construct a distribution over type b, and produces a distribution over type b by marginalizing P out of the joint distribution.

The 'bind' operator that implements marginalization is exactly what we can use to compose distributions together in a context-sensitive manner. Consider the following pairs of discrete distributions, where the second takes a parameter that describes its support:

```
> let num = Distribution [(10, 1 / 3), (11, 1 / 3), (12, 1 / 3)]
> let discrete n = Distribution [
      (n - 1, 1 / 5)
    , (n, 2 / 5)
    , (n + 1, 2 / 5)
    ]
```

We can compose ‘num’ and ‘discrete’ together by using ‘num’ as the input parameter to ‘discrete’, producing the following compound distribution:

```
> let compound = num >>= discrete
> compound
Distribution [
    (9,  1 % 15), (10, 2 % 15)
  , (11, 2 % 15), (10, 1 % 15)
  , (11, 2 % 15), (12, 2 % 15)
  , (11, 1 % 15), (12, 2 % 15)
  , (13, 2 % 15)
  ]
```

The ‘num’ distribution has been marginalized into ‘discrete’ by using the monadic bind operator. Both the support and its density structure of ‘discrete’ have been transformed; the support ranges from 9 to 13, and the associated probabilities are now exclusively either $1/15$ or $2/15$. It’s also worth striking home the point that the compound distribution could be written using do-notation as follows:

```
compound :: Distribution Int
compound = do
  n <- num
  discrete n
```

The 'discrete' distribution takes an integer as a parameter, but 'num' is a *distribution* over integers. The monadic bind allows us to deal with the 'num' distribution as a single integer and feed it into 'discrete', statically ensuring that the result is another distribution.

We aren't limited to just sequencing a single distribution together; since the 'discrete' distribution takes an integer as a parameter and is itself a distribution over integers, we can use it to feed integers into itself by 'looping' some number of times:

```
compounder :: Distribution Int
compounder = num >>= discrete >>= discrete >>= discrete >>= discrete
```

We of course get an appropriately-weighted distribution as in return:

```
> compounder
Distribution [(6,1 % 1875), ... ]
```

The bind operator can be used to sequence any arbitrary directed graph of distributions together. What's more the distribution resulting from a series of monadic binds is familiar to Bayesian statisticians as the *predictive distribution*. If $p(\theta)$ corresponds to a prior distribution and $p(x \mid \theta)$ a likelihood, then binding $p(\theta)$ and $p(x \mid \theta)$ together in that order yields the (prior) predictive distribution $p(x)$. We'll make much more use of this fact later in the dissertation.

To tie up loose ends: recall that in Section 2.4.3 we deferred implementing the applicative instance for 'Distribution'. Since every monad is an applicative functor by construction, it is often easier to implement a Monad instance *first*, and then implement the Applicative instance in terms of that. We can implement the Applicative instance as follows:

```
instance Applicative Distribution where
  pure  = return
  (<*>) = ap
```

Here, ‘ap’ is a function from the ‘Control.Monad’ module that all monads get for free — that is, it can always be implemented in terms of the monadic ‘return’ and ‘bind’ functions, and so a default implementation of it always exists given an existing Monad instance. It turns out to be equal to the applicative ‘apply’ function, and similarly the monadic ‘return’ function is equivalent to the applicative ‘pure’. So we can always hijack the functionality provided by a Monad instance in order to implement the corresponding Applicative instance, though in select cases it can be desirable (in terms of computational efficiency) to implement the Applicative instance directly in terms of non-monadic functions. An Applicative instance derived in this way is guaranteed to be law-abiding so long as the Monad instance obeys the monad laws, so we don’t usually need to prove the applicative laws explicitly.

2.5 Conclusion

2.5.1 An Embedded Language for Probability Distributions

The discrete probability distribution type is a particular representation for probability distributions. It’s fairly limited, but can accurately denote distributions over arbitrary countable supports. The distribution is represented by an explicit enumeration of both the support and associated probability mass at any point. It is a particular ‘object representation’ for probability distributions.

The discrete distribution type is also an instance of the Functor typeclass, meaning that we can use ‘fmap’ to transform the support of any distribution according

to some pointwise mapping function and be statically *guaranteed* to get a valid probability distribution in return. What's more, we are also statically guaranteed that the transformed distribution will have a *structurally similar* density structure to that of the original distribution.

The Applicative instance allows us to apply context-free effects to our distribution type, letting us (for example) combine distributions together using abstract addition, subtraction, and multiplication. That is, we can use *convolution* and its friends in order to produce new distributions from old ones: again having static guarantees that whatever emerges must be a valid probability distribution.

Finally, the monad instance gives us the powerful ability to apply context-sensitive effects to our distribution type. We can glue arbitrary directed graphs of distributions together via an appropriate marginalizing semantics, receiving a valid *predictive distribution* in return. The monad instance also allows us to write simple imperative programs involving distributions using *do-notation*, a special syntax that eliminates the line noise of working with the monadic bind operator $\gg=$.

These elements — a specific kind of data structure and facilities for transforming and combining them — constitute a limited embedded DSL for working with discrete probability distributions in Haskell. Were this implementation to be packaged up in a library, users could fruitfully manipulate, transform, and compose distributions with only a limited knowledge of the Haskell language more broadly.

The construction implemented here is usually distinguished as a *shallow* embedding, in that the DSL is implemented directly in terms of its semantics. This is in contract to a *deeply* embedded DSL in which language constructs are represented by abstract syntax (similar to in a standalone compiler). While typically lacking the same expressive power as deeply-embedded DSLs, shallow embeddings are more than suitable enough for many applications.

The EDSL in this chapter was explored in detail by Erwig and Kollmansberger [2006], who used the Monty Hall problem (amongst other examples) to illustrate its use. That example — particularly concise and beautiful when implemented in an embedded monadic language like this — is expanded on below.

First, the initial distribution over doors. Each door is equally likely to be chosen; one door contains the prize, while the others traditionally contain goats. We can encode this over a boolean type, where 'True' indicates the winning door:

```
doors :: Distribution Bool
doors = Distribution [
    (False, 1 / 3)
  , (False, 1 / 3)
  , (True,  1 / 3)
  ]
```

Now in a program we can represent an initial choice by (monadically) binding 'doors' to a variable. We also need to implement a function for switching, and this is straightforward:

```
switch :: Bool -> Distribution Bool
switch True  = return False
switch False = return True
```

Applying 'switch' to a bound choice returns the predictive distribution over the opposite choice. The entire program looks like this:

```
montyHall :: Distribution Bool
montyHall = do
  choice <- doors
  switch choice
```

It captures the procedural structure of the Monty Hall problem: first we make a choice from the distribution of doors, and then we switch our choice. The resulting distribution has the expected structure.

```
> montyHall
Distribution [(True, 2 % 3), (False, 1 % 3)]
```

If you're paying attention however, it should be clear that we don't need 'switch' to return a *distribution* over booleans. After all, switching is a deterministic choice:

```
switch :: Bool -> Bool
switch True  = False
switch False = True
```

The functorial structure makes it clear that we can just calculate the appropriate distribution by transforming the support by this non-effectful 'switch' function:

```
alternateMontyHall :: Distribution Bool
alternateMontyHall = fmap switch doors
```

```
> alternateMontyHall
Distribution [(True, 2 % 3), (False, 1 % 3)]
```

Using a monadic functional language for dealing with probability distributions lets insights like these fall out naturally; on the other hand, it's more difficult to realize that the Monty Hall problem corresponds to a simple transformation of support when dealing purely with maths, or untyped imperative code.

In the next chapter we'll further explore the monadic structure of probability distributions, as well as produce two shallow EDSLs for working with other concrete representations for them.

2.5.2 Notes

The value of embedding DSLs in Haskell has been notably touched on by Gill [2014] and Löh [2012]. Each author has additionally made important technical contributions to the area, such as the use of *type-safe observable sharing* [Gill, 2009] and *abstract syntax graphs* for deeply-embedded DSLs [Oliveira and Löh, 2013].

Chapter 3

Representing Probability Distributions

> Statistical models are monadic programs.
>
> Tom Nielsen

3.1 Abstract and Contributions

This chapter explores the *structure of probability distributions* under the *Giry monad*, the canonical probability monad that operates on the level of probability measures.

From categorical and measure-theoretic foundations, we build up an embedded language via a continuation-based implementation of the Giry monad. Probability measures are represented as programs for performing integration, and we

demonstrate that the algebraic structure afforded to them via their functor, applicative, and monad instances allow probabilistic concepts like image measure, product measure, and marginalization to be expressed naturally. The embedded language is demonstrated to be flexible as to the probability distributions it can represent, but exponential complexity in its fundamental operations makes it impractical outside of basic examples.

The primary contributions of this chapter are:

- Novel **probabilistic interpretations** of the Giry monad's algebraic structure. Most significantly, **we characterize image measure by functorial structure** and **product measure by applicative structure**. The functorial structure is demonstrated to be useful for transforming a measure's support while preserving its density structure, and the applicative/product measure structure is demonstrated to be useful for encoding independence between measurable functions, allowing us to express familiar constructs such as measure convolution.
- A novel **characterization of the Giry monad as a restricted continuation monad**. We implement a shallowly-embedded DSL for integration by using a dual interpretation for probability measures, encoding them as self-contained integration procedures that one can 'query' by integrating measurable functions against. We note that this language is structurally equivalent to the 'expectation monad' of Ramsey and Pfeffer [2002] since both are continuation-based encodings of the Giry monad. We develop a number of queries — notably **measure convolution** and recovery of **moment/cumulant-generating and cumulative distribution functions** — over measures defined over varying supports.

3.2 Motivation

The previous chapter built up a simple probability monad by representing a probability distribution as an explicit enumeration of its support and density structure. The functor, applicative, and monad instances of the discrete distribution type are useful for illustrating toy examples like the Monty Hall problem, but that representation is otherwise limited for obvious reasons.

The 'canonical' probability monad is the so-called *Giry monad* [Lawvere, 1962, Giry, 1981] which operates on the level of probability measures — themselves canonical representations of probability distributions. Measure theory is used to define and develop formal probability and is used almost exclusively in theoretical work where rigour is required. One typically proves some result about probability distributions using canonical measure-theoretic constructs, and then extends it to other representations as needed. A probability monad based on measures similarly establishes a canonical probabilistic semantics that can then be extended to other probability monads as required — the discrete probability monad type introduced in Chapter 2, for example.

The measure characterization of a probability distribution is chosen in lieu of things like probability density and mass functions because:

- a probability distribution represented by a probability measure is guaranteed to be well-defined for the measurable space under consideration,
- measures can be defined over abstract spaces in which alternate representations — such as probability mass or density functions, or even cumulative distribution functions — may not necessarily exist, and
- measures treat probability distributions on abstract spaces in a unified fashion, using a single language.

The cost of measure theory, 'that most arid of subjects when done for its own sake' [Williams, 1991], is its relative impenetrableness. Measures can be hard to think about because one must typically be inundated in some excruciatingly technical detail when introduced to them.

At least on a semantic level, though, measures seem to make sense when it comes to implementing a basis representation for a monadic probabilistic programming language. A language based on measures should be 'complete' in some sense, in that the representation must by construction be capable of denoting any valid probability distribution.

In this chapter we'll construct such an embedded monadic language by interpreting the theory and translating it into an implementation of the Giry monad.

3.3 Theoretical Background

It is useful to have a basic categorical language on hand for discussing the notions of functor, monad, and so on and ascribing rigorous probabilistic interpretations to them. In this section, we'll derive the Giry monad from first principles — from its categorical and measure-theoretic foundations — in order to establish this language and lay some theoretical groundwork. Some standard references for background material around the category theory, measure theory, functional analysis, and integration theory presented here are Mac Lane [1971], Awodey [2010], Aliprantis and Border [2006], and Pollard [2001].

3.3.1 Categorical Foundations

A *category* C is a collection of *objects* and *morphisms* between them. If W, X, Y, and Z are objects in C, then $f : W \to X$, $g : X \to Y$, and $h : Y \to Z$

are examples of morphisms. These morphisms can be composed in the obvious associative way, i.e.:

$$h \circ (g \circ f) = (h \circ g) \circ f$$

and there exist identity morphisms that simply map objects to themselves. An *isomorphism* is a morphism for which there exists an inverse — object X and Y are *isomorphic*, denoted $X \cong Y$, if there exist morphisms $i : X \to Y$ and $j : Y \to X$ such that $j \circ i = 1_X$ and $i \circ j = 1_Y$.

A *functor* is a mapping between categories (equivalently, it's a morphism in the category of so-called 'small' categories). The functor $F : C \to D$ takes every object in C to some object in D, and every morphism in C to some morphism in D, such that the structure of morphism identity and composition is preserved. An *endofunctor* is a functor from a category to itself, and a *bifunctor* is a functor from a pair of categories to another category, i.e. of the form $F : A \times B \to C$.

A *natural transformation* is a structure-preserving mapping between functors. So for two functors $F, G : C \to D$, a natural transformation $\epsilon : F \to G$ associates to every object c in C a morphism $\epsilon_c : F(c) \to G(c)$ in D such that for any $f : c \to c'$ in C, the following identity holds:

$$\epsilon_{c'} \circ F(f) = G(f) \circ \epsilon_c.$$

A *natural isomorphism* is a natural transformation for which each ϵ_c, as above, is an isomorphism.

A *monoidal category* C is a category with some additional monoidal structure, namely an identity object I and a bifunctor $\otimes : C \times C \to C$ called the *tensor product*, plus several natural isomorphisms that provide the associativity of the tensor product and its right and left identity with the identity object I.

A *monoidal functor* is a functor between monoidal categories. For monoidal categories $(C, \otimes, I)$ and $(D, \oplus, J)$, a monoidal functor $F : C \to D$ is a functor and associated natural transformations $\phi : F(A) \oplus F(B) \to F(A \otimes B)$ and

$i : J \to F(I)$ that satisfy some coherence conditions that we'll elide here. Notably, if ϕ and i are isomorphisms (i.e. are invertible) then F is called a *strong* monoidal functor. Otherwise, it's called *lax*. A useful special case occurs for endofunctors on a monoidal category $(C, \otimes, I)$, in which one only has $F : C \to C$, $\phi : F(A) \otimes F(B) \to F(A \otimes B)$, and $i : I \to F(I)$ to worry about.

A *monoid* (M, μ, η) in a monoidal category C is an object M in C together with two morphisms (obeying the standard associativity and identity properties) that make use of the category's monoidal structure: the associative binary operator $\mu : M \otimes M \to M$, and the identity $\eta : I \to M$.

Finally, a *monad* is (infamously) a 'monoid in the category of endofunctors', where the monoidal structure is functor composition. So take a category of endofunctors[1] $\mathcal{F}$ whose objects are endofunctors and whose morphisms are natural transformations between them. This is a monoidal category; there exists an identity endofunctor $1_{\mathcal{F}}(F) = F$ for all F in $\mathcal{F}$, plus a tensor product $\otimes : \mathcal{F} \times \mathcal{F} \to \mathcal{F}$ defined by functor composition such that the required associativity and identity properties hold. $\mathcal{F}$ is thus a monoidal category, and any specific monoid (F, μ, η) we construct on it is a specific monad.

3.3.2 Probabilistic Foundations

A *measurable space* $(X, \mathcal{X})$ is a set X equipped with a topology-like structure called a σ-algebra $\mathcal{X}$ that essentially contains every well-behaved subset of X in some sense. A *measure* $\nu : \mathcal{X} \to \mathbb{R}$ is a particular kind of set function from the σ-algebra to the nonnegative real line. A measure just assigns a generalized notion of area or volume to well-behaved subsets of X. In particular, if the total possible area or volume of the underlying set is 1 then we're dealing with a *probability*

[1] One always considers the category of endofunctors *on some particular category* — we sometimes omit this detail for brevity, but it is always implied.

measure. A measurable space completed with a measure, e.g. $(X, \mathcal{X}, \nu)$ is called a *measure space*, and a measurable space completed with a probability measure is called a *probability space*. As shorthand, if we denote a measurable space by $M = (X, \mathcal{X})$ then we can denote a corresponding measure space by (M, ν), for example, such that $((X, \mathcal{X}), \nu)$ is understood to mean $(X, \mathcal{X}, \nu)$.

A *measurable set* is an element of a σ-algebra in a measurable space, while a *measurable function* or *random variable* is a mapping between measurable spaces. Given a 'source' measurable space $(X, \mathcal{X})$ and 'target' measurable space $(Y, \mathcal{Y})$, a measurable function $(X, \mathcal{X}) \to (Y, \mathcal{Y})$ is a map $T : X \to Y$ with the property that, for any measurable set in the target, the inverse image is measurable in the source. Or, formally, for any B in $\mathcal{Y}$, one has that $T^{-1}(B)$ is in $\mathcal{X}$.

The collection of all measurable spaces and measurable functions between them constitutes a category, denoted **Meas**, in which objects are measurable spaces and morphisms are the measurable functions between them.

Meas is a monoidal category. For measurable spaces $M = (X, \mathcal{X})$ and $N = (Y, \mathcal{Y})$ — both objects in **Meas** — one can define the tensor product $M \otimes N$ as the Cartesian product $X \times Y$ equipped with the σ-algebra $\mathcal{X} \otimes \mathcal{Y}$ generated by the measurable sets $A \times B$ for any $A \in \mathcal{X}$ and $B \in \mathcal{Y}$. The identity I is then defined such that $M \otimes I = I \otimes M = M$ for any M an object in **Meas**.

We can also use the monoidal structure of **Meas** to define the concept of *product measure*. For probability spaces (M, μ) and (N, ν), the product measure $\mu \times \nu$ is the unique measure on $M \otimes N$ such that $(\mu \times \nu)(A \times B) = \mu(A)\nu(B)$ for $A \times B$ a measurable set in $M \otimes N$.

Independence is a fundamental probabilistic concept and can be realized in terms of measurable sets, σ-algebras, or measurable functions. For a probability space $(X, \mathcal{X}, \mathbb{P})$, measurable sets A and B are independent if

$$\mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B).$$

Two sub-σ-algebras $\mathcal{A}$ and $\mathcal{B}$ of $\mathcal{X}$ are independent if A and B are independent for *any* A in $\mathcal{A}$ and *any* B in $\mathcal{B}$. And lastly, for measurable functions f and g with codomain equipped with a σ-algebra $\mathcal{B}$ that each generate a sub-σ algebra of $\mathcal{X}$ via:

$$\mathcal{X}_f = \{f^{-1}(B) : B \in \mathcal{B}\}$$
$$\mathcal{X}_g = \{g^{-1}(B) : B \in \mathcal{B}\}$$

we have that f and g are independent if $\mathcal{X}_f$ and $\mathcal{X}_g$ are independent.

For any measurable space M in **Meas**, we can consider the space of all possible probability measures that could be placed on it and denote that $\mathcal{P}(M)$. To be clear, $\mathcal{P}(M)$ is a *space of measures* — that is, a space in which the points themselves are probability measures.

As a probability measure, any element of $\mathcal{P}(M)$ is a function from measurable subsets of M to the interval $[0, 1]$ in $\mathbb{R}$. That is: if M is the measurable space $(X, \mathcal{X})$, then a point ν in $\mathcal{P}(M)$ is a function $\mathcal{X} \to \mathbb{R}$. For any measurable A in M, there naturally exists an 'evaluation' mapping denoted $\tau_A : \mathcal{P}(M) \to \mathbb{R}$ that takes a measure on M and evaluates it on the set A. To be explicit: if ν is a measure in $\mathcal{P}(M)$, then τ_A simply evaluates $\nu(A)$. It 'runs' the measure in a sense; in Haskell, τ_A would be analogous to a function like $\lambda f.fa$ for some a.

This evaluation map τ_A corresponds to an *integral*. If one has a measurable space $(X, \mathcal{X})$, then for any A a subset in $\mathcal{X}$, $\tau_A(\nu) = \nu(A) = \int_X \chi_A d\nu$ for χ the characteristic or indicator function of A (where $\chi(x)$ is 1 if x is in A, and is 0 otherwise). And we can actually extend τ to operate over measurable mappings from $(X, \mathcal{X})$ to $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$, where $\mathcal{B}(\mathbb{R})$ is a suitable σ-algebra on $\mathbb{R}$. Here we typically use what's called the *Borel* σ-algebra, which takes a topology on the set and then generates a σ-algebra from the open sets in the topology (for $\mathbb{R}$ we can just use the 'usual' topology generated by the Euclidean metric). For $f : X \to \mathbb{R}$ a measurable function, we can define the evaluation mapping $\tau_f : \mathcal{P}(M) \to \mathbb{R}$ as $\tau_f(\nu) = \int_X f d\nu$.

We can abuse notation here a bit and just use τ to refer to mappings that evaluate measures over measurable sets or measurable functions depending on context. If we treat $\tau_A(\nu)$ as a function $\tau(\nu)(A)$, then $\tau(\nu)$ has type $\mathcal{X} \to \mathbb{R}$. If we treat $\tau_f(\nu)$ as a function $\tau(\nu)(f)$, then $\tau(\nu)$ has type $(X \to \mathbb{R}) \to \mathbb{R}$. Let $\tau_{\{A,f\}}$ refer to the mappings that accept either measurable sets or functions.

In any case: for a measurable space M, there exists a topology on $\mathcal{P}(M)$ called the *weak-$*$ topology* that makes all the evaluation mappings $\tau_{\{A,f\}}$ continuous for any measurable set A or measurable function f. From there, we can generate the Borel σ-algebra $\mathcal{B}(\mathcal{P}(M))$ that makes the evaluation functions $\tau_{\{A,f\}}$ measurable. The result is that $(\mathcal{P}(M), \mathcal{B}(\mathcal{P}(M)))$ is itself a measurable space, and thus an object in **Meas**.

3.3.3 $\mathcal{P}$ is a Functor

For any M an object in **Meas**, we have that $\mathcal{P}(M)$ is also an object in **Meas**. And if one looks at $\mathcal{P}$ like a functor, one notices that it takes objects of **Meas** to objects of **Meas**. Indeed, one can define an analogous procedure on morphisms in **Meas** as follows. Take N to be another object (read: measurable space) in **Meas** and $T : M \to N$ to be a morphism (read: measurable mapping) between them. Now, for any measure ν in $\mathcal{P}(M)$ we can define $\mathcal{P}(T)(\nu) = \nu \circ T^{-1}$ (this is called the image, distribution, or pushforward of ν under T). For some T and ν, $\mathcal{P}(T)(\nu)$ thus takes measurable sets in N to a value in the interval $[0, 1]$ — that is, it is a measure on $\mathcal{P}(N)$ (see Figure 3.1). So we have that:

$$\mathcal{P}(T) : \mathcal{P}(M) \to \mathcal{P}(N)$$

and so $\mathcal{P}$ is an endofunctor on **Meas**.

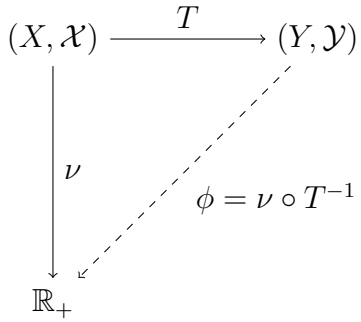


Figure 3.1: Mappings between various spaces. ν is a measure on $(X, \mathcal{X})$ and T is a measurable mapping from $(X, \mathcal{X})$ to $(Y, \mathcal{Y})$. The measure ϕ, defined on $(Y, \mathcal{Y})$, is the pushforward of ν under T.

To check the functor laws, note that for any $M \in$ **Meas** we have:

$$\begin{aligned}\mathcal{P}\left(1_M\right)(\nu) &= \nu \circ \left(1_M\right)^{-1} \\ &= \nu \circ 1_M \\ &= \nu\end{aligned}$$

such that $\mathcal{P}(1_M) = 1_{\mathcal{P}(M)}$, satisfying the identity law. For associativity, note for $T : M \to N$ and $S : N \to P$ that:

$$\begin{aligned}\mathcal{P}(S \circ T)(\nu) &= \nu \circ (S \circ T)^{-1} && \text{(image)} \\ &= \nu \circ \left(T^{-1} \circ S^{-1}\right) && \text{(inverse)} \\ &= \left(\nu \circ T^{-1}\right) \circ S^{-1} && \text{(associativity)} \\ &= \mathcal{P}(S)\left(\nu \circ T^{-1}\right) && \text{(image)} \\ &= \mathcal{P}(S)\left(\mathcal{P}(T)(\nu)\right) && \text{(image)} \\ &= \left(\mathcal{P}(S) \circ \mathcal{P}(T)\right)(\nu) && \text{(composition)}\end{aligned}$$

so that $\mathcal{P}(S \circ T) = \mathcal{P}(S) \circ \mathcal{P}(T)$, as required.

3.3.4 $\mathcal{P}$ is a Monad

What is required to assert that $\mathcal{P}$ is a monad is to define natural transformations μ and η such that $(\mathcal{P}, \mu, \eta)$ is a monoid in the category of endofunctors on **Meas**.

First, the identity. We want a natural transformation η between the identity functor $1_{\mathcal{F}}$ and the functor $\mathcal{P}$ such that $\eta_M : 1_{\mathcal{F}}(M) \to \mathcal{P}(M)$ for any measurable space M in **Meas**. Evaluating the identity functor simplifies things to $\eta_M : M \to \mathcal{P}(M)$.

We can define this concretely as follows. Grab a measurable space M in **Meas** and define $\eta(x)(A) = \chi_A(x)$ for any point $x \in M$ and any measurable set $A \subseteq M$. $\eta(x)$ is thus a probability measure on M — we assign 1 to measurable sets that contain x, and 0 to those that don't. If we peel away another argument, we have that $\eta : M \to \mathcal{P}(M)$, as required. So, η takes points in measurable spaces to probability measures on those spaces. In technical parlance, it takes a point x to the *Dirac measure* at x — the probability measure that places the entirety of its mass at x.

Recall that any category of endofunctors, $\mathcal{F}$, is monoidal, so there exists a tensor product $\otimes : \mathcal{F} \times \mathcal{F} \to \mathcal{F}$ that we can deal with, which here just corresponds to functor composition. We're looking for a natural transformation:

$$\mu : \mathcal{P} \circ \mathcal{P} \to \mathcal{P}$$

which is often written as:

$$\mu : \mathcal{P}^2 \to \mathcal{P}.$$

Take $M = (X, \mathcal{X})$ a measurable space in **Meas** and then consider the space of probability measures over it, $\mathcal{P}(M)$. Then take the space of probability measures *over the space of probability measures* on M, $\mathcal{P}(\mathcal{P}(M))$. Since $\mathcal{P}$ is an endofunctor, this is again a measurable space, and for any measurable subset A of M we again have a family of mappings τ_A that take a probability measure in $\mathcal{P}(\mathcal{P}(M))$

and evaluate it on A. We want μ to be the construct that turns a measure over measures ρ into a probability measure on $\mathcal{P}(M)$.

In the context of probability theory, this kind of semigroup action is a *marginalizing* operator. We're taking the 'uncertainty' captured in $\mathcal{P}(\mathcal{P}(M))$ via the probability measure ρ and smearing it into the probability measures in $\mathcal{P}(M)$.

Take ρ in $\mathcal{P}(\mathcal{P}(M))$ and some A a measurable subset of M. We can define μ as follows:

$$\mu(\rho)(A) = \int_{\mathcal{P}(M)} \tau_A d\rho.$$

Using some lambda calculus notation to see the argument for τ_A, i.e. equating the function $f(x) = y$ with the expression $\lambda x.y$, we can expand the integrals to get the following expression:

$$\mu(\rho)(A) = \int_{\mathcal{P}(M)} \left\{ \lambda\nu. \int_M \chi_A d\nu \right\} d\rho.$$

Notice what's happening here. For M a measurable space, we're integrating over $\mathcal{P}(M)$ the space of probability measures on M, with respect to the probability measure ρ, which itself is a point in the space of probability measures over probability measures on M, $\mathcal{P}(\mathcal{P}(M))$ (see Figure 3.2).

The spaces we're integrating over here are unusual, but ρ is still a probability measure, so when applied to a measurable set in $\mathcal{B}(\mathcal{P}(M))$ it results in a probability in $[0, 1]$. So, peeling back an argument, we have that $\mu(\rho)$ has type $\mathcal{X} \to \mathbb{R}$. In other words, it's a probability measure on M, and thus is in $\mathcal{P}(M)$. And if we peel back another argument, we find that:

$$\mu_M : \mathcal{P}(\mathcal{P}(M)) \to \mathcal{P}(M)$$

so, as required, that

$$\mu : \mathcal{P}^2 \to \mathcal{P}.$$

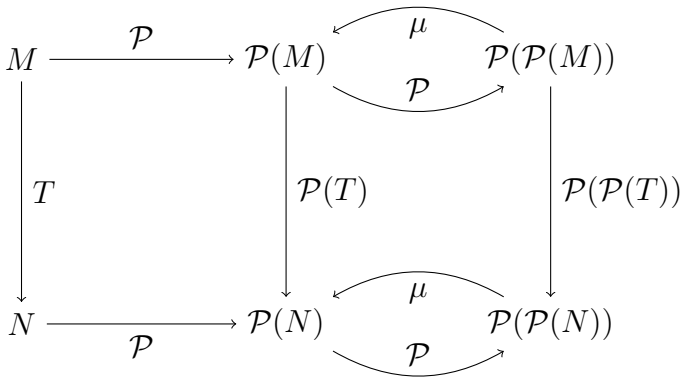


Figure 3.2: Mappings between various spaces. Each application of $\mathcal{P}$ to a measurable space brings it to the space of measures over itself, while applying $\mathcal{P}$ to a measurable mapping brings it to a mapping between the space of measures on each. The core monadic 'join' operator μ normalizes a tower of spaces of measures by one level.

It's also worth noting that we can overload the notation for μ in the same way we did for τ, i.e. to supply measurable functions in addition to measurable sets:

$$\mu(\rho)(f) = \int_{\mathcal{P}(M)} \left\{ \lambda\nu. \int_M f d\nu \right\} d\rho.$$

Combining the three components, we get $(\mathcal{P}, \mu, \eta)$, the canonical Giry monad.

Recall that in Haskell, when we're dealing with monads we typically use the bind operator $\gg\!=$ instead of manually dealing with the functorial structure and μ (called 'join'). Bind has the type:

$$\gg\!= : \mathcal{P}(M) \to (M \to \mathcal{P}(N)) \to \mathcal{P}(N)$$

and we can define $\gg\!=$ for the Giry monad like so:

$$(\rho \gg\!= g)(f) = \int_M \left\{ \lambda m. \int_N f dg(m) \right\} d\rho.$$

Here ρ is in $\mathcal{P}(M)$, g is in $M \to \mathcal{P}(N)$, and f is in $N \to \mathbb{R}$, so note that we potentially simplify the outermost integral enormously. It now operates over a *general* measurable space, rather than a space of measures in particular.

We can verify that the Giry monad satisfies the monad laws as follows. Note first that the Dirac measure δ_x at some point x has the property:

$$\begin{aligned}\delta_x(f) &= \int f d\delta_x \\ &= f(x)\end{aligned}$$

so that right-identity is established by:

$$\begin{aligned}\rho \gg\!= \delta &= \lambda f. \int_M \left\{ \lambda m. \int_M f d(\delta_m) \right\} d\rho \\ &= \lambda f. \int_M \{\lambda m. f(m)\}\, d\rho && \text{(Dirac)} \\ &= \lambda f. \int_M f d\rho \\ &= \rho.\end{aligned}$$

Left-identity follows similarly. Given some x, one has that:

$$\begin{aligned}\delta_x \gg\!= g &= \lambda f. \int_M \left\{ \lambda m. \int_N f d(g(m)) \right\} d\delta_x \\ &= \lambda f. \left(\lambda m. \int_N f d(g(m)) \right)(x) && \text{(Dirac)} \\ &= \lambda f. \int_N f d(g(x)) \\ &= g(x)\end{aligned}$$

as required. For associativity, note that for $\rho \in \mathcal{P}(M)$, $g : M \to \mathcal{P}(N)$, and $h : N \to \mathcal{P}(Q)$ we have:

$$\begin{aligned}(\rho \gg\!= g) \gg\!= h &= \left(\lambda f : N \to \mathbb{R}. \int_M \left\{ \lambda m. \int_N f dg(m) \right\} d\rho \right) \gg\!= h \\ &= \lambda f : Q \to \mathbb{R}. \int_M \left\{ \lambda m. \int_N \left\{ \lambda n. \int_Q f dh(n) \right\} dg(m) \right\} d\rho\end{aligned}$$

and

$$\begin{aligned}\rho \gg\!= (\lambda m.g(m) \gg\!= h) = \lambda f.\rho \gg\!= \left(\lambda m. \int_N \left\{\lambda n. \int_Q f dh(n)\right\} dg(m)\right) \\ = \lambda f. \int_M \left\{\lambda m. \int_N \left\{\lambda n. \int_Q f dh(n)\right\} dg(m)\right\} d\rho\end{aligned}$$

so that $(\rho \gg\!= g) \gg\!= h = \rho \gg\!= (\lambda m.g(m) \gg\!= h)$, as required.

3.3.5 $\mathcal{P}$ is an Applicative Functor

Any monad immediately generates an applicative functor, and the Giry monad is no exception. An applicative functor is a certain parameterization of a lax monoidal functor, so to demonstrate that $\mathcal{P}$ has a lax monoidal structure we need to define a natural transformation ϕ such that:

$$\phi : \mathcal{P}(M) \otimes \mathcal{P}(N) \to \mathcal{P}(M \otimes N).$$

We can write this in terms of the monadic natural transformations μ and η. Evaluating right-to-left, for measures ν and ρ we get:

$$\phi_{\nu\times\rho} = \mu\mathcal{P}\left\{\lambda m.\mu\mathcal{P}\left(\lambda n.\eta_{m\times n}\right)\mathcal{P}(\rho)\right\}\mathcal{P}(\nu)$$

which can equivalently be written using bind as:

$$\phi_{\nu\times\rho} = \nu \gg\!= \lambda m.\rho \gg\!= \lambda n.\eta_{m\times n}.$$

Re-using η for the identity, we have that $(\mathcal{P}, \phi, \eta)$ is a lax monoidal functor and thus an applicative functor (see Figure 3.3).

3.4 Measure, Integral, and Continuation

So, we've established the Giry monad as a triple $(\mathcal{P}, \mu, \eta)$, where $\mathcal{P}$ is an endofunctor on the category of measurable spaces **Meas**, μ is a marginalizing inte-

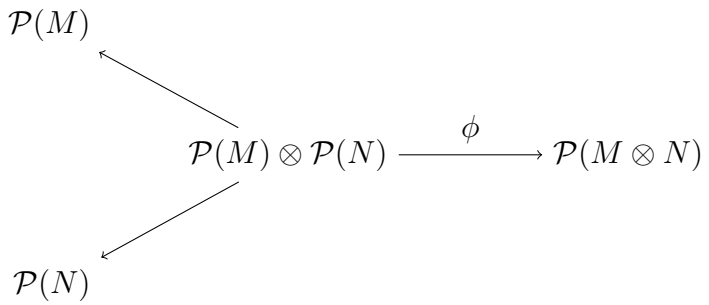


Figure 3.3: Mappings between various spaces. $\mathcal{P}(M) \otimes \mathcal{P}(N)$ is the product of the spaces of measures over M and N respectively and has the natural projections associated with a product. The natural transformation ϕ corresponding to the monoidal structure of $\mathcal{P}$ takes that product to the space of measures over the product $M \otimes N$.

gration operation defined by:

$$\mu(\rho)(A) = \int_{\mathcal{P}(M)} \left\{ \lambda\nu. \int_M \chi_A d\nu \right\} d\rho$$

and η is a monoidal identity, defined by the Dirac measure at a point:

$$\eta(x)(A) = \chi_A(x).$$

How do we actually implement this? The Giry monad is very generally constructed, but it presents numerous difficulties when it comes to implementing a consistent framework for measures in an embedded language. The most glaring difficulty is that a measure space is, in general, a fundamentally intractable object to represent explicitly, so that approach is a non-starter.

But it can be done. The key to implementing a general-purpose Giry monad is to notice that the fundamental operation involved in it is *integration*, and that we can avoid working with σ-algebras and measurable spaces directly if we focus on dealing with measurable *functions* instead of measurable *sets*.

Consider the integration map on measurable functions τ_f. For some measurable function f, τ_f takes a measure on some measurable space $M = (X, \mathcal{X})$ and uses

it to integrate f over X. In other words:

$$\tau_f(\nu) = \int_X f d\nu.$$

A measure in $\mathcal{P}(M)$ has type $X \to \mathbb{R}$, so τ_f has corresponding type $(X \to \mathbb{R}) \to \mathbb{R}$.

Switching back to programming languages, this type is analogous to the type of a *continuation*, a well-known data structure that is capable of representing programs in a well-defined sense. Continuations are *reifications* of the control state of a program as an explicit data structure; they can be used to implement various functionality in programming languages, such as exception handling, concurrency primitives, unbounded queues. They have even been used for handling *inference* – rather than representation – in certain embedded probabilistic programming frameworks [Kiselyov and Shan, 2008]. Loosely, a continuation can be thought of as 'a program with a missing piece'. To use a continuation, one supplies it with said missing piece and runs the resulting program.

A continuation in Haskell can be defined as follows:

```
newtype Cont a r = Cont ((a -> r) -> r)
```

Wadler [1994] demonstrated that this type forms a monad, and thus it also forms a functor and applicative functor. The *continuation monad* is a standard monad in the Haskell ecosystem and implementations can be found in the *transformers* library, amongst other places.

If we restrict the rightmost type parameter of 'Cont' to the reals, the type becomes quite faithful to the integration map:

```
newtype Integral a = Integral ((a -> Double) -> Double)
```

Now, let's overload notation and call the integration map τ_f *itself* a measure. That is, τ_f is a mapping $\nu \mapsto \int_X f d\nu$, so we'll just interpret the notation $\nu(f)$ to mean the same thing: $\int_X f d\nu$. This is convenient because we can dispense with τ and just pretend measures can be applied directly to measurable functions. There's no way we can get confused here; measures operate on *sets*, not functions, so notation like $\nu(f)$ is not currently in use. We just set $\nu(f) = \tau_f(\nu)$ and that's that. The 'Integral' type can be renamed to match:

```
newtype Measure a = Measure ((a -> Double) -> Double)
```

We can extract a very nice shallowly-embedded language for integration here, the core of which is a single term:

```
integrate :: (a -> Double) -> Measure a -> Double
integrate f (Measure nu) = nu f
```

Note that this is the same way we'd express integration mathematically; we specify that we want to integrate a measurable function f with respect to some measure ν:

$$\int f d\nu = \texttt{integrate f nu}.$$

The only subtle difference here is that we don't specify the space we're integrating over in the integral expression — instead, we'll bake that into the definition of the measures we create themselves.

What's interesting here is that the Giry monad has the *same* implementation as the continuation monad with the rightmost type parameter restricted to the reals. This isn't surprising when one thinks about what's going on here — we're representing measures as *integration procedures*, that is, *programs* that take a measurable function as input and then compute its integral in some particular way,

returning a real number. A measure, as we've implemented it here, is just a 'program with a missing piece'. And this is exactly the essence of the continuation monad in Haskell.

A continuation monad implementation like that in the *transformers* library often comes equipped with a supporting function like 'callCC', or 'call with current continuation' that does not have any obvious probabilistic interpretation. We define a separate 'Measure' type, rather than using a generic 'Cont' type, just to remove the possibility of accidentally using this function anywhere.

3.4.1 Typeclass Instances

We can fill out the functor, applicative, and monad instances for the 'Measure' type mechanically via reference to a standard continuation monad implementation, and each instance gives us some familiar conceptual structure or operation on probability measures. These semantics are identical to those described for the probability monad constructed in Chapter 2 — just on the level of measures, rather than discrete distributions.

The functor instance for 'Measure' lets us *transform the support* of a measure space while keeping its density structure invariant. If we have:

$$\nu(f) = \int_X f d\nu$$

for $f : X \to \mathbb{R}$, then mapping a measurable transformation $g : X \to Y$ over the measure corresponds to:

$$(\mathsf{fmap}\, g\, \nu)(f) = \int_X (f \circ g) d\nu$$

where now $f : Y \to \mathbb{R}$. The functor structure allows us to precisely express a pushforward measure or distribution of ν under g as described in Section 3.3.3. In Haskell, the functor instance corresponds exactly to the math:

```
instance Functor Measure where
  fmap g nu = Measure (\f ->
    integrate (f . g) nu)
```

It is straightforward (if tedious) to show this instance satisfies the functor laws. Note that we have:

```
fmap id nu
  = Measure (\f -> integrate (f . id) nu)
  = Measure (\f -> integrate f nu)
  = nu


fmap (g . h) (Measure nu)
  = Measure (\f -> integrate (f . (g . h)) (Measure nu))
  = Measure (\f -> integrate (f . g . h) (Measure nu))
  = Measure (\f -> nu (f . g . h))
  = Measure (\f' -> (\f -> nu (f . h)) (f' . g))
  = Measure (\f' -> integrate (f' . g) (Measure (\f -> nu (f . h))))
  = fmap g (Measure (\f' -> integrate f' (Measure (\f -> nu (f . h)))))
  = fmap g (Measure (\f' -> (\f -> nu (f . h)) f'))
  = fmap g (Measure (\f -> nu (f . h)))
  = fmap g (Measure (\f -> integrate (f . h) (Measure nu)))
  = fmap g (fmap h (Measure (\f -> integrate f (Measure nu))))
  = fmap g (fmap h (Measure nu))
  = (fmap g . fmap h) (Measure nu)
```

so that ‘fmap id = id’ and ‘fmap (g . h) = fmap g . fmap h’, as required.

The monad instance reflects exactly the Giry monad structure developed previously, and it allows us to sequence probability measures together by *marginalizing* one into another. We’ll write it in terms of bind, which went like:

$$(\rho \gg\!= g)(f) = \int_M \left\{ \lambda m. \int_N f dg(m) \right\} d\rho.$$

The Haskell translation is verbatim:

```
instance Monad Measure where
  return x  = Measure (\f -> f x)
  rho >>= g = Measure (\f ->
    integrate (\m -> integrate f (g m)) rho)
```

Since our monad encoding is equivalent to the continuation monad, we can cite Wadler [1994] to assert that the above definition satisfies the monad laws.

There's also the Applicative instance, which can be implemented in terms of the Monad instance in a standard fashion. For any monad, we can define an analogue of the natural transformation ϕ as:

```
phi :: Monad m => (m a, m b) -> m (a, b)
phi (m, n) = do
  a <- m
  b <- n
  return (a, b)
```

The probabilistic interpretation here is that ϕ takes a pair of probability measures to the product measure over the appropriate product space. For measures μ and ν on $(X, \mathcal{X})$ and $(Y, \mathcal{Y})$ respectively, we thus have:

$$\phi(\mu, \nu)(f) = \int_{X \times Y} f d(\mu \times \nu)$$

where $f : X \times Y \to \mathbb{R}$. And then for $g : X \times Y \to Z$ we can use the functor structure of $\mathcal{P}$ to do:

$$(\mathsf{fmap}\, g\, \phi(\mu, \nu))(f) = \int_{X \times Y} (f \circ g) d(\mu \times \nu)$$

where now $f : Z \to \mathbb{R}$, which corresponds to an applicative expression 'fmap g (phi mu nu)'.

Recall from Chapter 2 that the applicative structure of a probability monad allows us to perform convolution and similar algebraic operations on probability distributions. It is illustrative to examine why this is the case.

Take a measurable space $M = (X, \mathcal{X})$ where X is equipped with a *ring* structure so that it is closed under addition, subtraction, and multiplication. For measures μ and ν on M we can define the convolution of measures as follows:

$$(\mu + \nu)(f) = \int_X \int_X f(x + y) d\mu(x) d\nu(y).$$

The probabilistic interpretation here is that $\mu + \nu$ is the measure corresponding to the sum of independent measurable functions g and h with corresponding measures μ and ν respectively. To reiterate: if one has independent $g \sim \mu$ and $h \sim \nu$, then $g + h \sim \mu + \nu$.

Note that it is impossible for *measures* to be independent in any well-defined sense, but there is still a useful connection to independence that can be gleaned. To see it, complete M with some abstract probability measure $\mathbb{P}$ to form the probability space $(X, \mathcal{X}, \mathbb{P})$ and take g and h to be the measurable functions from X to $\mathbb{R}$ that fall out of the probabilistic interpretation of measure convolution. To say that g and h are independent is to say that their generated σ-algebras are $\mathbb{P}$-independent, and the measures that they correspond to are the pushforwards of $\mathbb{P}$ under g and h respectively. So, $\mu = \mathbb{P} \circ g^{-1}$ and $\nu = \mathbb{P} \circ h^{-1}$. The result is that the measurable functions correspond to different (pushforward) measures μ and ν, but are independent with respect to the same underlying probability measure $\mathbb{P}$.

The monoidal structure of $\mathcal{P}$ then gets us to convolution and friends. Given a product of measures μ and ν each on $(X, \mathcal{X})$ we can immediately retrieve their product measure $\mu \times \nu$ via the monoidal natural transformation ϕ. And from there we can get to $\mu + \nu$ via the functor structure of $\mathcal{P}$ — we just find the pushforward of $\mu \times \nu$ with respect to a function π that collapses a product via

addition. So $\pi : X \times X \to \mathbb{R}$ is defined as:

$$\pi(a \times b) = a + b$$

and then the convolution $\mu + \nu$ is thus:

$$\begin{aligned}
\mu + \nu &= (\mu \times \nu) \circ \pi^{-1} \\
&= \lambda f. \int_{X \times X} (f \circ \pi) d(\mu \times \nu) \\
&= \lambda f. \int_X \int_X f(x + y) d\mu(x) d\nu(y).
\end{aligned}$$

Other operations can be defined similarly, e.g. for $\sigma(a \times b) = a - b$ we get:

$$\mu - \nu = (\mu \times \nu) \circ \sigma^{-1}.$$

The crux of all this is that the concept of product measure *always* allows us to extract notions of independent measurable functions corresponding to its components. Moreover, the notion is not limited to convolution or other ring-like operations. For probability spaces $M = (X, \mathcal{X}, \mu)$, $N = (Y, \mathcal{Y}, \nu)$, and $Q = (Z, \mathcal{Z}, \rho)$ we can construct a product measure $\mu \times \nu \times \rho$ over the product $M \otimes N \otimes Q$ such that for a $\mathcal{X} \otimes \mathcal{Y} \otimes \mathcal{Z}$-measurable function $g : (X, Y, Z) \to \mathbb{R}$, we can say that the pushforward $(\mu \times \nu \times \rho) \circ g^{-1}$ corresponds to the measure for g applied to some independent random variables g_μ, g_ν, and g_ρ. Indeed this is trivially true for any finite product measure, which is all we need concern ourselves with here.

The same could not be said if we used the general monadic structure to apply a function like $g : X \to Y \to Z \to \mathcal{P}((\mathbb{R}, \mathcal{B}))$ to the measures μ, ν, and ρ. In such a case, the product structure is *not* enforced, so we can't guarantee that the resulting random variables are independent. For this reason, the applicative structure of the Giry monad turns out to be useful for *encoding* independence in expressions in a way that the monadic structure is not.

Given the 'phi' function defined previously, the applicative instance itself could be written in terms of the following monad-generated 'ap' function:

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap f x = fmap apply (phi f x) where
  apply (g, z) = g z
```

but instead, the CPS-type implementation we use here makes it possible to write the applicative instance without referring to any monadic binds at all:

```
instance Applicative Measure where
  pure x = Measure (\f -> f x)
  Measure g <*> Measure h = Measure (\f ->
    g (\k -> h (f . k)))
```

3.5 Conceptual Example

It's worth taking a look at an example of how things should conceivably work here. Consider the following probabilistic model:

$$\pi \sim \text{beta}(\alpha, \beta)$$
$$\mu \,|\, \pi \sim \text{binomial}(n, \pi)$$

It's a standard hierarchical presentation. A 'compound' measure can be obtained here by marginalizing over the beta measure π, leading to the well-known *beta-binomial* measure. We'll demonstrate how this proceeds in practice.

The beta distribution has support on the $[0, 1]$ subset of the reals, and the binomial distribution with argument n has support on the $\{0, \ldots, n\}$ subset of the integers, so we know that things should proceed like so:

$$\begin{aligned}\psi(f) &= (\pi \gg\!= \mu)(f) \\ &= \int_{\mathbb{R}} \left\{ \lambda p. \int_{\mathbb{Z}} f d\mu(p) \right\} d\pi.\end{aligned}$$

Eliding some theory of integration, we have that π is absolutely continuous with respect to Lebesgue measure (denoted dx) and that $\mu(p)$ is absolutely continuous with respect to counting measure (denoted $d\#$) for appropriate p. So, π admits a density $d\pi/dx = g_\pi$ and $\mu(p)$ admits a density $d\mu(p)/d\# = g_{\mu(p)}$, defined as:

$$g_\pi(p \mid \alpha, \beta) = \frac{1}{B(\alpha, \beta)} p^{\alpha-1}(1-p)^{\beta-1}$$

and

$$g_{\mu(p)}(x \mid n, p) = \binom{n}{x} p^x (1-p)^{n-x}$$

respectively, for B the beta function and $\binom{n}{x}$ a binomial coefficient. Again, we can reduce the integral as follows, transforming the outermost integral into a standard Riemann integral and the innermost integral into a simple sum of products:

$$\psi(f) = \int_0^1 \lambda p. \left\{ \lambda\alpha.\lambda\beta.g_\pi(p \mid \alpha, \beta) \sum_{z \in \{0,\ldots,n\}} f(z) \left(\lambda n. g_{\mu(p)}(z \mid n, p) \right) \right\} dx.$$

where dx again denotes Lebesgue measure. This can be expanded or simplified further (the beta and binomial are conjugates) but the point is that we have a way to evaluate the integral.

What is really required here then is to be able to encode into the definitions of measures like π and $\mu(p)$ the *method of integration* to use when evaluating them. For measures absolutely continuous with respect to Lebesgue measure, we can use the Riemann integral over the reals. For measures absolutely continuous with respect to counting measure, we can use a sum of products. In both cases, we'll also need to supply the density or mass function by which the integral should be evaluated.

3.6 Using The Measure Representation

In the following sections we'll describe what we can do with our encoding of the Giry monad. We'll describe constructing, querying, and manipulating measures by way of a number of small examples, ending with a larger example of a stochastic process measure for the Chinese Restaurant Process.

3.6.1 Constructing Measures

Recall that we are representing measures as *integration procedures*. So to create one is to define a program by which we'll perform integration.

Let's start with the conceptually simpler case of a probability measure that's absolutely continuous with respect to counting measure. We need to provide a support and a probability mass function so that we can weight every point appropriately. Then we just want to integrate a function by evaluating it at every point in the support, multiplying the result by that point's probability mass, and summing everything up. In code, this translates to:

```
fromMassFunction :: (a -> Double) -> [a] -> Measure a
fromMassFunction p support = Measure (\f ->
  foldl' (\acc x -> acc + p x * f x) 0 support)
```

So if we want to construct a binomial measure, we can do that like so:

```
binomial :: Int -> Double -> Measure Int
binomial n p = fromMassFunction (pmf n p) [0..n] where
  pmf n p k
    | x < 0 || n < k = 0
    | otherwise = choose n k * p ^^ k * (1 - p) ^^ (n - k)
```

the resulting measure corresponds to the following integration expression:

$$\nu(f) = \sum_{k \in \{0,\ldots,n\}} f(k) \binom{n}{k} p^k (1-p)^{n-k}.$$

The second example involves measures over the real line that are absolutely continuous with respect to Lebesgue measure. In this case we want to evaluate a Riemann integral over the entire real line, which is going to necessitate approximation on our part. There are numerous methods for approximating integrals, but a simple one for one-dimensional problems like this is *tanh-sinh quadrature* [Takahasi and Mori, 1974], an implementation for which exists in the *integration* package on Hackage.

```
fromDensityFunction :: (Double -> Double) -> Measure Double
fromDensityFunction d = Measure (\f ->
    quadratureTanhSinh (\x -> f x * d x))
  where
    quadratureTanhSinh = result . last . everywhere trap
```

Here we're using quadrature to approximate the integral, but otherwise it has a similar form as 'fromMassFunction'. The difference is that we're integrating over the entire real line, so don't have to supply a support explicitly. It's worth reiterating that we could in principle use any approximate integration procedure here, for example by plugging in a Monte Carlo framework implementing rejection or importance sampling. But quadrature provides an effective, low-effort choice for measures with unidimensional support.

We can use this to create a beta measure as follows:

```
beta :: Double -> Double -> Measure Double
beta a b = fromDensityFunction (density a b) where
  density a b p
```

```
  | p < 0 || p > 1 = 0
  | otherwise      =
      1 / exp (logBeta a b) * p ** (a - 1) * (1 - p) ** (b - 1)
```

and this corresponds to the quadrature approximation of the expected Riemann integral:

$$\eta(f) \approx \int_{-\infty}^{\infty} f(p) \frac{1}{B(\alpha, \beta)} p^{\alpha-1}(1-p)^{\beta-1} dp.$$

Note that since we're going to be integrating over the entire real line and the beta distribution has support only over $[0, 1]$, we need to implicitly define the support in the probability density function by specifying which regions of the domain will lead to a density of 0.

In any case, now that we've constructed those measures we can just use a monadic bind to create the beta-binomial measure described previously. It masks a lot of under-the-hood complexity:

```
betaBinomial :: Int -> Double -> Double -> Measure Int
betaBinomial n a b = beta a b >>= binomial n
```

We could equally express the beta-binomial using do-notation in order to make its internal dependency structure more explicit:

```
betaBinomial :: Int -> Double -> Double -> Measure Int
betaBinomial n a b = do
  p <- beta a b
  binomial n p
```

Another useful measure-creating function involves using a sample from some distribution in order to create an *empirical measure*. This is equivalent to passing in a specific support for which the mass function assigns equal probability to every element:

```
fromSample :: Foldable f => f a -> Measure a
fromSample = Measure . flip weightedAverage
```

where 'weightedAverage' is a function that, given a weighting function with type $(a \to r)$ and a container of values fa, returns a weighted average of type r. Flipping the argument order of 'weightedAverage' gives it the required type $(a \to r) \to r$, which is then wrapped into a continuation. Note that this also implies that we can define measures in terms of sampling functions by drawing a sample from a distribution and passing that to 'fromSample', but we will not illustrate this here.

3.6.2 Querying Measures

To *query* a measure is to simply get some result out of it, and we do that by integrating some measurable function against it. The measurable function that we integrate encodes the query.

Given a probability space $(X, \mathcal{X}, \nu)$, the simplest thing to do is to just integrate the constant function $f(x) = 1$ against ν over X:

$$\int_X 1 d\nu = \texttt{integrate (const 1) nu}.$$

This should trivially yield 1 for any probability space. We can illustrate this against the beta and beta-binomial measures created previously, for example (noting approximation error due to quadrature):

```
> integrate (const 1) (beta 10 10)
0.9999999999949557
> integrate (const 1) (betaBinomial 10 5 4)
1.000001641955738
```

The most fundamental query is the straightforward *expectation*, which simply involves integrating the identity function $f(x) = x$ against a measure. This query is so fundamental that we can build a new function for it:

```
expectation :: Measure Double -> Double
expectation = integrate id
```

We then have the identity:

$$\int_X \{\lambda x.x\}\, d\nu = \texttt{expectation nu}$$

and we can use it to calculate the means of the same beta and beta-binomial distributions. For a beta$(10, 10)$ distribution this is straightforward:

```
> expectation (beta 10 10)
0.49999999999501316
```

and since the mean of a beta(α, β) distribution is $\alpha/(\alpha+\beta)$ this is easily verified.

For the beta-binomial, or any measure not defined over the (Haskell-encoded) reals, the story is slightly different. Attempting to calculate an expectation of the beta-binomial$(10, 5, 4)$ measure yields the following:

```
> expectation (betaBinomial 10 5 4)
error:
    • Couldn't match type 'Int' with 'Double'
      Expected type: Measure Double
        Actual type: Measure Int
```

Note that the 'expectation' function assumes the target measure is defined over the reals, and this is not the case for the beta-binomial (which is defined over the integers). We can deal with this by integrating the 'fromIntegral' function $\lambda n : \mathbb{Z}.n : \mathbb{R}$ that simply casts integers to reals against the beta-binomial instead:

```
> integrate fromIntegral (betaBinomial 10 5 4)
5.5555721648806635
```

a beta-binomial(n, α, β) distribution has mean $n\alpha/(\alpha+\beta)$ so we can easily verify that this has hit the mark.

We are not limited to expectation and can indeed calculate arbitrary higher-order moments of measures. The second (central) moment of a distribution is the *variance*, for example, and we can express this in the usual way. Using the same running abstract probability space as an example, we can express the variance of ν as:

$$\begin{aligned}\text{var}(\nu) &= \int_X \{\lambda x.x^2\} d\nu - \left(\int_X \{\lambda x.x\} d\nu\right)^2 \\ &= \texttt{integrate (\^{} 2) nu - expectation nu \^{} 2}\end{aligned}$$

It is illustrative to demonstrate this over something like a binomial(n, p) distribution, for example. Analytically the variance is known to be $np(1-p)$, so for $n = 10$ and $p = 0.5$ we will expect to see 2.5. But as the binomial is defined over the integers, we have the same problem we encountered in the last example. Here instead of manually computing the variance using 'fromIntegral', we can exploit the functorial structure of measures to adapt the binomial measure to a different measurable space:

```
> variance (fmap fromIntegral (binomial 10 0.5))
2.5
```

More generally, we can calculate the n^{th} raw or central moment of a measure as follows:

```
rawMoment :: Int -> Measure Double -> Double
rawMoment n = integrate (^ n)
```

```
centralMoment :: Int -> Measure Double -> Double
centralMoment n nu = integrate (\x -> (x - rm) ^ n) nu where
  rm = rawMoment 1 nu
```

More exotic moment-based constructions follow similarly. The moment- and cumulant-generating functions for a measure ν, for example, can be defined as:

$$M_\nu(t) = \int \{\lambda x.e^{tx}\} d\nu$$

$$K_\nu(t) = \log M_\nu(t)$$

and these yield the following bindings:

```
mgf :: Measure Double -> Double -> Double
mgf nu t = integrate (\x -> exp (t * x)) nu

cgf :: Measure Double -> Double -> Double
cgf nu = log . mgf nu
```

For illustration, Figure 3.4 displays the recovered cumulant generating functions for a binomial measure, a beta-binomial measure, and a convolution of independent binomial measures (we'll discuss convolution in the next section).

The cumulative distribution function (CDF) is another good example of a function that can be extracted from a measure, assuming the measure is defined over a space that has a notion of order. For a measure defined on the reals, for example, we have that its corresponding CDF is:

$$\begin{aligned} F(x) &= \nu(\{-\infty, x\}) \\ &= \int_{\mathbb{R}} \chi_{\{-\infty,x\}} d\nu \end{aligned}$$

and we can encode it here by following the math:

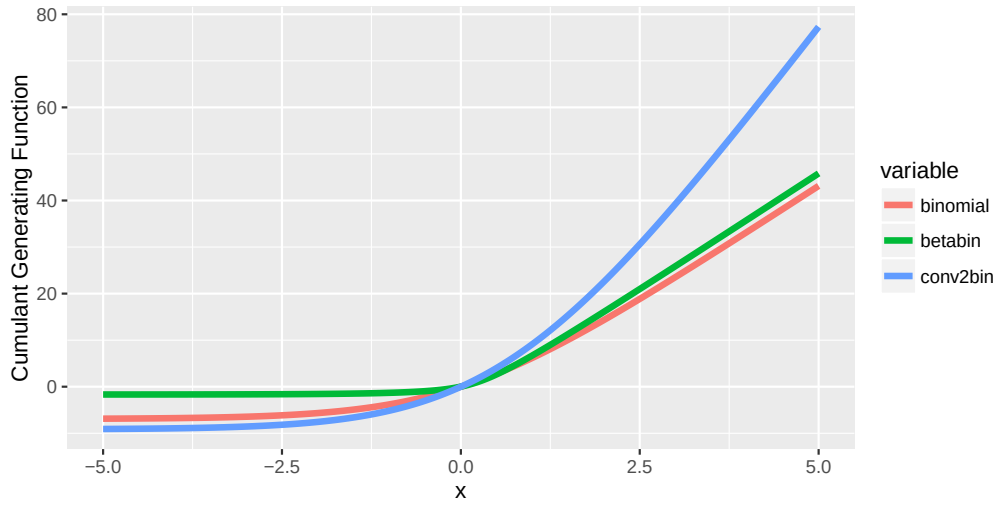


Figure 3.4: A plot of the cumulant generating functions (CGFs) recovered from various measures over the interval $t \in [-5, 5]$. The red curve corresponds to a binomial$(10, 0.5)$ measure, the green to a beta-binomial$(10, 1, 2)$ measure, and the blue to the measure defined by binomial$(10, 0.5)$ + binomial$(10, 0.2)$. The CGF for a binomial(n, p) measure is known to be $n \log(1 - p + pe^t)$ and is additive in the case of convolution.

```
cdf :: Measure Double -> Double -> Double
cdf nu x = integrate (ninfty `to` x) nu where
  ninfty = negate (1 / 0)
  to a b x
    | x >= a && x <= b = 1
    | otherwise        = 0
```

Every student of statistics is familiar with the probabilities of some standard regions of $\mathbb{R}$ under a Gaussian distribution, so it is a useful measure to use for demonstrating the CDF. We'll define a one-dimensional Gaussian measure using its density:

```
gaussian :: Double -> Double -> Measure Double
gaussian m s = fromDensityFunction (density m s) where
```

```
density m s x
  | s <= 0    = 0
  | otherwise =
      1 / (s * sqrt (2 * pi)) *
        exp (negate ((x - m) ^^ 2) / (2 * (s ^^ 2)))
```

Now, it is well-known that for a standard Gaussian measure, the probability of the region one standard deviation about the mean is approximately 0.68. We can check that using the CDF in the standard method one does in an introductory statistics course, à la $F_{N(0,1)}(1) - F_{N(0,1)}(-1)$:

```
> cdf (gaussian 0 1) 1 - cdf (gaussian 0 1) (negate 1)
0.6768132427467803
```

Other calculations proceed similarly. We can even create an analogue to the CDF for measurable spaces lacking any notion of order; the following 'containing' function calculates the probability of a region containing some given points, for example:

```
containing :: Eq a => [a] -> a -> Double
containing xs x
  | x `elem` xs = 1
  | otherwise   = 0
```

Then for a simple unordered space and measure like:

```
data Foo = Bar | Baz | Qux deriving Eq

fooMeasure :: Measure Foo
fooMeasure = fromSample [Bar, Bar, Baz, Qux, Baz, Baz]
```

we can still calculate a cumulative probability analogue:

```
> integrate (containing [Baz]) fooMeasure
0.5
> integrate (containing [Baz, Bar]) fooMeasure
0.8333333333333334
> integrate (containing [Baz, Bar, Qux]) fooMeasure
1.0
```

3.6.3 Operations on Product Measures

Since the applicative instance trivially grants us simple analogues for our ring operations, we can implement a 'Num' instance for the 'Measure' type as follows:

```
instance Num a => Num (Measure a) where
  (+)         = liftA2 (+)
  (-)         = liftA2 (-)
  (*)         = liftA2 (*)
  abs         = fmap abs
  signum      = fmap signum
  fromInteger = pure . fromInteger
```

The implementation for the addition operator '+' in particular grants us measure convolution (see Figure 3.5).

It is instructive to demonstrate convolution by constructing a χ^2 measure. Rather than define it directly via its probability density function, we can express it in terms of a sum of squared standard Gaussian measures as follows:

```
chisq :: Int -> Measure Double
chisq k = sum (replicate k sqgauss) where
  sqgauss = fmap (^ 2) (gaussian 0 1)
```

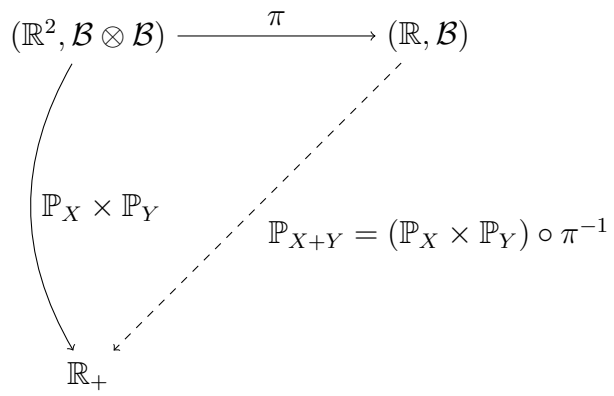


Figure 3.5: Mappings between various spaces. Here, $\mathbb{P}_X \times \mathbb{P}_Y$ is a product measure over $(\mathbb{R}^2, \mathcal{B} \otimes \mathcal{B})$. The function $\pi : \mathbb{R}^2 \to \mathbb{R}$ is defined as $\pi(\{x, y\}) = x + y$ and collapses any element of $\mathbb{R}^2$ into an element of $\mathbb{R}$ by summing its components together. Pushing it onto the product measure $\mathbb{P}_X \times \mathbb{P}_Y$ creates the pushforward measure $\mathbb{P}_{X+Y}$. This is an equivalent construction for measure convolution as described in Section 3.3.

The 'chisq' expression creates a list of k squared standard Gaussians and then simply adds them up, resulting in a χ^2 measure with degrees of freedom k. We can sanity check the implementation by calculating its mean and variance, which are known to be k and $2k$ respectively:

```
> expectation (chisq 2)
2.0000000000000004
> variance (chisq 2)
4.0
```

As a second example, consider a measure defined by the *product* of independent Gaussian measures. This is a trickier distribution to deal with analytically, but we can use some well-known identities for general independent random variables in order to verify our results. For independent random variables f and g, we have:

$$\mathbb{E}(fg) = \mathbb{E}f\mathbb{E}g$$

and

$$\mathrm{var}(fg) = \mathrm{var}(f)\mathrm{var}(g) + \mathrm{var}(f)(\mathbb{E}g)^2 + \mathrm{var}(g)(\mathbb{E}f)^2$$

for the expectation and variance of their product. Since independence is enforced under the product measure, we can calculate these for a product of Gaussian$(1, 2)$ and Gaussian $(2, 3)$ measures and expect to see 2 for its expectation and 61 for its standard deviation:

```
> expectation (gaussian 1 2 * gaussian 2 3)
2.0000000000000001
> variance (gaussian 1 2 * gaussian 2 3)
61.000000000000003
```

It is worth pointing out that while convolution and associated operators carry some familiar algebraic intuitions, they do not correspond one-to-one with ring operations *per se*. In particular, the expression $\nu+\nu$ does not really correspond to the intuitive 2ν, nor does the expression $\nu-\nu$ correspond to 0, and so on. As detailed in Section 3.3, the correct intuition is that a measure expression consisting of ring operations corresponds to the distribution of independent random variables pieced together using the same expression. The reason we don't get true ring operations on measures is because we don't define any notion of *equality* on measures, and so $\nu + \nu$ must be interpreted as the distribution of the sum of two independent and identically-distributed random variables — *not* the sum of two copies of the same measure. This is typically made explicit when discussing random variables; a sum $\sum_{i\in I} f_i$ of independent and identically-distribution (*iid*) random variables f_i is never interpreted to mean $|I|f$.

Similarly, it's also worth noting that we don't define any notion of 'measure division'. Its expression would correspond to:

$$\frac{\mu}{\nu}(f) = \int_X \int_X f\left(\frac{x}{y}\right) d\mu(x) d\nu(y)$$

which is only defined if X does not contain an additive identity element (i.e. if X has only a semigroup or semiring structure). Since we are always integrating over the entire space X, we can't avoid touching $y = 0$ if it exists, leaving $f(x/y)$ and thus the integral undefined for any x in X.

A final interesting example that illustrates the flexibility of our implementation involves convolving an empirical measure with a measure absolutely continuous with respect to Lebesgue measure. Given a sample of points in $\mathbb{R}$ drawn from some distribution (possibly via Monte Carlo or some other experimental procedure), we can 'smooth' it via convolution with a proper Gaussian, for example:

```
-- kept abstract
observations :: [Double]

empirical :: Measure Double
empirical = fromSample observations

smoothed :: Measure Double
smoothed = gaussian 0 1 + empirical
```

We can then query the smoothed measure for its moments, CDF (see Figure 3.6), and so on. If 'observations' is a list of 15 values sampled from a Gaussian$(0.5, 1)$ distribution, for example, then we find moments like the following from the smoothed measure:

```
> expectation smoothed
0.2853184714048533
> variance smoothed
2.5247155156731
```

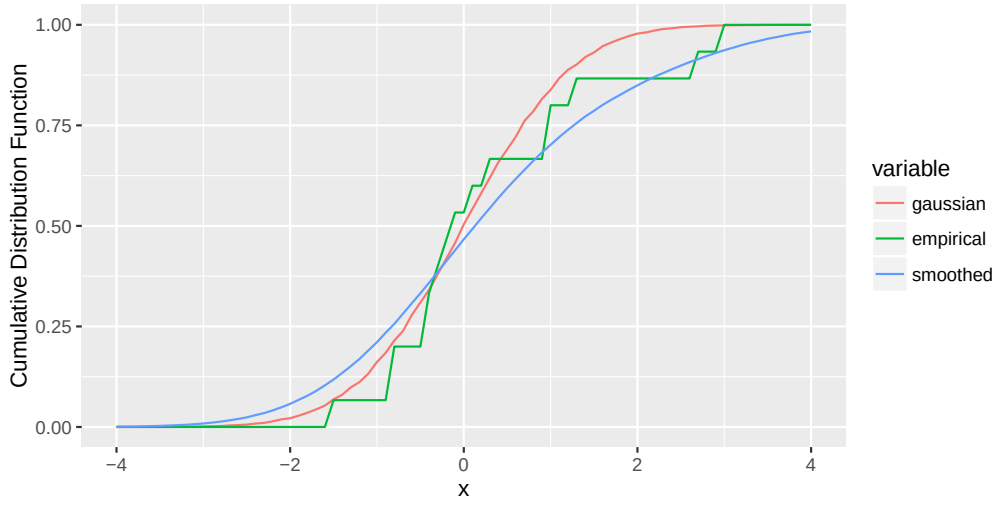


Figure 3.6: A plot of the cumulative distribution functions recovered from three measures. The red CDF corresponds to a standard Gaussian measure, the green CDF to an empirical measure constructed by sampling 15 *values from a Gaussian*$(0.5, 1)$ *distribution, and the blue CDF to the smoothed Gaussian obtained by convolving the previous two measures together.*

3.7 Example: Chinese Restaurant Process Measure

As a more heavyweight example, we can create a measure characterizing the Chinese Restaurant Process (CRP), a stochastic process used as a prior in various nonparametric Bayesian models. Figure 3.7 shows a standard visualization of the CRP; it is a probability distribution over the space of *finite partitions of the natural numbers* and is useful in clustering and partitioning problems.

The CRP describes the arrival of customers at a hypothetical Chinese restaurant with an infinite number of tables. The first customer arrives from the left and sits at the first table deterministically. For $\mathcal{R}_n$ denoting the state of the restaurant with n customers, the $n + 1^{th}$ customer decides where to sit according to the

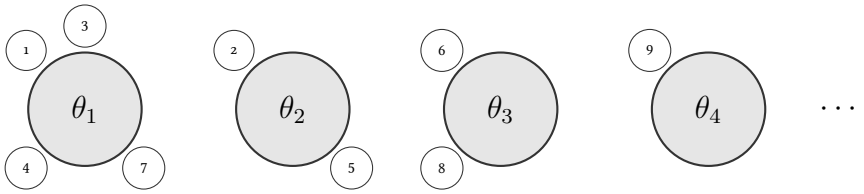


Figure 3.7: A sequential visualization of the Chinese Restaurant Process, where the indexed θ parameters represent tables and the smaller 'orbiting' circles represent customers. The customers are labelled by their arrival order.

probability mass function

$$P(\text{occupied table } k \mid \mathcal{R}_n) \propto \text{number of customers at table } k \tag{3.1}$$
$$P(\text{next unoccupied table} \mid \mathcal{R}_n) \propto \alpha$$

where $\alpha > 0$ is a dispersion parameter such that the expected number of occupied tables increases monotonically with it.

To demonstrate how our shallowly-embedded language operates with standard facilities and libraries in the host language, we can implement a measure for a CRP by using the 'Measure' type and its associated functions alongside a mix of ad-hoc Haskell data structures and other host language functionality.

We'll make use of an 'IntMap', which is a simple key/value data store where the key type is restricted to the integers. It is available in the *containers* library on Hackage, and we'll define a couple of helper functions to count what will be the number of customers and tables in a Chinese restaurant respectively:

```
import qualified Data.IntMap.Strict as IMS

customers :: IMS.IntMap Int -> Int
customers = IMS.foldl' (+) 0

tables :: IMS.IntMap a -> Int
tables = IMS.size
```

We'll denote a table in our restaurant by a simple pair of integers. The first component will be the table's label, and the second will be the number of diners currently seated at it. We'll define an entire restaurant at the n^{th} customer arrival by an IntMap, which encodes the number of diners seated at each occupied table:

```
type Table = (Int, Int)

type Restaurant = IMS.IntMap Int
```

The strategy here is to create a measure over individual tables, and then use that to create a 'kernel' measure over restaurants of size $n + 1$, given a restaurant of size n. We can then create a measure over generic restaurants of size n by recursively iterating the kernel measure n times. The key is to focus on solving the smaller problems, and then compose those solutions together to solve the grander problem.

Let $d(t)$ denote the number of diners sitting at a table t. Building on Equation 3.1, the precise probability mass function for a table t given a restaurant $\mathcal{R}_n$ of size n is:

$$P(t \mid \mathcal{R}_n, \alpha) = \begin{cases} \frac{\alpha}{n+\alpha} & d(t) = 0 \\ \frac{d(t)}{n+\alpha} & d(t) > 0 \end{cases}$$

and we can encode it analogously as follows:

```
pmf :: Restaurant -> Double -> Table -> Double
pmf restaurant a (table, ndiners)
    | tables restaurant == table = a / denom
    | otherwise = fromIntegral ndiners / denom
  where
    denom = fromIntegral (customers restaurant) + a
```

We can use τ_n to denote the measure over tables in a restaurant of size n. The measure over tables for the $n + 1^{th}$ arrival can be defined via:

```
tau
  :: Foldable f
  => Restaurant -> Double -> f Table -> Measure Table
tau restaurant a support =
  fromMassFunction (pmf restaurant a) support
```

and in maths we can express that as $\tau_{n+1}(\cdot \mid \alpha, \mathcal{R}_n, \mathcal{S})$.

Now for the measure over restaurants of size $n+1$ given a restaurant of size n, we'll define it according to the following procedure.

Denote the number of occupied tables in a restaurant $\mathcal{R}_n$ by $|\mathcal{R}_n|$ and let $\mathcal{R}_n^*$ denote the augmented restaurant obtained by populating a new table in $\mathcal{R}_n$ with a ghost, such that $|\mathcal{R}_n^*| = |\mathcal{R}_n| + 1$ even though the number of (living) customers is the same in each. Let $\mathcal{S}_n$ denote the support — the possible states of the restaurant — of the measure when the $n+1^{th}$ customer arrives.

Define the following function that creates a restaurant from a table and existing restaurant pair:

$$h(\{t, \mathcal{R}_n\} \mid \mathcal{R}_m) = \begin{cases} d(t) & |\mathcal{R}_m| = \mathcal{R}_n + 1 \\ d(t) + 1 & \text{otherwise.} \end{cases}$$

Denote updating the number of diners at a table t to l in a restaurant $\mathcal{R}_n$ by $\mathcal{R}_n \vdash d(t) \to l$. We can then define the measure ρ_{n+1} over restaurants $\mathcal{R}_{n+1}$, given a restaurant $\mathcal{R}_n$, as:

$$\rho_{n+1}(\cdot \mid \alpha, \mathcal{R}_n) = \tau_{n+1}(\alpha, \mathcal{R}_n^*, \mathcal{S}_n) \gg\!= \lambda t.\delta_{\mathcal{R}_n \vdash d(t) \to h(t, \mathcal{R}_n)}.$$

In code, that looks like:

```
rho :: Restaurant -> Double -> Measure Restaurant
rho restaurant a = do
```

```
    let ntables   = tables restaurant
        support   = IMS.insert (ntables + 1) 1 restaurant
        virtual   = IMS.insert (ntables + 1) 0 restaurant

    (idx, diners) <- tau virtual a (IMS.toList support)

    let ndiners
          | idx == ntables + 1 = diners
          | otherwise          = diners + 1

    return (IMS.insert idx ndiners restaurant)
```

The CRP measure for a restaurant of size n itself can then finally be defined by recursively iterating the 'next restaurant' measure ρ for n arrivals, each time conditioning on the restaurant measure corresponding to the previous customer arrival:

```
crp :: Int -> Double -> Measure Restaurant
crp epochs a = loop epochs IMS.empty where
  loop n restaurant
    | n <= 0    = return restaurant
    | otherwise = do
        newRestaurant <- rho restaurant a
        loop (n - 1) newRestaurant
```

Now, let's construct measures for three restaurants of size $n = 2$, $n = 5$, and $n = 10$ and make some queries over them.

First, the measures:

```
small :: Double -> Measure Restaurant
small a = crp 2 a
```

```
medium :: Double -> Measure Restaurant
medium a = crp 5 a

large :: Double -> Measure Restaurant
large a = crp 10 a
```

The most intuitive thing to query a CRP for is its expected number of tables for some number of arrivals n and dispersion parameter α. It is known that a CRP observed for n arrivals has expected number of tables equal to $\Psi(\alpha + n) - \Psi(\alpha)$ for Ψ the digamma function. The expected number of tables for the small, medium, and large restaurants with $\alpha = 1$ should thus be 1.5, 2.28, and 2.93 respectively. Note that we can use the existing 'tables' function to build our query:

```
> integrate (fromIntegral . tables) (small 1)
1.5
> integrate (fromIntegral . tables) (medium 1)
2.283333333333333
> integrate (fromIntegral . tables) (big 1)
2.9289682539682533
```

To calculate the variances (for which I can't find a closed-form expression easily, at least) we can just adapt the respective measures to the reals via their functor structure and then apply the variance query:

```
> variance (fmap (fromIntegral . tables) (small 1))
0.25
> variance (fmap (fromIntegral . tables) (medium 1))
0.8197222222222225
> variance (fmap (fromIntegral . tables) (big 1))
1.3792005228017157
```

The restaurants become more antisocial as we increase the dispersion parameter — diners prefer to sit by themselves instead of joining an existing table. Here is the expected number of tables and variance for the 'big' restaurant when using $\alpha = 10$ instead:

```
> expectation (fmap (fromIntegral . tables) (big 10))
7.18771403175428
> variance (fmap (fromIntegral . tables) (big 10))
1.798162757106006
```

As a sort of sanity check, we can calculate the expected number of customers for a restaurant of size n. It should be n, naturally:

```
> integrate (fromIntegral . customers) (big 10)
10.0
```

Finally, for illustration we can recover the cumulant generating function for the pushforward measure of a CRP under the number-of-tables query:

```
> cgf (fmap (fromIntegral . tables) (big 5))
```

Figure 3.8 displays the CGF over the region $t \in [-5, 5]$.

3.8 Summary

Our continuation-based Giry monad constitutes a small DSL shallowly-embedded in Haskell. At its core it has just a small number of components. To review, it consists of:

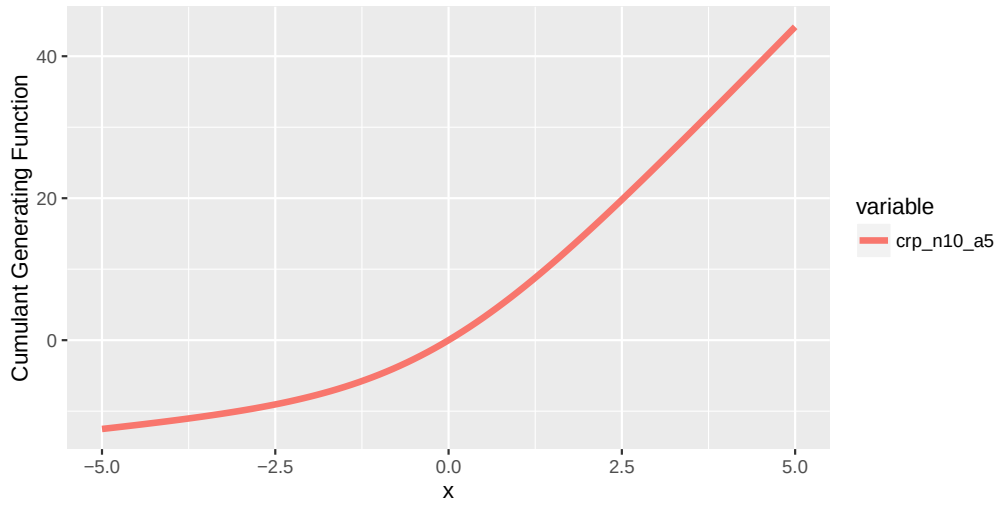


Figure 3.8: A plot of the cumulant generating function recovered from the pushforward of the CRP$(10, 5)$ *measure under a number-of-tables query over the region* $t \in [-5, 5]$.

- The 'Measure' type, which simply holds a continuation representing a measure as an abstract integration procedure. We parameterize the measure type over an abstract type parameter in order to support arbitrary measurable spaces, and provide the 'fromMassFunction', 'fromDensityFunction', and 'fromSample' in order to create 'Measure'-typed values.

- The functor, applicative, and monad instances for the 'Measure' type. Each foundational typeclass instance immediately grants us a familiar probabilistic tool that we can use to manipulate, examine, or modify measures. The functor instance lets us create image measures from existing measures via 'fmap', whereas the applicative instance lets us create product measures that encode independence (and thus perform convolution, etc.). The monad instance allows us to compose measures together, marginalizing one into another in standard hierarchical fashion.

- The core evaluation function 'integrate' that evaluates a measure by integrating it against a measurable function. More advanced queries such as

higher-order moments, moment and cumulant generating functions, and cumulative distribution functions can all be built using this one primitive.

- Various facilities from our host language, Haskell. A parser, compiler, module system, function support, recursion, external libraries, data structures, and so on.

A version of the EDSL presented here is available under the MIT license as the *measurable* library [Tobin, 2013b].

3.8.1 Computational and Feature Limitations

The Giry monad is a useful and general abstract construction for formalizing the monadic structure of probability distributions, and as canonical probabilistic objects, measures and integrals become tremendously useful when working theoretically and proving things rigorously. But any reasonably-faithful implementation of the Giry monad faces serious problems when it comes to getting work done in practice.

The crucial problem that one quickly runs into when using the Giry monad is that **measures are a poor way to represent probability distributions from a computational perspective**. Consider the following integral expression that arises from a monadic bind:

$$(\nu \gg\!= \mu)(f) = \int_M \left\{ \lambda m. \int_M f d\mu(m) \right\} d\nu.$$

For tractability, assume that M is discrete and has cardinality $|M|$. The integral thus reduces to:

$$(\nu \gg\!= \mu)(f) = \underbrace{\sum_{m \in M} d\nu(m) \underbrace{\sum_{n \in M} f(n) d\mu(m)(n)}_{O(|M|)}}_{O(|M|)}$$

where $d\mu(m)$ and $d\nu$ are the appropriate Radon-Nikodym derivatives. One can see that the total number of operations involved in the integral is $O(|M|^2)$, and indeed, for p monadic binds the computational complexity involved in evaluating all the integrals involved is exponential, on the order of $|M|^p$. It was no coincidence that Section 3.6.3 demonstrated a variance calculation for a $\chi^2(2)$ distribution instead of a $\chi^2(10)$ — even a modest number of monadic binds can make the running time and space usage blow up to unacceptable levels. We can avoid using monadic binds by using applicative combinators (for example for convolution, etc.), but the running time and space usage turn out to be comparable to the monadic case. The complexity of applicative operators is harder to characterize formally, and we won't attempt to do that here.

A particular pain point involves integrating 'continuous' measures, i.e. those that are absolutely continuous with respect to Lebesgue measure. In the one-dimensional case of the reals that we support here, we resort to using quadrature in order to approximate a Riemann integral. Quadrature can be a comparatively expensive operation as it is, and the exponential complexity of only a handful of monadic binds makes this very obvious very quickly. What's more, quadrature does not scale well beyond a few dimensions (indeed, the quadrature implementation used here supports only one dimension), so what we can achieve with it is limited. To support higher-dimensional spaces we would need a more appropriate approximate integration procedure.

An additional problem is that the utility of the measure representation is limited from a practical perspective. A measure obviously provides functionality for *integration*, but many desirable features of day-to-day work are poorly supported by measures — such as optimization, sampling, and visualization. Since querying a measure involves integrating it, we're also restricted to queries that return a real number, which seems unnecessarily limiting. So while we can encode the Monty Hall problem from Chapter 2 just as easily using measures as we could using the discrete distribution type, we can't query it quite as easily. At least not

without first specifying some mapping between booleans and reals that we can use for integration.

When taking a high-level view, the problems with the measure representation should not be surprising: approximate integration procedures like MCMC exist precisely *because* exact integration is difficult, so a representation of probability distributions based on integration does not turn out not to be a particularly fruitful way to get work done. The most useful contribution of this work is the probabilistic interpretation that we have developed around the structure of the Giry monad, but a simple conclusion for practical work is: relegate the measures to measure theory, where they seem to belong.

3.8.2 Comparison with Other Work

Perhaps the greatest strength of the Giry monad is that it can represent any probability distribution and be used to derive concise probabilistic semantics around structures like functor, applicative, and monad by way of category and measure theory. We can carry these intuitions to other probability types that can be put in correspondence with the Giry monad, such as the toy language discussed in Chapter 2, or the types we will discuss in Chapter 4. This idea of using the Giry monad to derive denotational semantics of probabilistic programming languages was also noted by Ścibior et al. [2017], though the authors did not extrapolate on interpretations of functorial nor applicative structure.

Since Giry [1981] and especially in recent years there seems to have been some activity in the category theory literature around the categorical structure of various concepts in probability, measure, and integration theory. But these (naturally) seem to have focused on establishing rigorous categorical formalisms rather than extracting satisfying probabilistic interpretations. Our focus here is obviously the latter.

Two notable, recent works here are Rodrigues [2009] — which outlined a lengthy, formal roadmap for 'categorifying' measure theory — and Sturtz [2014], who rigorously discussed structural properties of **Meas** as well as alternate theoretical constructions for the Giry monad. This author suspects that the notion of image or pushforward measure as a functor encoding of an underlying probability distribution is common (or even plainly obvious) to category theorists, but it does not seem to have been well-disseminated outside of the category theory literature. Sturtz [2014] discussed the monoidal structure of **Meas**, but did not note the monoidal structure of the Giry monad that connects it to applicative functors and product measure.

Due to its inherent computational complexity and the limited practical use of many integration-based queries, direct encodings of the Giry monad are uncommon in practice. Borgström et al. [2011] investigated using measure-theoretic constructs to define semantics in probabilistic programs, but did not actually implement a language in which measures were first-class citizens. Recent development versions of Hakaru [Hakaru, 2014] do make some use of some so-called measure constructs in their implementation, but their purpose in the language's internals is unclear, and the language is clearly not based around direct integration semantics. Most shallowly-embedded probabilistic programming frameworks opt for some other fundamental basis that can offer more practical utility. Erwig and Kollmansberger [2006], Kiselyov and Shan [2008], and Sato [2009] each used an explicit representation of discrete distributions in line with the embedded language described in Chapter 2, which can be useful, but limits the distributions under consideration to discrete supports only. *Samplng functions* are a more practically useful and flexible basis and were discussed by both Ramsey and Pfeffer [2002] and Park et al. [2008], and other embeddings tend to make use of a collection of *abstract* probability distributions that can be associated with some particular semantics (or collection of semantics) such as sampling functions. The latter strategy typically involves using a deeper embedding, and we'll discuss the merits of that and the sampling function basis in Chapter 4.

The most important comparison to the implementation work developed in Section 3.4 is that of Ramsey and Pfeffer [2002], who presented an 'expectation monad' that is structurally equivalent to the probability monad presented here, and thus also corresponds to the Giry monad in the sense that it represents probability distributions by way of integration. It is worth making a special comparison to their work. The primary construct they deal with is an abstract representation of the Giry monad in the form of a typeclass, defined as follows:

```
class Monad m => ExpMonad m where
  expectation :: (a -> Double) -> m a -> Double
```

An appendix also defines a concrete instance of 'ExpMonad':

```
newtype Exp a = Exp ((a -> Double) -> Double)

instance Monad Exp where
  return x    = Exp (\h -> h x)
  Exp d >>= k = Exp (\h ->
    let apply (Exp f) arg = f arg
        g x = apply (k x) h
    in  d g)

instance ExpMonad Exp where
  expectation h (Exp d) = h d
```

It is easy to see the similarity to the implementation of Section 3.4, though the primarily-discussed abstract 'ExpMonad' class is sufficiently different to the present implementation that it was not obvious the two both represented the Giry monad until some point after the ideas here had first materialized. But as pointed out in 3.4, the Giry monad *is* the continuation monad restricted to the reals, and thus it is in hindsight no surprise that *both* the present implementation and the

expectation monad of Ramsey and Pfeffer [2002] are structurally equivalent to the continuation monad of Wadler [1994].

We explicitly claim differentiation from Ramsey and Pfeffer [2002]'s work on the expectation monad on two dimensions. On the theoretical side, aside from pointing out the connection between the Giry and continuation monads, we also develop several probabilistic interpretations not discussed by Ramsey and Pfeffer [2002], namely the functor-provided pushforward operation via 'fmap' and the applicative-provided notion of product measure, independence, and convolution (indeed, applicative functors were not well-known when Ramsey and Pfeffer [2002] published their work). On the implementation side, we develop a number of more exotic queries that can be expressed in an integration-based language and that were not discussed by Ramsey and Pfeffer [2002], such as higher-order moments, cumulant generating functions, and CDFs, and demonstrate that these can be used on measures defined over varied supports.

There is an interesting developing collection of theoretical work that seems to connect the Giry monad to the so-called *codensity monad* of a functor, which has an almost equivalent implementation to the continuation monad [Kmett, 2008b]. We do not investigate this topic any further here, but refer to Leinster [2013] and Avery [2016] for some rigorous categorical work in the area.

3.9 Conclusion

This chapter has demonstrated that the Giry monad is sufficient for representing arbitrary probability distributions in an embedded language. We provided probabilistic interpretations for its functorial, applicative, and monadic structure, and then implemented a small, shallowly-embedded DSL for creating, transforming, and querying measures. The shallowly-embedded DSL is seen to be equivalent to that constructed from a restricted continuation monad.

While the Giry monad is sufficient for *representation*, however, it is not particularly useful in practice. There are three crucial problems with it:

- Querying measures is prohibitively expensive.
- We are limited to performing integration-based queries.
- We can't examine the structure of a measure expression.

In the next chapter we'll resolve these problems by moving from the Giry monad to two other probability monads, including one that can reify probabilistic models in a structure-preserving form.

Chapter 4

Representing Structured Probabilistic Models

> The limits of my language means the limits of my world.
>
> Wittgenstein

4.1 Abstract and Contributions

This chapter extends the work we did on representing probability distributions via the Giry monad in Chapter 3. Instead of representing distributions as measures, we'll now capture probability distributions in a general and *structure-preserving* way, distinguishing them as structured probabilistic *models* that are amenable to arbitrary interpretation.

To prepare for the sequel, the first part of the chapter details the well-known *sampling monad*, which, unlike the Giry monad, is computationally efficient and

practically useful. Like the Giry monad, though, the marginalizing interpretation of the sampling monad is *lossy* in that evaluating a monadic bind discards information about a model's internal structure, which we generally require access to in order to do some useful forms of approximate inference. We demonstrate how uninterpreted *abstract syntax* captures the structure of a program, and note that a deep embedding of our language is required in order to get access to it.

We then introduce the concept of algebraic *freeness* and discuss the *free monad*, a monad for which the bind operator is structure-preserving. The free monad is used to embed a monadic probabilistic language capable of denoting models that are amenable to inference. Exploiting the free structure also means that the marginalizing interpretations captured by the Giry and sampling monads can trivially be grafted onto the deeply-embedded language as interpreters.

We can also exploit freeness and its algebraic dual, *cofreeness*, when it comes to applicative functors and *comonads*. Free applicative functors let us encode conditional independence via product measure, while cofree comonads let us perturb a model's internal parameters in order to do (for example) single-site Metropolis-Hastings.

The primary contributions of this chapter are:

- A novel technique for **embedding a statically-typed probabilistic programming language in a purely functional language**. We use the *free monad* of a probabilistic base functor in order to define our embedded language, giving us the same syntax as the language based on the Giry or sampling monads, but with considerably more flexibility when it comes to interpretation.
- A novel **characterization of execution traces as cofree comonads**. We demonstrate that probabilistic programs encoded using the free monad have a dual representation as execution traces under the *cofree comonad*,

which allows us to 'move about' and perturb a model's internal parameters. We then implement a novel **comonadic Markov Chain Monte Carlo algorithm** that makes use of this characterization.

- A novel technique for **statically encoding conditional independence** between terms in the embedded language. Following from the applicative/product measure structure derived in Chapter 3, we use the *free applicative functor* in order to capture applicative expressions in a structure-preserving way.

4.2 A Sampling Function-Based Representation

The Giry monad-based DSL presented in the previous chapter is a 'canonical' probability monad in a sense, but it is not the only probability monad. Ramsey and Pfeffer [2002] described a 'sampling monad' — a probability monad based on *sampling functions* — in addition to their 'expectation monad' encoding of the Giry monad. A sampling function is a function that takes as input a source of randomness and returns as output a sampled value from the support of some target probability distribution. Ignoring implementation details around pseudo-random number generators, they can safely be thought of as random variables.

Random variables characterize probability distributions uniquely in the same way that measures or density functions do, and this claim extends to sampling functions [Ramsey and Pfeffer, 2002]. Park et al. [2008] gave a seminal treatment on the use of sampling functions as a basis for monadic probabilistic programming languages; they developed a standalone language called λ_0 based on sampling functions.

A sampling function-based embedded language is more practically useful than one based on the Giry monad. We can generate observations from the predictive

distribution described by a model by 'forward-sampling' from it, and then approximate a posterior distribution by any number of 'backward-sampling' inference algorithms, such as rejection or importance sampling or MCMC. One can get surprisingly far with a shallowly-embedded sampling function-based language when it comes to approximate inference. But there are certain inference algorithms for which a shallow embedding does not suffice, and we need some more structure in order to proceed.

In this section we'll run through an implementation of a probability monad based on sampling functions, in order to set the stage for the deeply-embedded probabilistic language we'll develop in the sequel. The probabilistic semantics encoded by the Giry monad have precise analogues here: to implement them, we'll create a new type to represent probability distributions, and then use the now-familiar technique of wrapping some monadic structure around it.

4.2.1 Implementation and Computational Complexity

There are two key concepts that we'll use to implement a 'sampling monad'. The first is the same kind of parametric polymorphism used to characterize the support type of the distributions we've seen thus far, and the second is a pseudo-random number generator (PRNG). As Haskell is a purely functional language, we can't trivially pluck random numbers out of thin air like we might be used to in a language like R or Python. Instead, we need to pass the state of a PRNG along with the state of our computation, and use *that* as an explicit source of randomness whenever we need to generate random numbers.

There are a number of good libraries for randomness in Haskell's library ecosystem. One of the best is the *mwc-random* library [O'Sullivan, 2009], which implements performant and high-quality random number generation via the multiply-with-carry algorithm of Marsaglia and Zaman [1991]. *mwc-random* uses a monad

to handle passing the state of the PRNG, but we can implement a simple probability monad by adding another layer with the familiar semantics on top of that.

Given the underlying functionality of *mwc-random*, constructing a probability monad is almost trivial. First, the probability distribution type we'll use as a basis:

```
data Prob m a = Prob { sample :: Gen (PrimState m) -> m a }
```

Our probability distribution type is called 'Prob', and it is a very simple wrapper around a particular function type. The 'Prob m a' type wraps a function called 'sample' that takes a PRNG (the 'Gen (PrimState m)' term) and returns a monadic value of type 'm a'. The restriction here is that 'm' must be a particular kind of primitive monad, but we don't need to examine this in detail here.[1]

The rest of the machinery falls out naturally:

```
instance Monad m => Functor (Prob m) where
  fmap h (Prob f) = Prob (\x ->
    fmap h (f x))

instance Monad m => Applicative (Prob m) where
  pure  = return
  (<*>) = ap

instance Monad m => Monad (Prob m) where
  return x = Prob (\_ -> return x)
  m >>= h  = Prob (\g -> do
    z <- sample m g
    sample (h z) g)
```

[1]To be precise, the monad 'm' here must be a member of the 'PrimMonad' typeclass, which describes IO-like monads that make use of GHC's low-level state handling features. This restriction doesn't apply to the type or instance definitions - just values that we'll create later.

The semantics derived from the Giry monad are unchanged: the functor instance allows us to create pushforwards by transforming the support of a distribution, the applicative instance encodes independence, and the monad instance allows us to compose distributions together by marginalizing one into the other.

We can verify the functor and monad laws as follows. First, for functor we have:

```
fmap id (Prob f)
  = Prob (\x -> fmap id (f x))
  = Prob f

fmap (g . h) (Prob f)
  = Prob (\x -> fmap (g . h) (f x))
  = Prob (\x -> (g . h . f) x)
  = fmap g (Prob (\x -> (h . f) x))
  = fmap g (fmap h (Prob (\x -> f x)))
  = (fmap g . fmap h) (Prob f)
```

so that identity and composition are preserved. For monad, left and right-identity can be verified via:

```
return x >>= f

  = Prob (\g -> do
      z <- sample (Prob (\_ -> return x)) g
      sample (f z) g)

  = Prob (\g -> do
      z <- return x
      sample (f z) g)

  = Prob (\g -> sample (f x) g)
```

```
  = Prob (sample (f x))

  = f x

m >>= return

  = Prob (\g -> do
      z <- sample m g
      sample (return z) g)

  = Prob (\g -> do
      z <- sample m g
      sample (Prob (\_ -> return z) g))

  = Prob (\g -> do
      z <- sample m g
      return z)

  = Prob (sample m)

  = m
```

Associativity follows via:

```
(m >>= f) >>= h

  = Prob (\g -> do
      z <- sample m g
      sample (f z) g) >>= h

  = Prob (\g -> do
```

```
      z <- sample (Prob \g' -> do
        z' <- sample m g'
        sample (f z) g') g
      sample (h z) g)

  = Prob (\g -> do
      z' <- sample m g
      z  <- sample (f z') g
      sample (h z) g)

m >>= \x -> (f x >>= h)

  = m >>= \x -> Prob (\g -> do
      z <- sample (f x) g
      sample (h z) g)

  = Prob (\g -> do
      z <- sample m g
      sample ((\x -> Prob (\g' -> do
        z' <- sample (f x) g'
        sample (h z') g')) z) g)

  = Prob (\g -> do
      z <- sample m g
      sample (Prob (\g' -> do
        z' <- sample (f z) g'
        sample (h z') g')) g)

  = Prob (\g -> do
      z  <- sample m g
      z' <- sample (f z) g
      sample (h z') g)
```

so that $(m \gg\!= f) \gg\!= h = m \gg\!= (\lambda x.fx \gg\!= h)$, as required.

That is more or less it. The sampling function implementation is simple to implement and simple to extend. Individual sampling functions can be created from others by familiar techniques such as the Box-Muller transformation and friends; Park et al. [2008] denote a number of distributions by sampling functions in this fashion. The *mwc-random* library contains a plethora of sampling functions out of the gate, and in many cases we can exploit the monadic structure we have in order to define others. A log-normal distribution can be defined by pushing an exponential function onto a sampling function for the Gaussian distribution:

```
logNormal :: PrimMonad m => Double -> Double -> Prob m Double
logNormal m sd = fmap exp (gaussian m sd)
```

Similarly an inverse-gamma distribution can be characterized by fmapping a reciprocal function over an existing sampling function for the gamma distribution:

```
inverseGamma :: PrimMonad m => Double -> Double -> Prob m Double
inverseGamma a b = fmap recip (gamma a b)
```

We can use the applicative or monadic structure if we want to exploit gamma sampling functions to characterize the beta distribution:

```
beta :: PrimMonad m => Double -> Double -> Prob m Double
beta a b = do
  u <- gamma a 1
  w <- gamma b 1
  return (u / (u + w))
```

And so on.

Unlike the measure representation, the sampling function representation is computationally efficient. Compare the monad instance for this 'sampling monad' with the monad instance for the Giry monad:

```
-- Giry
instance Monad Measure where
  return x  = Measure (\f -> f x)
  rho >>= g = Measure (\f ->
    integrate (\m ->
      integrate f (g m))
    rho)

-- sampling
instance Monad m => Monad (Prob m) where
  return x = Prob (const (return x))
  m >>= h  = Prob (\g -> do
    z <- sample m g
    sample (h z) g)
```

The marginalizing semantics of the continuation-based Giry monad are based on integration, so as mentioned in Chapter 3 for any monadic bind we need to perform an expensive iterated integral. Notably, the inner integral in general doesn't have a constant value, so the complexity of additional binds is multiplicative, and thus exponential in the number of monadic binds.

The marginalizing semantics of the sampling monad, however, involve simply *drawing a sample*. We sample first from some distribution and then use that value to sample from another distribution depending on it. The complexity in this case is additive, and thus linear in the number of monadic binds.

4.2.2 A Sampling-Based Embedded Language

The sampling monad implements a shallowly-embedded DSL in the same spirit as the Giry monad, and indeed the syntax around its functorial, applicative, and monadic structure are identical. An MIT-licensed version of this EDSL is available as the *mwc-probability* library [Tobin, 2014].

Zinkov [2012] enumerated a number of popular models used in applied statistics and formulated them using JAGS. It is instructive to compare some of those implementations to the programs that can be expressed by using monadic structure; the resulting code is typically shorter and clearer, with the added benefit of being composable and type-safe.

We can implement a simple Bayesian linear regression model like so:

```
regression obs = do
  a <- gaussian 0 10
  b <- gaussian 0 10
  v <- uniformR (0, 100)
  let model x = gaussian (a + b * x) (sqrt v)
  for obs model
```

The order of the monadic binds matters, in contrast to a language like BUGS/-JAGS, where the corresponding linear regression model can be expressed as follows (following the example of Zinkov [2012]):

```
model {
  for (i in 1:N){
    y[i] ~ dnorm(y.hat[i], tau)
    y.hat[i] <- a + b * x[i]
  }
  a ~ dnorm(0, 0.01)
```

```
  b ~ dnorm(0, 0.01)
  tau <- pow(sigma, -2)
  sigma ~ dunif(0, 100)
}
```

In the BUGS example we can use parameters like 'tau', 'a', and 'b' before we actually declare them, whereas in a monadic context we have to declare things from the top-down as we encounter them (in technical parlance, the monadic bindings are not *mutually recursive*). Also note the technique we use in the monadic code; we define 'model' to be a function that models the output for any given covariate input, and then effectfully map that function over the inputs using the 'for' function.

It's worth highlighting again the difference between the JAGS denotation of the model and what can be expressed by using a monadic structure. Models assembled by monadic binds are *composable*. In the regression example we can split the prior and likelihood into separate parameter and data models, and then combine them using a bind:

```
prior = do
  a <- gaussian 0 10
  b <- gaussian 0 10
  v <- uniformR (0, 100)
  return (a, b, v)

likelihood obs (a, b, v) = do
  let model x = gaussian (a + b * x) (sqrt v)
  for obs model

predictive obs = prior >>= likelihood obs
```

This style of (type safe) composition and reuse is not possible in a language like

BUGS, JAGS, or Stan. In the JAGS example above, the 'model' block cannot be used as an input to other models — it is a block of code that must stand on its own. It's not just useful for the abstract purpose of being modular; it can also be useful if one wishes to sample from the prior, rather than from the predictive distribution.

We can adapt the simple linear regression model to use arbitrary other basis functions. Here is a sinusoidal regression model, for example:

```
sinusoidal obs = do
  a <- gaussian 0 10
  b <- gaussian 0 10
  v <- uniformR (0, 100)
  let model x = gaussian (a * cos x + b * sin x) (sqrt v)
  for obs model
```

Arbitrary link functions can also be supported; here is an implementation of Bayesian logistic regression:

```
logistic x = 1 / (1 + exp (negate x))

logisticRegression obs = do
  a <- gaussian 0 10
  b <- gaussian 0 10
  let prob x = logistic (a + b * x)
  for obs (bernoulli . prob)
```

Recursive structures, like those found in autoregressive models, are expressed particularly naturally in a functional language. A purely functional language contains no primitive looping statements, so iteration must be done recursively. Consider the following ARIMA(1, 1, 1) model implemented in BUGS (again from Zinkov [2012]):

```
model {
  y[1] ~ dnorm(0,1e-5)
  eps[1] <- 0

  for (i in 2:T) {
     y[i] ~ dnorm(mu[i],tau)
     mu[i] <- w0 + w1*y[i-1] + x[i] + eps[i-1]
     eps[i] <- y[i] - mu[i]
  }

  w0 ~ dnorm(0, .0001)
  w1 ~ dnorm(0, .0001)
  tau <- pow(sigma, -2)
  sigma ~ dunif(0, 100)
}
```

We can implement this in our embedded language by using several monadic combinators from our host, Haskell. The model is as follows:

```
prior = do
  [w0, w1, y0] <- replicateM 3 (gaussian 0 1)
  s <- uniform
  return (w0, w1, s, y0)

likelihood obs (w0, w1, s, y0) = unfoldrM process (y0, 0, obs) where
  process (_, _, []) = return Nothing
  process (y, e, xs) = do
    let mu = w0 + w1 * y + head xs + e
    yn <- gaussian mu s
    return (Just (yn, (yn, yn - mu, tail xs)))

model obs = prior >>= likelihood obs
```

In particular the implementation uses the 'unfoldrM' combinator representing a *right-unfold* — a typical manner of producing a recursive structure in Haskell. 'unfoldrM' recursively calls the 'process' function which produces another iteration of the model if there are more inputs to process and terminates otherwise. We've also used the 'replicateM' function in the prior to call the standard normal function three times.

4.3 Preserving Model Structure

4.3.1 Motivation

Each of the probability monads we've looked at has an easily-interpretable marginalizing semantics that 'collapses' the graph of distributions implicitly described by a given program. Whenever we've composed two distributions together via the bind operator $\gg\!=$, we've performed some operation that propagates the uncertainty from the distribution left of the bind into the one on the right: in the beta-binomial distribution expressed by $\text{beta}(\alpha, \beta) \gg\!= \text{binomial}(n, \cdot)$, for example, the bind operator propagates the uncertainty captured by the beta distribution into the p parameter accepted by the binomial distribution. In the case of the Giry monad this marginalization occurs by direct integration, while for the sampling monad we simply sample and propagate the result directly (yielding a much more computationally efficient method of propagating uncertainty).

These marginalizing semantics are both by definition *lossy* in that, by marginalizing something, the monadic bind operator throws away information about the *structure* of the things it's binding together. It's easy to see that the overall structure of programs encoded by the Giry or sampling monads is identical, but what we can't do is *examine* this structure, as evaluating a term via 'integrate' or 'sample' collapses the programs to a single point (an expectation or sample over the

predictive distribution, respectively).

The structure is evident to us in the program that we write, but it's not amenable to analysis or manipulation. It is 'write-only', if you will. This is problematic for implementing arguably the most important and compelling feature of Bayesian statistics, inference. Consider Bayes' theorem set in the following form:

$$p(\text{causes} \mid \text{effects}) \propto p(\text{effects} \mid \text{causes}) p(\text{causes}) \tag{4.1}$$

If we have a model of our assumptions about the world $p(\text{causes})$ (encoded as a parameter model, or *prior*) and a model of the world conditional on those assumptions $p(\text{effects} \mid \text{causes})$ (the data model, or *likelihood*), then we can calculate the *posterior* distribution $p(\text{causes} \mid \text{effects})$ that models our assumptions about the world, given what we have observed from it.[2] The posterior is like our data model, but with the conditioning flipped around; extracting a posterior from a conditioned distribution can be thought of as *inverting* it. Equation 4.1 expresses the inversion at the distribution level, but given a structure for our model, we can recover a structure for the inverse model as well. The inversion is particularly easy to see if we visualize a distribution by its graph structure — we merely switch what nodes are conditioned on (see Figure 4.1). When doing inference we're typically interested in evaluating integrals over the posterior distribution, which in practice must usually be approximated by some form of Monte Carlo, and typically MCMC. **We need access to the structure of the underlying conditional distribution in order to implement many popular approximate inference algorithms**. In particular, for MCMC we typically need to propose transitions over parameters and also know how to calculate the probability of that proposal. It is difficult to do this without being able to analyze the internal structure of a distribution.

Defining, manipulating, and interpreting structured Bayesian models is the essence of *probabilistic programming*. A user of a probabilistic programming language

[2] Of course, we're typically only interested in calculating something *proportional* to the posterior, but we can usually ignore this detail.

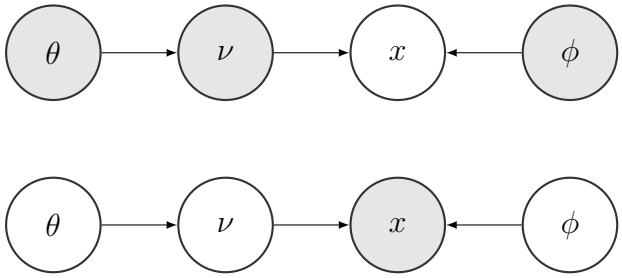


Figure 4.1: Comparing the structures of a data model/likelihood with a posterior. The likelihood is visualized at top; we condition on the parameters θ, ϕ, and ν and propagate information forwards into the unobserved data node x via the graph structure. The posterior is visualized below and has an inverted conditional structure relative to the likelihood; here we condition on the data x and propagate information backwards through the graph to the parameters.

writes a program that defines a structured model, and then the language supplies specialized interpreters that evaluate, compile, or otherwise analyze that structure in order to extract useful information from the model, whether that be by way of samples, probabilities, integrals, and so on. The model and its structural inverse lend itself to natural 'forward' and 'backward' interpretations of the program, and the dichotomy between the two is one of the most important concepts in probabilistic programming. Typically, probabilistic programming languages use a *sampling* interpretation for both directions; interpreting a model 'forwards' samples from the (prior) predictive distribution, while interpreting 'backwards' yields an approximate sample or samples from the posterior distribution. An efficient form of this backwards sampling interpretation over the posterior is the most difficult and important open problem in probabilistic programming, and Bayesian statistics more broadly [Mansinghka, 2009, Goodman and Stuhlmüller, 2014].

4.3.2 Towards A Deep Embedding

So: preserving the structure of a model is desirable. Our models are implemented as programs, and preserving structure will make the internals of the program open to examination and manipulation. This is necessary if we want to implement some important algorithms for performing approximate inference. Recall that both the measure and sampling-based DSLs have been *shallow embeddings*, in that the terms of the languages are defined by their respective semantics (integration and sampling, respectively). Now we'd like to move to a *deep embedding* in which the language is described in terms of its own *abstract syntax*.

Abstract syntax is the raw, uninterpreted data that describes a program. Consider the following example of an expression language for ring-operator arithmetic on integers:

```
data Arithmetic =
    Lit Int
  | Add Arithmetic Arithmetic
  | Sub Arithmetic Arithmetic
  | Mul Arithmetic Arithmetic
  deriving (Eq, Show)
```

It is an expanded version of the original example we saw in Chapter 2. This data type defines a tiny language consisting of four terms: 'Lit', 'Add', 'Sub', and 'Mul'. 'Lit' corresponds to a literal value and carries with it an integer, while the other three correspond to their respective arithmetic operators '$+$', '$-$', and '$*$'. Each of these carries two other values of type 'Arithmetic' with it such that the 'Arithmetic' type is defined recursively. Recall from Chapter 2 that the recursive definition reads exactly like the equivalent Backus-Naur Form for denoting context-free grammars:

```
<arithmetic> ::= lit <int>
               | add <arithmetic> <arithmetic>
               | sub <arithmetic> <arithmetic>
               | mul <arithmetic> <arithmetic>
```

Values of the 'Arithmetic' type form a binary-branching tree — indeed, such expressions are called *abstract syntax trees* or ASTs. We can write an value of the 'Arithmetic' type in GHCi:

```
> Sub (Mul (Lit 3) (Lit 5)) (Add (Lit 1) (Lit 4))
Sub (Mul (Lit 3) (Lit 5)) (Add (Lit 1) (Lit 4))
```

Notice that nothing 'happens' when the expression is entered — we just get handed back the raw data that we passed in. The abstract data do not carry any meaning until an interpreter assigns one; until then, they're just data. We can define a simple recursive interpreter to evaluate expressions of the 'Arithmetic' type in the standard way:

```
eval :: Arithmetic -> Int
eval (Lit j)     = j
eval (Add e0 e1) = eval e0 + eval e1
eval (Sub e0 e1) = eval e0 - eval e1
eval (Mul e0 e1) = eval e0 * eval e1
```

And we can then use it to interpret our previous expression:

```
> eval (Sub (Mul (Lit 3) (Lit 5)) (Add (Lit 1) (Lit 4)))
10
```

The 'eval' function does exactly what we expect, recursively evaluating the expression until we're left with a final, single integer value. Note that by evaluating

an expression, we necessarily lose the structure attached to it — the same issue faced by the probability monads implemented thus far. But with the structure of the expression accessible, we can examine it or apply arbitrary transformations to it. Consider the following function that converts all instances of multiplication to addition, for example:

```
mulToAdd :: Arithmetic -> Arithmetic
mulToAdd (Add e0 e1) = Add (mulToAdd e0) (mulToAdd e1)
mulToAdd (Sub e0 e1) = Sub (mulToAdd e0) (mulToAdd e1)
mulToAdd (Mul e0 e1) = Add (mulToAdd e0) (mulToAdd e1)
mulToAdd e = e
```

This interpreter transforms a program into another program, while preserving structure. Applying it to an expression yields the predictable result:

```
> mulToAdd (Sub (Mul (Lit 3) (Lit 5)) (Add (Lit 1) (Lit 4)))
Sub (Add (Lit 3) (Lit 5)) (Add (Lit 1) (Lit 4))
```

So we'd like to define a suitable abstract syntax for our embedded probabilistic language in the same spirit as we've done for this toy language for arithmetic. The catch is that we'd like to write our model in much the same way as we did before, reusing primitives like do-notation, function application, variable binding, and so on from the host language. But instead of marginalizing to some element describing the predictive distribution as did the Giry and sampling monads, we want our model definition to *generate* abstract syntax that we can later interpret.

4.3.3 Algebraic Freeness and the Free Monad

This section presents an elegant way to capture abstract syntax from a monadic program, using the properties of algebraic *freeness*. Freeness corresponds to a

kind of *genericness* or *minimalness* in that a free 'foo' is a sort of template 'foo'; a foo that obeys the minimum possible laws required to be a foo, and nothing more [Piponi, 2014].

As an example, consider a *monoid* over a set. A monoid is an algebraic structure that is characterized by the following properties:

- A set S.
- An associative binary operator $+$ over S such that for all s_0, s_1 in S, $s_0 + s_1$ is also in S.
- A unique identity element 0 in S such that for any s in S, $s + 0 = 0 + s = s$.

Two simple examples of monoids are the integers under addition and the integers under multiplication. In the former case the identity element is 0, and in the latter it is 1. But neither of these are *free* monoids; note that they each obey at least one other property — commutativity — that is not included in their definition.

The free monoid happens to be the set of *finite sequences* of elements from some other set A, where $+$ concatenates sequences together and the identity element is the empty sequence $\{\}$. Concatenation is associative (as required) but not commutative, satisfying exactly the minimal properties required to be a monoid, but no more. In Haskell the free monoid can be represented by the list type '[a]' of lists containing values of type 'a'.

Free constructions are *structure-preserving* in a sense. Take the monoid of the integers under addition, for example, where evaluating the expression $3+1$ yields 4. Working backwards from 4, we have no sense of what the initial structure of the expression was — it could have been $1 + 1 + 1 + 1$, $1 + 2 + 1$, or any other number of possible expressions. On the other hand, concatenating the sequences $\{1, 2\}$ and $\{3, 4\}$ yields $\{1, 2, 3, 4\}$. When working backwards from

the expression $\{1, 2, 3, 4\}$ we may not know exactly what elements were used to create the result — possibly $\{1\} + \{2\} + \{3, 4\}$, $\{1, 2, 3\} + \{4\}$, etc. But we do have some information as to the structure of the initial expression — namely, the elements of all non-empty sequences used to construct it, where the internal ordering of elements has also been preserved. If we did not have precisely this structure, then we could demonstrate that the free monoid failed to characterize a monoid in the required 'minimal' sense.[3]

A *free monad*, analogously, is a structure-preserving monad: evaluating a free monad does not destroy the information about what components were used to create it. The free monad happens to be the set of *finite trees* containing elements from a set A, rather than a sequence [Piponi, 2014]. This makes sense when considering the structure of a monadic expression: the result of any given monadic bind can be analyzed and a further course of action taken based on it, yielding potential branching behaviour that is not characteristic of the free monoid, for example. But just as the free monoid preserves information about the elements of nonempty sequences used in monoidal expressions, the free monad preserves information about the elements contained in any nonempty trees used to produce monadic expressions.

It is illustrative to define freeness formally, starting with the concept of *adjoint functors*. Two functors $F : C \to D$ and $G : D \to C$ are said to be adjoint if there exists a natural transformation:

$$\eta : 1_C \to G \circ F$$

that has the so-called *universal mapping property* that, for any $X \in C$, $Y \in D$, and $f : X \to G(Y)$, there exists a unique $g : F(X) \to Y$ such that

$$f = G(g) \circ \eta. \tag{4.2}$$

[3]Another way to say this is that the free monoid satisfies the monoid laws independent of the set used to generate it.

The functor F is then called the *left-adjoint* and G is called the *right-adjoint*, written $F \dashv G$.

A *forgetful functor*, often denoted U, is a functor from one category to another, such that the target category is a reduced version of the source in which some structure has been dropped (or 'forgotten'). A *free functor* is then left-adjoint to a forgetful functor, and a *free object* is an object created by a free functor. One can thus see why a free object is *minimal* in the sense of structure; a free functor takes a generic template object and *endows it* with just enough structure to be a member of a requisite category.

Looking at the free monoid example formally: if we have a forgetful functor $U : \mathbf{Mon} \to \mathbf{Set}$ and free functor $F : \mathbf{Set} \to \mathbf{Mon}$ such that $F \dashv U$, then for X an object of **Set** we have that the free monoid is an object $F(X)$ in **Mon**. Operationally, F creates a monoid over finite sequences of elements of X by endowing X with concatenation and identity, creating the monoid $(X, +, \{\})$. One can demonstrate that F is the functor that satisfies the universal mapping property of Equation 4.2 (see Awodey [2010] for a proof, for example).

Now, let $U : \mathbf{Monad} \to \mathcal{F}$ be a forgetful functor $(G, \mu, \eta) \mapsto G$ from the category of monads to the category of functors. It takes a monad (or monad morphism), to the underlying functor (or natural transformation), discarding the monadic structure encoded via the natural transformations μ and η. A free monad is an object in **Monad** constructed by applying Free : $F \to \mathbf{Mon}$ to a functor in $\mathcal{F}$.

In code: for a functor 'f', the free monad 'Free f' can be given an explicit definition in Haskell:

```
data Free f a =
    Free (f (Free f))
  | Pure a
```

It is a sum type that encodes a simple recursive structure that can terminate,

regardless of the characteristics of the functor 'f' that parameterizes it. This corresponds neatly to the structure of an abstract syntax tree: a program encoded via a free monad either continues evaluation for another step under the 'Free f' branch (and then via any branching structure defined in 'f'), or it terminates with result 'a' under the 'Pure' branch. 'Free f' is trivially a functor so long as 'f' is also a functor (the specific 'f' used is often called a *base functor*):

```
instance Functor f => Functor (Free f) where
  fmap f (Pure a) = Pure (f a)
  fmap f (Free g) = Free (fmap (fmap f) g)
```

By definition 'Free f' is also a monad (and thus an applicative functor):

```
instance Functor f => Monad (Free f) where
  return       = Pure
  Free f >>= g = Free (fmap (>>= g) f)
  Pure a >>= f = f a
```

This is precisely the minimal structure required to make a functor 'f' into a monad. And since the free monad is a kind of 'template' monad, we can use it to do all of our usual monadic, functorial, and applicative tricks — it automatically satisfies the functor and monad (and thus applicative) laws [Swierstra, 2008].

The free monad is well-supported in Haskell via the *free* library [Kmett, 2008a], which contains both the free monad implementation above and also an asymptotically more efficient *Church encoding* of the free monad that we won't discuss here. Additionally, it provides a number of useful functions for working with free monads. One particularly useful function is called 'liftF', defined as follows:

```
liftF :: Functor f => f a -> Free f a
liftF f = Free (fmap Pure f)
```

The *free* library also defines the following function, 'iterM':

```
iterM :: (Monad m, Functor f) => (f (m a) -> m a) -> Free f a -> m a
iterM _   (Pure x) = return x
iterM phi (Free f) = phi (fmap (iterM phi) f)
```

We'll make use of both of these in the next section. In particular, 'iterM' is a kind of *recursion scheme*: a structured way of traversing, producing, or consuming a recursive structure. Recursion schemes are elegant and useful patterns for expressing computation, but a discussion of them is beyond the scope of this dissertation: see Meijer et al. [1991] for the seminal reference, or Tobin [2015] for a brief and informal overview.

4.4 A Concrete Deep Embedding

Previously we defined functions 'beta', 'bernoulli', 'gaussian', etc. to represent terms in each of our shallowly-embedded DSLs. In this section we'll define these distributions as different *constructors* of a sum type that will constitute the base functor for the free monad.

To provide theoretical justification for this idea, recall the category of measurable spaces **Meas**. The Giry monad $(\mathcal{P}, \mu, \eta)$ is defined around the endofunctor $\mathcal{P}$ that takes a measurable space $M \in$ **Meas** to the space of probability measures $\mathcal{P}(M) \in$ **Meas** on M. To keep the probabilistic semantics abstract, define a category of *abstract probability distributions* **Prob** such that there exists a one-to-one correspondence between every probability measure in $\mathcal{P}(M)$, for any $M \in$ **Meas**, and every abstract probability distribution in **Prob** (these abstract distributions can just be represented by unique names). This can be done by defining a functor $\mathcal{P}^* :$ **Meas** $\to$ **Prob** via the following rule: if $M \in$ **Meas**

is a probability measure, then let $\mathcal{P}^*(M)$ be a corresponding abstract probability distribution; if it is some other measurable space, let $\mathcal{P}^*(M)$ be the abstract probability distribution corresponding to $\mathcal{P}(M)$ (and note that the definition of morphisms follows in kind).

Now, given the forgetful functor $U : \textbf{Monad} \to \mathcal{F}$, the left-adjoint free functor $F : \mathcal{F} \to \textbf{Monad}$ can be used to convert a functor into a monad. The monad $F\mathcal{P}^* = (\mathcal{P}^*, \mu, \eta)$ is then free monad over abstract probability distributions, where μ encodes abstract marginalizing semantics and η encodes a Dirac distribution.

With a hat tip to probability theory, these abstract probability distributions are probably best understood as *laws*. We can construct a base functor that encodes an arbitrary number of abstract probability distributions as follows:

```
data Prob r =
    Beta Double Double (Double -> r)
  | Bernoulli Double (Bool -> r)
  | Gaussian Double Double (Double -> r)
  | ...
  deriving Functor
```

Note the structure of this type: it has a number of sum type constructors, each corresponding to a particular law, and each constructor also carries a number of typed parameters. 'Beta Double Double' corresponds to a beta distribution with the two usual real parameters — α and β, respectively. 'Bernoulli Double' similarly corresponds to a Bernoulli distribution with the usual probability parameter p. Each constructor also carries a function type as its final field: for example, 'Gaussian' contains a field with type 'Double -> r'. The type to the left of the arrow in this function type – in the 'negative position' – denotes the support that the distribution is defined over: observe that the Gaussian and beta

distributions are defined over the reals, while the Bernoulli distribution is defined over the Boolean set $\{\text{True}, \text{False}\}$. On the right hand of the arrow in the function type – the 'positive position' – is the type parameter of the functor.

Since the type parameter can only occur in the positive position, this type is guaranteed to be a functor. GHC can thus derive the Functor instance for us — as is done above — but it is illustrative to demonstrate what the functor instance looks like:

```
instance Functor Prob where
  fmap f term = case term of
    Beta a b r    -> Beta a b (f . r)
    Bernoulli p r -> Bernoulli p (f . r)
    ...
```

Note that the structure of the definition at each branch is the same — we just compose the mapped function with the continuation at each branch and proceed to the next one. The following thus suffices to verify that the functor laws hold:

```
  fmap id r
    = id . r
    = r

  fmap (f . g) r
    = (f . g) . r
    = f . g . r
    = fmap f (g . r)
    = fmap f (fmap g r)
    = (fmap f . fmap g) r
```

The 'Prob' type corresponds to $\mathcal{P}^*$, and applying the free functor 'Free' to it yields the free monad $F(\mathcal{P}^*)$ that we'll call 'Model', for 'probabilistic model':

```
type Model = Free Prob
```

This is our abstract, structure-preserving probability monad. Since the base functor satisfies the functor laws, an appeal to our argument from Section 4.3.3 demonstrates that 'Model' satisfies the functor and monad laws.

When wrapped up in 'Free', the 'r' that parameterized 'Prob' indicates recursive points. 'Free Prob' terms are thus fixed-points that capture the recursive structure 'Free (Prob (Free (Prob (Free ...))))' of a probabilistic program. Concretely, we can now assemble values of type 'Model a', such as the following translation of the beta-bernoulli model:

```
betaBernoulli :: Double -> Double -> Model Bool
betaBernoulli a b =
  Free (BetaF a b (\p0 ->
    Free (BernoulliF p0 (\p1 ->
      Pure p1))))
```

Writing the model in this way corresponds to manually assembling its abstract syntax, which isn't so user-friendly. But we can exploit the monadic structure of the free monad in order to make this convenient. Define the following two 'smart constructors' using the 'liftF' function described previously:

```
beta :: Double -> Double -> Model Double
beta a b = liftF (BetaF a b id)

bernoulli :: Double -> Model Int
bernoulli p = liftF (BernoulliF p id)
```

And now we can use conventional do-notation to stitch our model together:

```
betaBernoulli :: Double -> Double -> Model Bool
betaBernoulli a b = do
  p <- beta a b
  bernoulli p
```

This program has the same abstract syntax as the previous example, but it is every bit as user-friendly and declarative as were the shallowly-embedded Giry and sampling monads. The difference is that expressions of type 'Model' evaluate to pure, uninterpreted, structured data.

The free monad thus allows us to create a deep embedding for our probabilistic programming language — expressions in the embedded language now describe probabilistic programs without ascribing any particular probabilistic interpretation to them *a priori*. A value with type 'Free f' represents a *program*, and the functor 'f' that parameterizes it can be fruitfully interpreted as the available *instruction set* that can be used to define that program. Our probabilistic base functor thus represents a collection of *probabilistic instructions* that can be assembled together to create a probabilistic program. Indeed, expressions now represent their uninterpreted abstract syntax directly, and we can ascribe various probabilistic interpretations to them after-the-fact. Meanwhile the terms and syntax of the embedded language remain typed, composable, and user-friendly.

This general technique for embedding a monadic probabilistic programming language is as follows:

- Define a base functor like 'Prob', parameterized over some abstract type variable. The base functor contains any number of constructors, each corresponding to some known probability distribution with parameters of the appropriate types over some known support type. The constructors have the form

  ```
  <Distribution_0> <p_0> .. <p_n> (<support_0> -> k)
  ```

```
  | <Distribution_1> <p_0> .. <p_m> (<support_1> -> k)
  | ..
```

These constructors represent proto-terms of our embedded language. Put equivalently: the terms of our embedded language will be constructed from abstract, named representations of probability distributions.

- Construct a free monad from the base functor by wrapping it in the 'Free' type defined previously, creating a type like 'Model' that denotes terms of the embedded language.
- Create user-friendly terms for the embedded language for defining them via 'liftF':

```
<distribution_0> <p_0> .. <p_n> =
  liftF (<Distribution_0> <p_0> .. <p_n> id)

<distribution_1> <p_0> .. <p_m> =
  liftF (<Distribution_1> <p_0> .. <p_m> id)

..
```

The result is a simple and elegant embedded monadic language for working with probability distributions. Arbitrary interpreters corresponding to other probability monads can easily be constructed by using various recursion schemes available to free monads (for example the 'iterM' function previously).

A version of this language is available as the MIT-licensed *deanie* library [Tobin, 2017], and an older version is similarly available under the name *observable* [Tobin, 2013c].

4.4.1 Forward-Mode Interpretation

It is easy to use the aforementioned technique to implement any number of minimal languages based on some underlying probabilistic instruction set. Using the distributions we've been focusing on thus far as our instruction set, here is the core of a very simple embedded language, for example:

```
data Prob r =
    Bernoulli Double (Bool -> r)
  | Beta Double Double (Double -> r)
  | Gaussian Double Double (Double -> r)
  deriving Functor

type Model = Free Prob

bernoulli :: Double -> Model Bool
bernoulli p = liftF (Bernoulli p id)

beta :: Double -> Double -> Model Double
beta a b = liftF (Beta a b id)

gaussian :: Double -> Double -> Model Double
gaussian m s = liftF (Gaussian m s id)
```

These abstract distributions can be viewed as foundational in a sense; they can be used to create other, more complex distributions:

```
uniform :: Model Double
uniform = beta 1 1

binomial :: Int -> Double -> Model Int
binomial n p = fmap count coins where
```

```
  count = length . filter id
  coins = replicateM n (bernoulli p)

betaBinomial :: Int -> Double -> Double -> Model Int
betaBinomial n a b = do
  p <- beta a b
  binomial n p
```

The decision as to which 'foundational' probability distributions to include in the underlying instruction set, exactly, is left open. It should probably consist of a minimal collection of core distributions that can easily be used to create others, in the spirit of Park et al. [2008]. The exponential family might constitute a good sweet spot of foundational distributions, for example.

Since the free monad is a kind of template for monads, we can reuse the monadic structure of the shallowly-embedded measure- and sampling function-based languages here via the following interpreters:

```
-- measure-based
measure :: Model a -> Measure a
measure = iterM alg where
  alg (Bernoulli p r)  = Measurable.bernoulli p >>= r
  alg (Beta a b r)     = Measurable.beta a b >>= r
  alg (Gaussian m s r) = Measurable.gaussian m s >>= r

-- sampling-function based
rvar :: Model a -> MWC.Prob IO a
rvar = iterM alg where
  alg (Bernoulli p r)  = MWC.bernoulli p >>= r
  alg (Beta a b r)     = MWC.beta a b >>= r
  alg (Gaussian m s r) = MWC.gaussian m s >>= r
```

Here the 'iterM' recursion scheme simply collapses a free monad expression from the top down. Every abstract probability distribution is replaced with a corresponding measure or sampling representation, which is then bound to the continuation 'r' using the appropriate measure- or sampling-based marginalization semantics.

A simple Gaussian mixture makes for a useful and aesthetically-pleasing demonstration of these interpreters. We can define it like so:

```
mixture :: Double -> Double -> Model Double
mixture a b = do
  p      <- beta a b
  accept <- bernoulli p
  if   accept
  then gaussian (negate 2) 0.5
  else gaussian 2 0.5
```

The 'rvar' interpreter takes a model and converts it into a sampling function-based program that can be executed to produce a sample from the predictive distribution of the model (see Figure 4.2). Similarly, the 'measure' interpreter takes a model and produces the corresponding predictive measure, which can then be queried as desired (see Figure 4.3). Here we make use of the 'iterM' recursion scheme, which consumes a free monad from the top-down. It allows us to express an ancestral pass over a model, resulting in a sample from or measure over the predictive distribution respectively.

4.4.2 Backward-Mode Inference

To encode a 'backward' sampler we can reach for any number of approximate inference algorithms. The simplest is a *rejection sampler*, which produces a sample, checks it against some predicate, and retains or rejects the sample accordingly:

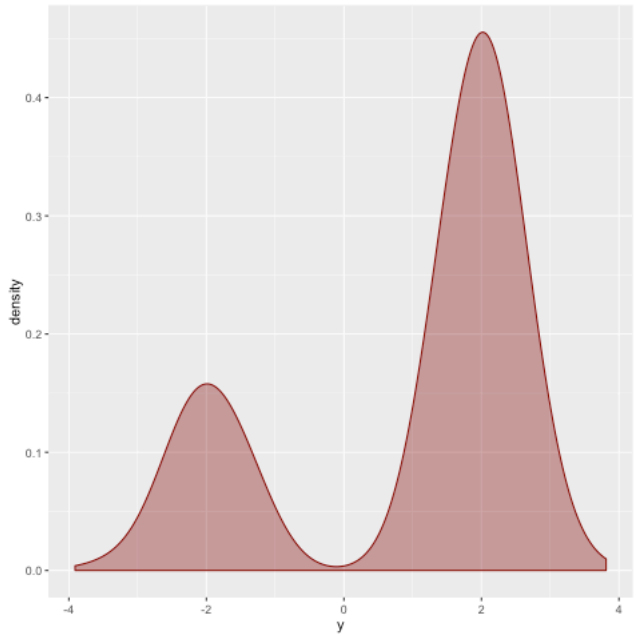


Figure 4.2: Kernel density estimate of the simple Gaussian mixture model defined by 'mixture 1 3', constructed from 1000 samples drawn via the forward-mode 'rvar' interpreter.

```
grejection
  :: (Foldable f, Monad m)
  => ([a] -> [b] -> Bool) -> f b -> m c -> (c -> m a) -> m c
grejection predicate observed proposal model = loop where
  len  = length observed
  loop = do
    parameters <- proposal
    generated  <- replicateM len (model parameters)
    if   predicate generated (Foldable.toList observed)
    then return parameters
    else loop

rejection
  :: (Foldable f, Monad m, Eq a)
```

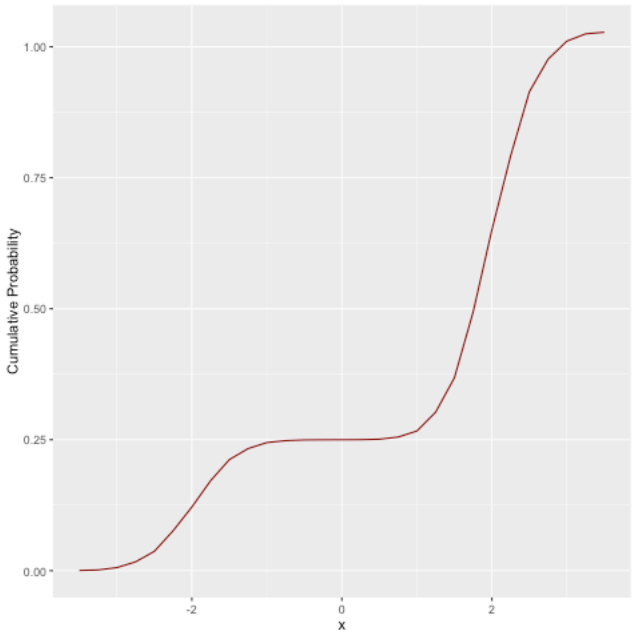


Figure 4.3: Cumulative distribution function (CDF) for the simple Gaussian mixture model defined by 'mixture 1 3', recovered via the 'cdf' query composed with the forward-mode 'measure' interpreter.

```
    => f a -> m b -> (b -> m a) -> m b
rejection = grejection (==)
```

The 'grejection' interpreter is a generalized rejection sampler that takes a generic predicate as an argument, while 'rejection' is a rejection sampler simply specialized to check for equality. Each also requires a collection of observations to condition on, as well as a parameter and data model. Note that we don't actually need to concretely refer to our 'Model' type here — this kind of rejection sampling can be defined in a totally abstract fashion such that it works for *any* monad.

To illustrate, here is a distribution for a simple Bernoulli model conditioned on nine 'True' and three 'False' values, approximated via rejection sampling. We

repeatedly propose parameters p from a uniform distribution over $(0, 1)$ and use them to generate a sample from the model. If a proposed parameter successfully generates a sample with the same number of 'true' values as found in the observations, the sample is not rejected:

```
obs :: [Bool]
obs =
  [ True, True, True, False, True, True
  , True, False, True, True, True, False
  ]

rinverse :: Model Double
rinverse = grejection count obs uniform bernoulli where
  count = length . filter id
```

Note that this distribution has the type 'Model Double', so like anything else we can interpret it into a sampling function using the 'rvar' interpreter in order to draw samples from it (see Figure 4.4):

```
> replicateM 1000 (simulate (rvar rinverse))
```

We can similarly encode an importance sampling interpreter. Note again that we can actually write this in terms of *any* monad:

```
importance
  :: (Foldable f, Monad m, Floating w)
  => f a -> m b -> (b -> a -> w)
  -> m (w, b)
importance obs proposal likelihood = do
  parameter <- proposal
  let cost = L.fold (L.premap (likelihood parameter) L.sum) obs
  return (exp cost, parameter)
```

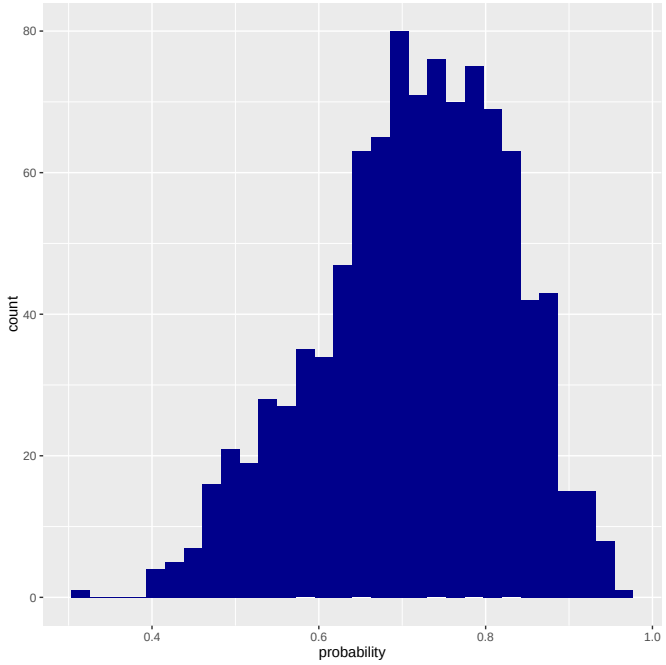


Figure 4.4: Histogram of 1000 samples from the inverse distribution of Bernoulli parameters when the model is conditioned on nine 'True' and three 'False' values.

The 'importance' interpreter requires a collection of observations to condition a model on, a parameter model to make proposals, and a weighting function (typically a log-density) to calculate importance weights for the proposed parameters. Note that the 'L' qualifier refers to helper functions from the *foldl* library [Gonzalez, 2013] that we can otherwise ignore.

Unlike the rejection sampling interpreter, the importance sampling interpreter also returns an importance weight for every sample drawn from it. We can write the appropriate weighting function and importance sampling-encoded inverse model for the simple Bernoulli model like so:

```
logDensityBernoulli :: Double -> Bool -> Double
logDensityBernoulli p x
  | p < 0 || p > 1 = log 0
```

```
  | x              = log p
  | otherwise      = log (1 - p)

iinverse :: Model (Double, Double)
iinverse = importance obs uniform logDensityBernoulli
```

Sampling from the inverse model yields a collection of weighted samples of parameter values for the conditioned Bernoulli distribution. We can just apply a weighted average to those in order to estimate the expected parameter:

```
> samples <- replicateM 1000 (simulate (rvar iinverse))
> waverage samples
0.7204870938868593
```

As a final example we can implement a variant of Metropolis-Hastings, a stateful MCMC algorithm which has historically been the baseline for reliable approximate inference in Bayesian statistics. It has the following straightforward implementation:

```
data MHP a = MHP {
    n       :: Int
  , current :: a
  , ccost   :: Double
  }

metropolis
  :: Foldable f
  => Int -> f a -> Model b -> (b -> a -> Double)
  -> Model [b]
metropolis epochs obs prior model = do
    current <- prior
```

```
    unfoldrM mh MHP { n = epochs, ccost = cost current, .. }
  where
    cost param = L.fold (L.premap (model param) L.sum) obs

    mh MHP {..}
      | n <= 0    = return Nothing
      | otherwise = do
          proposal <- prior
          let pcost = cost proposal
              prob  = moveProbability ccost pcost
          accept <- bernoulli prob
          let (nepochs, nlocation, ncost) =
                if   accept
                then (pred n, proposal, pcost)
                else (pred n, current, ccost)
          return (Just (nlocation, MHP nepochs nlocation ncost))

moveProbability :: Double -> Double -> Double
moveProbability current proposal =
    whenNaN 0 (exp (min 0 (proposal - current)))
  where
    whenNaN val x
      | isNaN x   = val
      | otherwise = x
```

Note that we encode the state of the algorithm in its own data structure, 'MHP'. The algorithm implementation itself follows the same 'unfoldrM' recursion scheme pattern used to define the autoregressive model in Section 4.2.2 (this turns out to be a common pattern for encoding stochastic processes in this type of embedded language). The kernel of the algorithm is encoded in the local 'mh' binding. This is also the first sampling interpreter that must be specified in terms of the 'Model' monad, since we need to make a Bernoulli draw in the accept/reject step

of the algorithm itself. As an implementation detail that we also use in the sequel, note that we use GHC's 'RecordWildCards' extension to use syntax like 'MHP {..}' that allows us to easily unpack and use the records of a product type like 'MHP'.

Given the 'metropolis' interpreter we can encode another inverse of the running Bernoulli model as follows. This one is encoded via 10000 iterations of the Metropolis-Hastings algorithm:

```
minverse :: Model [Double]
minverse = metropolis 10000 obs uniform logDensityBernoulli
```

Note that this is actually a distribution over the results of those 10000 MH iterations, so sampling *once* from this model via the 'rvar' interpreter will return a collection of 10000 samples. Otherwise, the procedure is the same as always:

```
> samples <- sample (rvar minverse)
```

The trace of the Markov chain is displayed in Figure 4.5.

4.5 Working with Structure

Recall the simple mixture model from Section 4.4.1, repeated below:

```
mixture :: Double -> Double -> Model Double
mixture a b = do
  p      <- beta a b
  accept <- bernoulli p
  if   accept
  then gaussian (negate 2) 0.5
  else gaussian 2 0.5
```

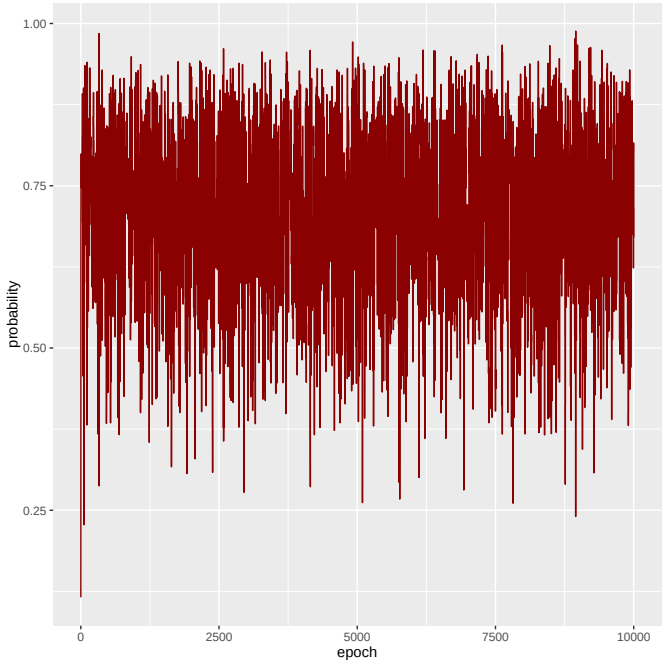


Figure 4.5: Trace of 10000 iterations of the Metropolis-Hastings sampler over the inverse distribution of Bernoulli parameters when the model is conditioned on nine 'True' and three 'False' values. The chain moves as expected (making proposals from the prior).

This model has two internal parameters; a beta-distribution probability parameter p and a Bernoulli(p)-distributed parameter denoted by 'accept'. Its structure is visualized in Figure 4.6. It's important to note that in this embedded framework, the only pieces of the syntax tree that we can observe are those related directly to our primitive instructions. For our purposes this is excellent — we can focus on programs entirely at the level of their probabilistic components, and ignore the deterministic parts that might otherwise distract.

Since sampling is lossy, the Metropolis-Hastings implementation from the previous section doesn't give us a way to observe, perturb, or otherwise interpret the internal Bernoulli parameter when performing inference. Like the rejection and importance samplers, we had to divide things into a parameter model (to

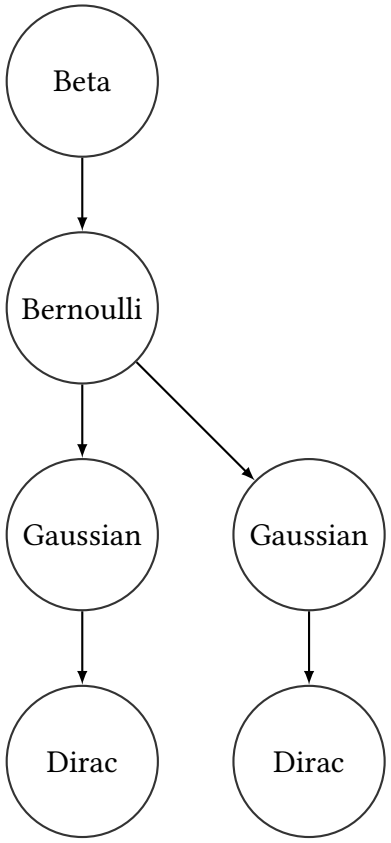


Figure 4.6: A visualization of the core probabilistic structure of the simple Gaussian mixture model, in terms of its AST. A probability p is beta-distributed according to some supplied hyperparameters, and then that probability is used to denote a Bernoulli(p)-distributed coin flip. The program branches according to the coin flip, where each branch denotes a distinct Gaussian distribution. The program then terminates at each branch via an implicit Dirac distribution depending on that branch's Gaussian.

generate proposals) and data model (to score the generated samples, or accept/reject a proposed move) such that we are only able to make proposals from the prior or parameter model. This is unsatisfying as it yields very little control over the inference algorithm; by using the prior to propose moves we can be prone to making overly-aggressive proposals that are unlikely to be accepted. It is typically more productive to perturb the state of the Markov chain in parameter

space *only slightly* in order to achieve an efficient proposal acceptance rate.

4.5.1 Algebraic Cofreeness and the Cofree Comonad

We've demonstrated that the free monad provides a convenient way to generate abstract syntax for a model. But there exists another structure called the *cofree comonad* that can be used to implement approximate inference algorithms. In this section we'll briefly develop the cofree comonad and then describe how it can be used for inference.

The cofree comonad is the categorical dual of the free monad — it corresponds to a structure in the *opposite category* defined by reversing the direction of all morphisms and interchanging the order of composition.

A *comonad* is a triple (F, δ, ϵ) such that $\delta : F \to F^2$ is a 'duplicating' natural transformation and $\epsilon : F \to I$ is a 'coidentity' or 'counit'. Note that both of these natural transformations are simply reversed versions of their monadic counterparts. The monadic bind operator $\gg=$ also has a comonadic analogue in the following operator, called *extend*:

$$\ggg: F(M) \to (F(M) \to N) \to F(N).$$

Akin to monads, comonads in Haskell are implemented via the 'Comonad' typeclass, defined as follows:

```
class Functor w => Comonad w where
  extract   :: w a -> a          -- 'counit'
  duplicate :: w a -> w (w a)    -- 'cojoin'
```

The 'extend' function corresponding to a flipped-argument form of $\ggg$ is then defined like so:

```
extend :: Comonad w => (w a -> b) -> w a -> w b
extend f = fmap f . duplicate
```

Comonads must obey certain laws — namely, those dual to the monad laws. They are most succinctly expressed in terms of morphisms of the form[4] $wa \to b$ under the comonadic function composition operator '=>=', defined as:

```
(=>=) :: Comonad w => (w a -> b) -> (w b -> c) -> w a -> c
f =>= g = g . extend f
```

Thus defined, we can state the comonad laws as the familiar associative and identity requirements. For appropriate f, g, h we have:

```
extract =>= f    =  f                    -- left-identity
f =>= extract    =  f                    -- right-identity
(f =>= g) =>= h  =  f =>= (g =>= h)      -- associativity
```

Dual to a free functor, a *cofree functor* is *right*-adjoint to a forgetful functor. So defining a forgetful functor U : **Comonad** $\to \mathcal{F}$ from the category of comonads to the category of functors, a right-adjoint functor $F : \mathcal{F} \to$ **Comonad** is one that endows a functor with the minimum required structure to be a comonad. This is the *cofree comonad* we're looking for.

In code, we can define the cofree comonad of a functor 'f' as follows:

```
data Cofree f a = Cofree a (f (Cofree f))
```

Note that the free monad is a *sum type* and the cofree comonad is a *product type*. But they each have a similar recursive structure. Whereas a free monad

[4] These are the morphisms of the so-called *co-Kleisli* category of the comonad.

represents a syntax tree that can terminate independently of its base functor, a cofree comonad represents an *annotated* syntax tree that can only terminate according to its base functor. Like the free monad, the cofree comonad is well-supported in the Haskell ecosystem by the *free* library, which encodes it in using the following infix constructor ':<' that we'll refer to in the sequel:

```
data Cofree f a = a :< f (Cofree f)
```

Like the free monad, the cofree comonad of any functor is automatically a comonad, and thus satisfies the requisite functor and comonad laws [Uustalu and Vene, 2008].[5]

If we consider the functor $\mathcal{P}^*$ from Section 4.4 that takes a measurable space to the space of abstract probability distributions over it, we can construct the cofree comonad $(\mathcal{P}^*, \delta, \epsilon)$. The idea here is that **we can use the cofree comonad of the abstract probability functor in order to annotate syntax trees with information about execution state**, such as likelihood value, current position in parameter space, random seeds, histories, and so on. The counit ϵ then extracts this execution state. A cofree-wrapped 'Prob' functor can thus be used to encode the *execution trace* of a probabilistic program in the sense of Wingate et al. [2011], for example. Indeed, an execution trace has the following type:

```
type Execution a = Cofree (Prob a) Node
```

where 'Node' is a type that encodes all the execution information we need in order to implement our inference algorithm of choice. Figure 4.7 shows an annotated visualization of the mixture model shown in Figure 4.6.

[5]Note that we can simply appeal to categorical duality here — since a free monad satisfies the monad laws, the cofree comonad must satisfy the comonad laws.

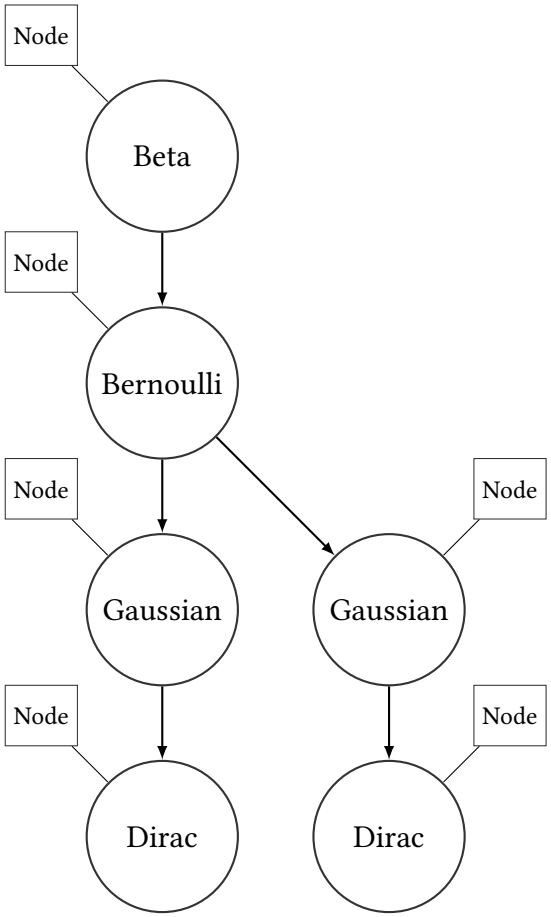


Figure 4.7: A visualization of an execution trace of the Gaussian mixture model. Each primitive probabilistic instruction from the base functor is annotated with a data structure called 'Node' that stores execution information like parameter space position, likelihood value, and so on.

4.5.2 Representing Programs That Terminate

Recall our probabilistic instruction set 'Prob' consisting of abstract beta, Bernoulli, and Gaussian distributions. Note that it doesn't include a *terminating* instruction of its own; the last field of each of the branches is a continuation that is used to pass control to the 'rest' of the program from that point. When wrapped up

with the 'Free' type, we implicitly used the 'Pure' constructor — which has the semantics of a Dirac distribution — in order to terminate programs.

One can convert a free monad-encoded program into a cofree comonad-encoded program mechanically, so long as the underlying functor contains a terminating instruction. We can do this by adjusting the underlying base functor in order to include an abstract Dirac distribution, rather than by relying on the 'Free' type to provide it for us:

```
data Prob a r =
    Bernoulli Double (Bool -> r)
  | Beta Double Double (Double -> r)
  | Gaussian Double Double (Double -> r)
  | Dirac a
  deriving Functor
```

Note that we've added an extra type parameter to 'Prob'. Since the 'r' type parameter is used to denote recursive points, the 'a' parameter holds information about what type (if anything) a program terminates with. 'Dirac' is a terminating instruction because it doesn't contain a continuation — it's structurally equivalent to the 'Pure a' branch of 'Free' that we used previously. We can then wrap 'Prob' up with 'Free' just like before, except now there is an extra type parameter to lug around:

```
type Model a = Free (Prob a)

beta :: Double -> Double -> Model a Double
beta a b = liftF (Beta a b id)

-- ...
```

```
dirac :: a -> Model a b
dirac x = liftF (Dirac x)
```

Our interpreters can be augmented accordingly as well, whereas nothing changes at all from the user's perspective. Note that in 'Model a b', the 'a' type parameter can only be concretely instantiated via use of the terminating 'dirac' term. On the other hand, the 'b' type parameter is *unaffected* by the 'dirac' term; it can only be instantiated by the other nonterminating terms encoded in the base functor.

We can distinguish between terminating and nonterminating programs at the type level, like so:

```
type Terminating a = Model a Void

type Program b = forall a. Model a b
```

'Void' is the uninhabited type, brought into scope via 'Data.Void'. Any program that ends via a 'dirac' instruction *must* be 'Terminating', and any program that doesn't end with a 'dirac' instruction *can not* be 'Terminating'. We'll call a non-'Terminating' model a 'Program'. Note that the notion of termination used for programs here is extremely weak. A 'Terminating' program may fail to terminate, for example as in the following program:

```
looper :: Terminating a
looper = (loop 1) >>= dirac where
  loop a = do
    p <- beta a 1
    loop p
```

A less-churlish program may still fail to terminate with probability 1, for example. The only guarantee we have is that termination *can be expressed* at the

language level, which is weak, but suffices for transforming a free-encoded program into a cofree-encoded one. Note that all primitive probabilistic instructions aside from 'Dirac' are nonterminating (i.e. they contain a continuation accepting a value from their support), so 'Dirac' is the *only* terminating instruction we ever need concern ourselves with.

4.5.3 Running Markov Chains over Execution Traces

To write a comonadic 'backward-mode' sampling interpreter, we'll use a 'Cofree'-encoded model in the following manner:

- Sample from a parameter model, recording the way the program executed in order to return the sample that it did.
- Compute the cost (in the log-likelihood sense) of generating the provided observations, using the sample from the parameter model as input.
- Propose a new sample from the parameter model *by perturbing the way the program executed* and recording the new sample outputted by the program.
- Compute the cost of generating the provided observations using this new sample from the parameter model as input.
- Compare the costs of generating the provided observations under the respective samples from the parameter models.
- With probability depending on the ratio of the costs, flip a coin. If we see a head, we'll move to the new, proposed execution trace of the program. Otherwise we'll stay at the old execution trace.

This procedure generates a Markov chain over the space of possible execution traces of the program — essentially, plausible ways that the program could have

executed in order to generate the supplied observations. Implementations of Church [Goodman et al., 2008] use variations of this method to do inference, the most famous of which is a low-overhead transformational compilation procedure described by Wingate et al. [2011].

The following 'Node' type is what we'll use to describe the internal state of each parameter in the program:

```
data Node = Node {
    nodeCost    :: Double
  , nodeValue   :: Dynamic
  , nodeSeed    :: MWC.Seed
  , nodeHistory :: [Dynamic]
  } deriving Show
```

Since the parameters of any model may be at different types, we store them using the 'Dynamic' type from Haskell's 'Data.Dynamic' library that defers potential type errors until runtime.[6] Importantly, it also happens to be the case that one can't easily interleave monadic and comonadic effects, so we annotate each parameter in the model with its own random seed in order to do random number generation purely. The history of each position visited in parameter is also stored.

Initializing, perturbing, and scoring the traces can be done with similar functions that describe how to initialize, perturb, or score a given node by matching on the branches of the probabilistic instruction set. To initialize an execution trace we can use a function like the following:

```
initialize :: Typeable a => MWC.Seed -> Prob a b -> Node
initialize seed instruction = case instruction of
```

[6]Another option here would just be to define an explicit 'Parameter' type that encoded the supported value types of the embedded language: booleans, doubles, etc.

```
  Bernoulli p _ -> runST (do
    (nodeValue, nodeSeed) <- samplePurely (Prob.bernoulli p) seed
    let nodeCost    = logDensityBernoulli p (unsafeFromDyn nodeValue)
        nodeHistory = []
    return Node {..})

  -- other branches are handled similarly
  ..
```

The idea here is that we match on every possible branch of the base functor and describe a way to create a 'Node'. The 'samplePurely' function provides us with a 'nodeValue' for the node, as well as a 'nodeSeed', and we can compute the 'nodeCost' by applying the appropriate cost function to the sampled value. Note that here and in the sequel we make use of Haskell's so-called 'strict state monad' (visible via the 'runST' function) for pseudorandom number generation, but this is an implementation detail that is unnecessary to discuss further.

Given an 'initialize' function, we can transform a 'Terminating' program into an execution trace by simple recursion. We use the 'Terminating' information in the program's type and the 'absurd' function from 'Data.Void' to rule out the possibility of ever visiting the free monad's 'Pure' constructor:

```
execute :: Typeable a => Terminating a -> Execution a
execute = annotate defaultSeed where
  annotate seeds term = case term of
    Pure r -> absurd r
    Free instruction ->
      let (nextSeeds, generator) = xorshift seeds
          seed  = MWC.toSeed (V.singleton generator)
          node  = initialize seed instruction
      in  node :< fmap (annotate nextSeeds) instruction
```

It is worth explaining this in some detail.

Here we are essentially converting a 'Free'-encoded description of a program into a 'Cofree'-encoded description of a program. Each node in a free monad-encoded program carries information of *either* the type 'f r', for 'f' a base functor and 'r' a recursive point, *or* some arbitrary type 'a'. The 'f r' information is indicated by the constructor 'Free', and the information 'a' is indicated by the constructor 'Pure'. Dually, each node in a cofree-encoded program carries information about *both* the type 'f r' *and* some arbitrary type 'a'. Here both the 'f r' and 'a' information is captured at each node via the ':<' constructor; the 'a' information is positioned on its left and the 'f r' information on its right, yielding the form 'a :< f r'.

To convert from 'Free' to 'Cofree', we can recurse over the 'Free' description and mechanically replace each 'Free' constructor we find with the appropriate cofree constructor, ':<'. However we also need to provide an annotation for that constructor at *every node*. This is the 'Node' type that we annotate with. In the code above, the line containing:

```
in  node :< fmap (annotate nextSeeds) instruction
```

describes this recursive procedure, where 'node' is the appropriately-constructed annotation and 'fmap' lets us recurse as needed. Note that if we ever encountered a 'Pure' node while recursing, we would *not* be able to construct any corresponding node for the cofree-encoded program. This is why we have to rule out ever visiting a 'Pure' node statically in order to convert from 'Free' to 'Cofree', and we do this using the information captured in the 'Terminating' type signature. The 'absurd' function used for the 'Pure' branch above asserts, in a type-safe and thus verifiable manner, that it is *impossible* to encounter a 'Pure' node when recursing over any 'Terminating' program.

Note that the 'annotate' function used above requires an additional argument in

the form of a seed for the PRNG to be located at a given node. We use the xorshift PRNG of Marsaglia et al. [2003] in order to construct the seeds themselves purely and pseudorandomly.

So the 'execute' function lets us create an execution trace from a 'Terminating' program. To perturb a trace, we define a perturbing function similar to 'initialize' that describes how to perturb any given primitive instruction:

```
perturbNode :: Execution a -> Node
perturbNode (Node {..} :< instruction) = case instruction of
  Bernoulli p _ -> runST (do
    (nvalue, nseed) <- samplePurely (Prob.bernoulli p) nodeSeed
    let nscore = logDensityBernoulli p (unsafeFromDyn nvalue)
    return (Node nscore nvalue nseed nodeHistory))

  -- other branches are handled similarly
  ..
```

Here we perturb a 'Node' by sampling in some manner appropriate to the branch of the base functor we encounter, and then update the state of the node accordingly. We then use a comonadic 'extend' to apply this perturbation to an entire execution trace:

```
perturb :: Execution a -> Execution a
perturb = extend perturbNode
```

The intuition here is that when we 'extend' a function over an execution trace, the trace first gets duplicated in a comonadic context, meaning that each parameter in the trace becomes annotated with a view of *the rest of the execution trace* from that point forward (see Figure 4.8 for a visualization of this). We then map the

'extended' function over the duplicated trace, reducing it back down to a run-of-the-mill trace. In this case we're reducing a duplicated trace via the 'perturbNode' function that operates on the level of individual nodes.

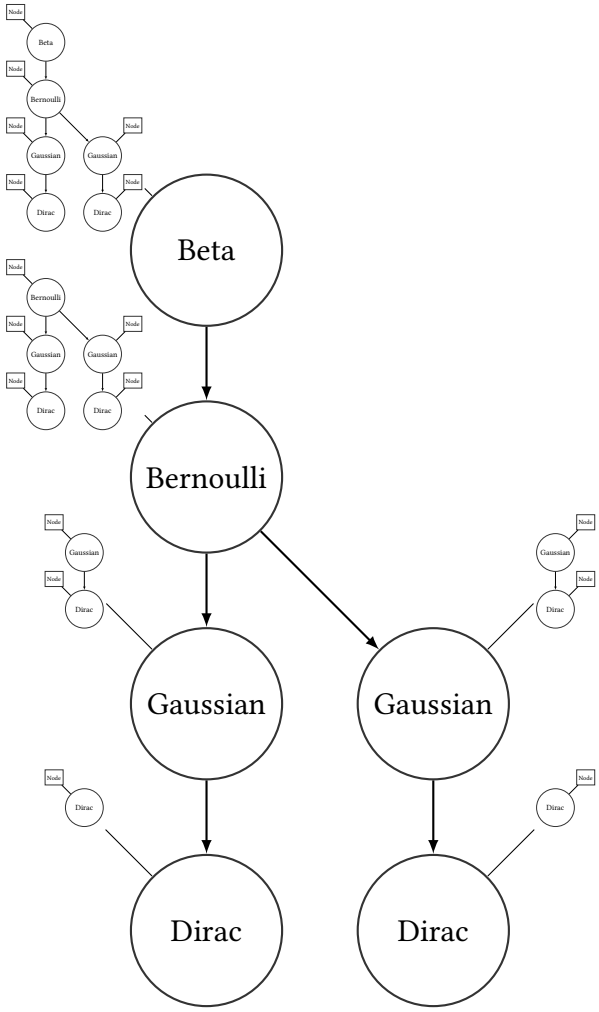


Figure 4.8: A visualization of an execution trace that has been duplicated in a comonadic context. Each primitive probabilistic instruction from the base functor becomes annotated with a view of the rest of execution trace from that point forwards.

To move around in trace space, we'll propose state changes by perturbing the current state, scoring them, and then accepting/rejecting proposals accordingly.

Eliding extra detail for readability, the comonadic backward-mode interpreter can be defined as follows (see Appendix A for additional detail):

```
invert
  :: (Eq a, Typeable a, Typeable b)
  => Int -> [a] -> Program b -> (b -> a -> Double)
  -> Program (Execution b)
invert epochs obs prior ll =
    loop epochs (execute (prior >>= dirac))
  where
    loop n current
      | n == 0    = return current
      | otherwise = do
          let proposal = perturb current

              -- <calculate costs and acceptance probability>

          accept <- bernoulli prob
          let next = if accept then proposal else stepGenerators current
          loop (pred n) (snapshot next)
```

Note that this interpreter returns a model over execution traces, so we can inspect or manipulate them at the embedded language level.

There are two important points to note in order to get this implementation right. Regardless of whether or not we accept a proposed move, we need to snapshot the current value of each node and add it to that node's history. This can be done using another comonadic extend:

```
snapshotValue :: Cofree f Node -> Node
snapshotValue (Node {..} :< cons) =
    Node { nodeHistory = history, .. }
```

```
    where
      history = nodeValue : nodeHistory

snapshot :: Functor f => Cofree f Node -> Cofree f Node
snapshot = extend snapshotValue
```

The other point is an extremely easy detail to overlook. Since we're handling random value generation at each node purely, using on-site random seeds, we need to iterate the generators forward a step in the event that we don't accept a proposal (otherwise we'd propose a new execution based on the same generator states used previously). This can be done by forcing a cheap sample at each node and then just throwing away the result, implemented via another comonadic extend:

```
stepGenerator :: Cofree f Node -> Node
stepGenerator (Node {..} :< cons) = runST (do
  (nval, nseed) <- samplePurely (Prob.beta 1 1) nodeSeed
  return Node {nodeSeed = nseed, ..})

stepGenerators :: Functor f => Cofree f Node -> Cofree f Node
stepGenerators = extend stepGenerator
```

4.5.4 Working With Execution Traces

To demonstrate how we can work with the comonadic MCMC implementation, let's examine samples from the inverse distribution of the mixture model achieved by conditioning its output on some observations. The inverse of a 'mixture 3 2' model can be encoded like so, iterating the Markov chain over executions 1000 times:

```
cinverse :: Program (Execution Bool)
cinverse = invert 1000 obs prior ll where
  obs = [ -1.7, -1.8, -2.01, -2.4, 1.9, 1.8]

  prior = do
    p <- beta 3 2
    bernoulli p

  ll left
    | left      = logDensityNormal (negate 2) 0.5
    | otherwise = logDensityNormal 2 0.5
```

Since we store the location history of each parameter in its associated metadata, a single sample from 'cinverse' will return a trace with each parameter's history cached on-site:

```
> execution <- sample (rvar cinverse)
```

We can step through the returned execution in order to examine its structure, using the stored values recorded at each parameter to 'automatically' step through execution, or we can supply our own parameter values to investigate arbitrary branches. The following 'step' function lets us walk through a trace step by step, using the current value stored at each node to proceed further (the more generic 'stepWithInput' allows us to supply our own input):

```
step :: Typeable a => Execution a -> Execution a
step prog@(Node {..} :< _) = stepWithInput nodeValue prog

stepWithInput :: Typeable a => Dynamic -> Execution a -> Execution a
stepWithInput value prog = case unwrap prog of
  Bernoulli _ r  -> r (unsafeFromDyn value)
```

```
Beta _ _ r     -> r (unsafeFromDyn value)
Gaussian _ _ r -> r (unsafeFromDyn value)
Dirac _        -> prog
```

Walking through the above trace, we can extract the information recorded about all parameters in the prior. Figures 4.9 and 4.10 show a trace and density of the mixing parameter p internal to the prior, while Figures 4.11 and 4.12 display similar information for the Bernoulli(p)-distributed mixture component.

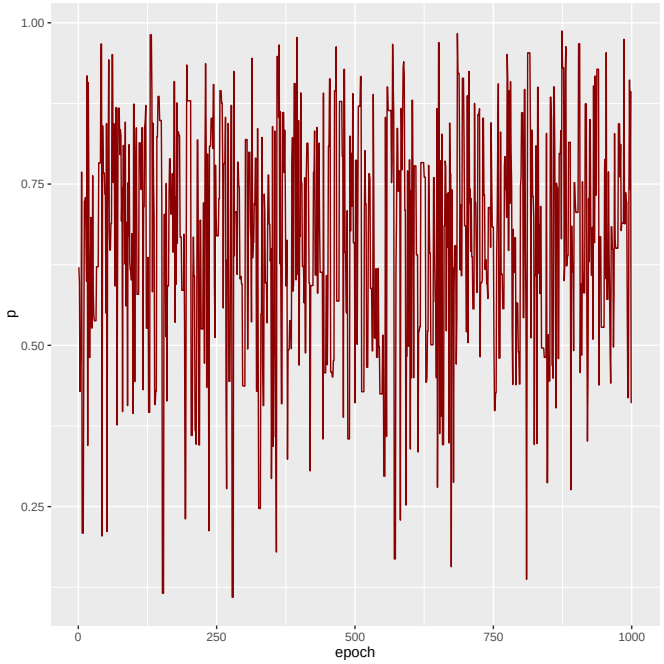


Figure 4.9: Positions of the mixing parameter p gathered from 1000 epochs of a Markov chain running over executions of the 'mixture 3 2' model conditioned on some observations. The chain moves as expected according to the perturbation function used (proposing moves from the prior).

Note that we can alter the 'perturbNode' function to support arbitrary perturbations to executions. In the above example we perturbed each parameter by sampling from the prior at that node, but we can instead sample according to

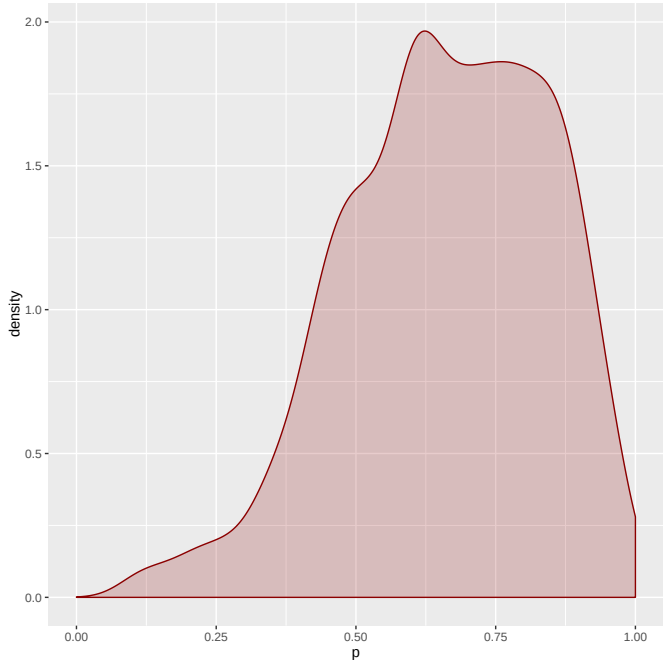


Figure 4.10: Kernel density estimate of the inverse distribution of the mixing parameter p gathered from 1000 epochs of a Markov chain running over executions of the 'mixture 3 2' model conditioned on some observations. The density has the expected shape given the beta$(3, 2)$ prior and mostly-negative observations.

Gaussian 'bubbles' with a given step size at each continuous parameter in single-site fashion, for example:

```
altPerturb :: Double -> Execution a -> Execution a
altPerturb eps = extend (perturbNode eps)

altPerturbNode :: Double -> Execution a -> Node
altPerturbNode eps (node@Node {..} :< cons) = case cons of
  BetaF a b _ -> runST (do
    let mval = unsafeFromDyn nodeValue :: Double
        val  = if mval < 0 || mval > 1 then 0 else mval
    (nvalue, nseed) <- samplePurely (Prob.normal val eps) nodeSeed
```

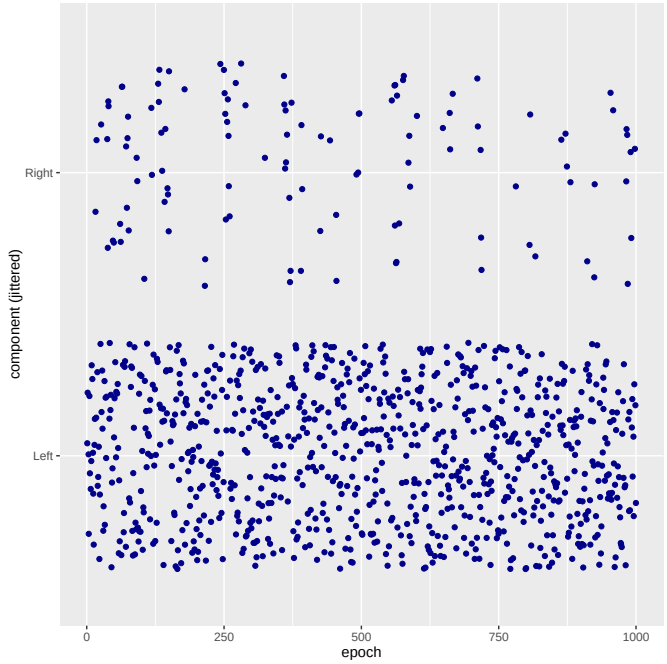


Figure 4.11: Jittered positions of the mixture component gathered from 1000 epochs of a Markov chain running over executions of the 'mixture 3 2' model conditioned on some observations. The chain tends to jump out of the rightmost component rapidly after entering it.

```
    let nscore   = logDensityNormal val eps (unsafeFromDyn nvalue)
        ndist    = Beta a b
    return (Node nscore nvalue nseed nodeHistory ndist))

  NormalF m s _ -> runST (do
    let val = unsafeFromDyn nodeValue :: Double
    (nvalue, nseed) <- samplePurely (Prob.normal val eps) nodeSeed
    let nscore   = logDensityNormal val eps (unsafeFromDyn nvalue)
        ndist    = Normal m s
    return (Node nscore nvalue nseed nodeHistory ndist))
```

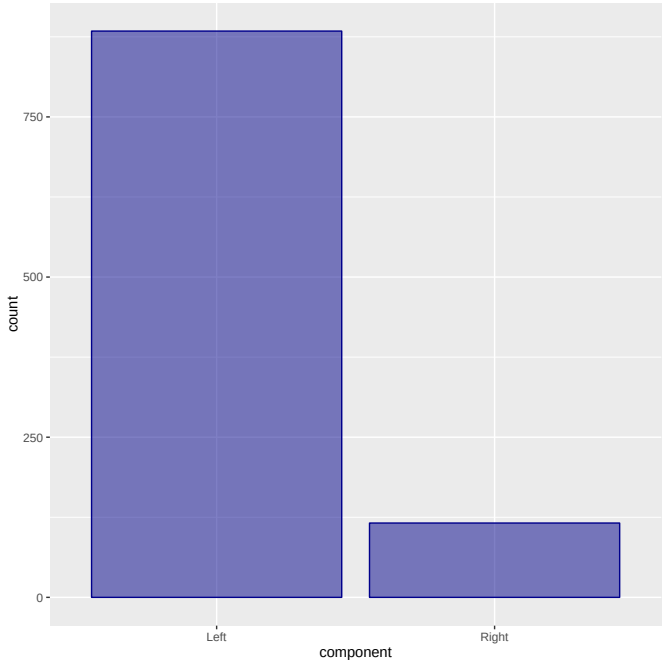


Figure 4.12: Count of the positions of the mixture component gathered from 1000 epochs of a Markov chain running over executions of the 'mixture 3 2' model conditioned on some observations. Most of the time is spent in the leftmost component of the mixture.

Figure 4.13 displays a trace of a Markov chain over the internal p parameter when using 'altPerturb 0.1' as the perturbation function.

4.6 Encoding Structural Independence

Before concluding this chapter, it is worth noting a small but interesting technique for encoding structural independence information in a probabilistic program using the *free applicative functor* [Capriotti and Kaposi, 2014]. We don't take this idea far here, but simply want to note it exists, and how it can be used

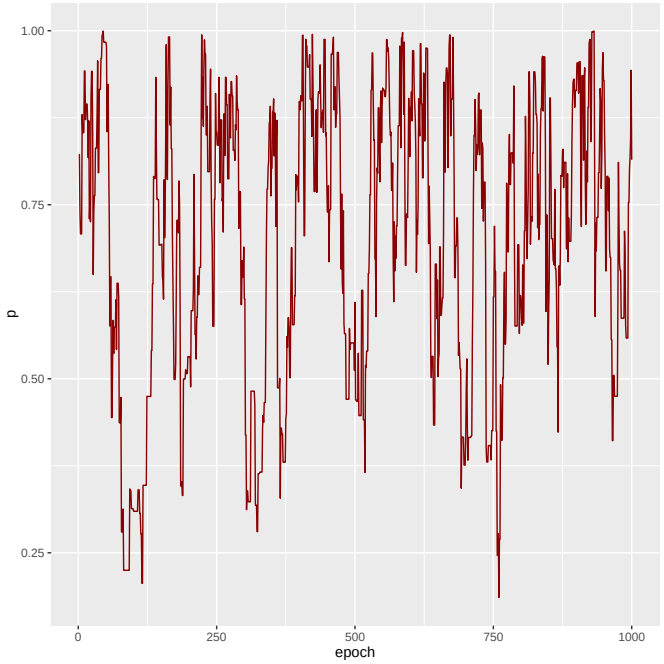


Figure 4.13: Positions of the mixing parameter p gathered from 1000 epochs of a Markov chain running over executions of the 'mixture 3 2' model conditioned on some observations, using an alternate perturbation function. The chain moves as expected according to the perturbation function used (proposing small, Gaussian-distributed steps about its present location).

in conjunction with the kind of 'Free'-embedded language we've been discussing thus far.

Categorically, the free applicative is defined in similar fashion to the free monad — it is left-adjoint to a forgetful functor that simply drops the applicative structure around a functor. In code, it is defined as follows:

```
data Ap f a where
  Pure :: a -> Ap f a
  Ap   :: f a -> Ap f (a -> b) -> Ap f b
```

Recall from Chapter 3 that applicative expressions encode independence in probabilistic models. The 'Ap' type is just a structure-preserving analogue of this, so to encode independence structurally we can use it to wrap up a probabilistic base functor, and then include that wrapped structure in our syntax tree.

Given our simple beta/Bernoulli/Gaussian base functor 'Prob', the following types allow us to use the free applicative to encode independence in our programs:

```
newtype Model a = Model (Sum Prob (Ap (Free Model)) a)
  deriving Functor

type Program = Free Model
```

Here 'Sum' is a *coproduct of functors*. A *coproduct* of objects X and Y in some category is an object $X + Y$ such that there exist morphisms $i : X \to X + Y$ and $j : Y \to X + Y$ satisfying the universal property that, for any Z and morphisms $k : X \to Z$ and $l : Y \to Z$, there exists a unique morphism f such that $k = f \circ i$ and $l = f \circ j$. A coproduct *of functors* is a coproduct in the category of functors between some categories C and D. In terms of Haskell code, it simply represents a sum type where each branch holds a value of one of the provided functors, parameterized with some type parameter common to both branches. The following code defines our typical language terms; note the 'InL' and 'InR' constructors for each branch of our aforementioned sum type:

```
bernoulli :: Double -> Program Bool
bernoulli p = liftF (ProgramF (InL (BernoulliF p id)))

beta :: Double -> Double -> Program Double
beta a b = liftF (ProgramF (InL (BetaF a b id)))

gamma :: Double -> Double -> Program Double
```

```
gamma a b = liftF (ProgramF (InL (GammaF a b id)))

gaussian :: Double -> Double -> Program Double
gaussian m s = liftF (ProgramF (InL (GaussianF m s id)))
```

Note the recursive structure of 'Program' – we have a base functor wrapped up in 'Free' as per usual, but the base functor is slightly more complicated than before. At each recursive point of the model, we can either have a primitive probabilistic instruction or a collection of conditionally independent expressions. These latter expressions could themselves either be primitive probabilistic instructions, or yet another collection of conditionally independent expressions, etc.

To encode independence structurally we can use the following combinators:

```
replicateA :: Applicative f => Int -> f a -> f [a]
replicateA n = sequenceA . replicate n

product :: Ap (Free Model) a -> Program a
product term = liftF (ProgramF (InR term))

iid :: Int -> Program a -> Program [a]
iid n term = product (replicateA n (liftAp term))
```

Here 'liftAp' is analogous to the free monad's 'liftF', and 'sequenceA' is a function that simply sequences together all applicative effects in a collection. In particular, the 'iid' function denotes a *plate* of n (conditionally) independent and identically distributed variables in the traditional probabilistic graphical model sense.

For illustration, the following program illustrates a model that combines functorial, applicative, and monadic structure:

```
nestedPlates :: Int -> Int -> Double -> Double -> Program [[Bool]]
nestedPlates m n a b = do
```

```
c <- fmap (+ 1) (beta a b)
d <- fmap (+ 2) (beta a b)
iid m (do
  p <- beta c d
  iid n (bernoulli p))
```

The two nested plates of m and n variables each in the above program can be statically identified as such during interpretation, and their contents are amenable to some static analysis that we're unable to achieve via monads [Capriotti and Kaposi, 2014]. Here we simply note that this technique allows us to denote structural conditional independence information in a model explicitly, but leave further investigation out of scope.

4.7 Summary and Comparison to Other Work

4.7.1 Free Monad Encoding

The free monad encoding technique is well-known in functional programming circles for deeply-embedding domain specific languages, but to the best of this author's knowledge it had not been applied in a probabilistic programming context until recently.[7] Ścibior et al. [2015] used a very similar technique to embed a probabilistic programming language in Haskell — their approach seems more akin to an *operational* monad, to which a free monad is known to be 'roughly' isomorphic [Apfelmus, 2010]. That approach differs in that it does not use a base functor of abstract probability distributions to parameterize 'the' free monad;

[7] It seems that at least three free monad-like implementations were developed independently at around the same time in mid-2015: the implementation presented here, that of Ścibior et al. [2015], and also an implementation posted at `https://goo.gl/RpGxfe` that has not seen any development since.

instead the foundational probability distributions are defined via top-level functions that return a value at a specific type corresponding to a probability distribution, similar to the Giry monad of Chapter 3 or the sampling monad presented near the start of this chapter. Zinkov and Shan [2016] described a similar abstract language based on a sort of manually-assembled free or operational monad structure, which is structurally similar to that in the present work, but was (clearly intentionally) not designed with pleasant user-facing syntax in mind.

The free encoding presented here seems to have some advantages around simplicity of implementation and re-use. The probabilistic instruction set encoded as a base functor can be re-used to parameterize any number of higher-kinded recursive types, and indeed we use this feature to implement the comonadic MCMC algorithm in Section 4.5.3. The base functor can similarly be re-used to encode different structure present in a program, for example via the free applicative in order to encode independence. The operational-style monad of Ścibior et al. [2015] seems to be more of a monolith in comparison. The pros and cons of either implementation seem to reduce to 'free' vs 'operational' pedantry, and we won't consider it any further here. We note that Ścibior et al. [2015] found that the operational encoding was insufficient to implement a single-site Metropolis-Hastings algorithm, which can be implemented via transformation to a cofree encoding (see Section 4.5.4).

The free monad encoding that we've used here is known to be asymptotically inefficient – having quadratic complexity in the number of monadic binds – compared to an alternate *Church encoding* [Voigtländer, 2008]. However, it's worth noting that this author has experimented with several Church-encoded probabilistic programming languages not discussed in this thesis and has actually observed *worse* performance from them. The naïve encoding tends to be reasonably quick, and is also much easier to use than the Church encoding.

The free monad must be weighed against other means of deeply-embedding a language in Haskell. The free monad technique is particularly lightweight and

can be implemented with a minimum of code; a significantly more heavyweight deep embedding was originally found in Hakaru [Hakaru, 2014], which has since moved to a standalone language. Such a deep embedding is able to capture more structure than can be done via the free monad, if one is willing to discard features like function application, type system, variable binding, etc. from the host language. But the implementation cost is higher; since probabilistic models are monadic anyway, the free monad approach works particularly well here for implementing minimalist embedded probabilistic programming languages.

A deeper embedding than can be provided via a free monad can be useful when one needs to be able to examine additional structure in expressions. Function application is an important one in particular; to illustrate, consider the following model:

```
shiftedBeta :: Double -> Double -> Program Double
shiftedBeta a b = fmap (+ 1) (beta a b)
```

This is a beta distribution whose support has been shifted from $(0, 1)$ to $(1, 2)$. Sampling from this model will work as expected, but a naïve interpreter that attempts to calculate a beta log-likelihood at a point by matching on the primitive instruction 'Beta' will be foiled. For instance, if one merely looks up the beta log-density and attempts to evaluate it at a point in the shifted support such as 1.5, the result will be incorrect. The problem is that the application of 'fmap (+ 1)' to the language term cannot be observed in this embedding, since it comes entirely from the host language.

If one does not wish to live with these restrictions, other useful techniques for deeply-embedding languages in Haskell include higher-order abstract syntax (HOAS) [Pfenning and Elliott, 1988], parametric HOAS (PHOAS) [Chlipala, 2008], and abstract syntax graph encodings [Oliveira and Löh, 2013]. Mainland and Morrisett [2010] describe a method for implementing observable function application in a deeply-embedded language, in particular. Ramsey and Pfeffer

[2002] developed a stochastic lambda calculus for denoting probabilistic programs. They described how expressions in that language can be evaluated via interpretation to a probability monad (such as the sampling or Giry monad), so it follows that expressions of the stochastic lambda calculus can trivially be evaluated to a free monad encoding of a probability monad in turn.

An important point about the free monad technique is that it is primarily useful for *embedding a language in a statically-typed, purely functional host language like Haskell*. It stands to reason that the free monad technique should transfer well to languages like Agda or Idris, and to some degree in an ML-flavoured language like OCaml, but is best avoided as an implementation technique for a standalone language, or for languages without a strong type system or those lacking support for functional programming idioms.

An extended lambda calculus is also used to implement Church [Goodman et al., 2008] (which is at its core an extension of Scheme) and some of its relatives. More generally, most probabilistic programming languages, embedded or no, tend to use as a basis a collection of abstract probability distributions that can be associated with some particular semantics or collection of semantics — many popular probabilistic programming languages such as Anglican [Wood et al., 2014], Venture [Mansinghka et al., 2014], BLOG [Li and Russell, 2013], FACTORIE [McCallum et al., 2009], PyMC [Salvatier et al., 2016], and Edward [Tran et al., 2016] use this pattern.

4.7.2 Inference

The rejection, importance, and Metropolis-Hastings algorithms described in Section 4.4.2 are implemented in the same spirit as those described by Ścibior et al. [2015] and Zinkov and Shan [2016] in that each is encoded as a probabilistic program. When it can be done, it tends to be very easy to implement these al-

gorithms in this fashion. However it cannot always be done, e.g. the case of single-site Metropolis.

So-called single-site Metropolis methods have traditionally been implemented in languages like Church and probabilistic-js according to the lightweight transformational compilation technique developed in Wingate et al. [2011]. The comonadic implementation mirrors this technique; it essentially replaces the 'database of randomness' described in that implementation by directly annotating parameters in the AST. The direct lightweight transformational compilation technique (or similar techniques that rely on dynamic typing, e.g. Bertschinger [2012]) requires some additional legwork to get right in a purely-functional language like Haskell. Earlier versions of Hakaru used it, but the implementation appeared significantly more complex than does the cofree-encoded variant.

The comonadic inference implementation is representationally elegant, in that we use 'Free' to encode models and 'Cofree' to perform inference on them given a single underlying base functor. Similarly, the cofree encoding allows us to perform 'interactive' inference, in that we can step through an execution trace of a probabilistic program and examine, perturb, evaluate it however we might want. Similar interactive capabilities are also available in Venture [Mansinghka et al., 2014], which lets users interact with execution traces directly. But the ability to represent and manipulate execution traces as first-class objects in a minimal embedded language is noteworthy, and a useful feature.

Kiselyov [2016] noted that the lightweight transformational compilation technique of Wingate et al. [2011] can encounter problems in some cases, observing that (intuitively) semantic-preserving transformations can seriously change a program's behaviour. The author points out an example in which there is a collision when looking up names following a conditional branch of the program. It is worth noting that by storing a node's history on-site in the cofree encoding, we completely avoid this problem since collisions are ruled out structurally. Kiselyov [2016] also proposed a structural solution to the name collision prob-

lem, but not one that involved a cofree encoding.

As for performance, the comonadic inference as implemented here is somewhat slow by any metric. It remains to be seen how performance can be improved, but we assert this out of scope for this thesis. Storing the history of each node on-site is likely needlessly costly and results in some unnecessary allocation; on a semantic level, it also mixes up the notions of state and identity (why, after all, should a single execution trace know anything about traces that preceded it?). The implementation can likely be improved by accumulating histories in another data structure. There is likely some low-hanging fruit around strictness and PRNG management as well.

It is also worth mentioning that there has been a flurry of recent work around the use of particle MCMC algorithms [Andrieu et al., 2010] and variants in probabilistic programming. We don't deal with these in this thesis, but refer in particular to work by Wood et al. [2014], Goodman and Stuhlmüller [2014], Ścibior et al. [2015], and Ścibior et al. [2017] as noteworthy research in the area.

4.8 Conclusion

In this chapter we used a free monad encoding to deeply-embed a probabilistic programming language in Haskell. Unlike the shallowly-embedded language based on the Giry monad from Chapter 3 or the sampling function-based language based on the sampling monad, the free monad allows us to write structured probabilistic programs that are amenable to a greater degree of interpretation.

We demonstrated that the semantics of the Giry and sampling monads could be grafted onto the free monad-encoded language via simple monadic interpreters, and that we were able to transform a free monad-encoded program into a cofree comonad-encoded execution trace. We used the same limited and reusable prob-

abilistic instruction set to parameterize each, delegating control flow to the corresponding higher-kinded recursive types, and demonstrated how we can do MCMC over these execution traces.

The next chapter changes pace, moving away from deep embeddings and onto a shallowly-embedded language useful for describing strategies for performing MCMC.

Chapter 5

Declarative Markov Chains

> The secret of success in battle lies often not so much in the use of one's own strength but in the exploitation of the other side's weaknesses.
>
> John Christopher

5.1 Abstract and Contributions

This chapter develops a shallowly-embedded language for dealing with another important component in applied Bayesian statistics: the *Markov transition operators* used in Markov Chain Monte Carlo. As in previous chapters, this language is based on the monadic structure of these operators; in particular, they form a *state monad* over an annotated parameter space.

After some motivating discussion, we briefly review the structure of Markov transitions and describe how they can be represented using the state monad.

We demonstrate that transition operators can be combined in a sound manner, using a simple monadic language that requires a minimum of syntax. We review several popular and efficient 'primitive' transition operators for MCMC: those based on the reliable Metropolis-Hastings and slice sampling algorithms, as well as gradient-exploiting transitions based on the Hamiltonian Monte Carlo (HMC), Metropolis-adjusted Langevin diffusion (MALA), and No U-Turn Sampler (NUTS) algorithms. Section 5.7 presents some simulations, demonstrating that compound transition operators can be more effective than primitive transitions on simple problems. We close the chapter by noting some simple extensions to the framework.

The primary contributions of this chapter are:

- A novel technique for **building custom transition operators for use in Markov Chain Monte Carlo (MCMC)**. Markov transition operators can be denoted by a particular instance of the *state monad* such that familiar monadic combinators can be used to build composite transition operators from a set of base, 'known-good' primitives.
- **General-purpose Haskell implementations** of the MH, slice sampling, HMC, MALA, and NUTS algorithms.

5.2 Motivation

The motivation for this chapter is perhaps best led off with an anecdote from the author, from a talk attended on the topic of measuring the efficacy of various MCMC algorithms on an astrostatistics problem. The presenters had found that the eminently-reliable Metropolis-Hastings and slice sampling algorithms had been the most effective samplers on this problem, and couldn't decide which algorithm should be preferred. This author asked the question "have you thought

about interleaving a Metropolis transition with a slice sampling transition?", which was met with laughter from the presenters over the perceived difficulty of the implementation.

Not so. The aim of this chapter is to demonstrate that a system for 'interleaving' transition operators in this fashion can be implemented as a borderline-trivial, shallowly-embedded monadic language.

First, some brief background on MCMC. MCMC is a family of algorithms used for approximating integrals. One is only ever interested in doing it at all because a particular integral — typically an expectation over a posterior density function — is difficult or impossible to evaluate analytically. MCMC is fundamentally about exploiting the properties of *Markov chains* — stochastic processes which wander over the parameter space of a problem and recover a sample of points that can be used to approximate an integral. The reason Markov chains are used in lieu of a suitably dense grid over some appropriate region of parameter space is that they — unlike grids — scale well to higher dimensions. One constructs a Markov chain such that its invariant or stationary distribution P^* *is* the distribution being integrated over, ensuring that in the limiting case the chain visits regions of the parameter space in proportion to their probability. This offloads to probability theory the problem of selecting an appropriate grid of points for doing approximate integration.

Markov chains are constructed by *transition operators* that obey the Markov property: that the probability of transitioning to the 'next' location in parameter space — conditional on the history of the chain — depends only on the current location. In MCMC we're also interested in operators that satisfy the *reversibility* property — that is, that the probability of being in state q_0 and transitioning from state q_0 to q_1 is the same as the probability of being in state q_1 and transitioning from state q_1 to q_0. A chain is characterized by a single transition operator T that drives it from state to state, and for MCMC we want the *stationary* distribution P^* of the chain to be the distribution we're trying to approximate an integral

over.

In practice, what matters is how fast the Markov chain can visit a sufficient number of representative points of parameter space, as a practitioner just wants to evaluate an integral and get on with his or her life. For any given space, the success of any chain depends entirely on the operator used (issues of starting location and burn-in aside).

One of the major cottage industries in Bayesian research is inventing new transition operators to drive Markov chains used for MCMC. This has historically been a fruitful endeavour, but it could potentially be aided by a practical way to make existing transitions work *together*. Note that this is easy to do in theory: consider transition operators $T_1, T_2, \ldots, T_n$, each satisfying the Markov and reversibility properties and each having a common stationary distribution P^*. Then composing these operators together by concatenation produces another operator $T_{1\ldots n} = T_1 T_2 \ldots T_n$ that is also reversible, obeys the Markov property, and has stationary distribution P^*. This concatenation can be defined as simply performing one transition after the other; in the example above, a single $T_{1\ldots n}$ transition is accomplished by transitioning the state using T_1 and then immediately transitioning the resulting state via T_2, continuing up to T_n. A Markov chain can then be defined via the compound transition $T_{1\ldots n}$.

Moreover, this is not the only way that a Markov transition operator can be defined in terms of others. Take the same Markov operators $T_1, T_2, \ldots, T_n$, and define the probability distribution P_T over them such that transition T_k is associated with probability p_{T_k}. Then we can define the transition operator $T^*_{1\ldots n}$ as the operator that transitions a state according to P_T. Again, this preserves all the properties we're interested in.

Note that by construction these methods of composition can be combined, and indeed form a recursive grammar defined abstractly by the following BNF:

```
transition ::= literal <primitive>
             | concatT transition transition
             | sampleT transition transition
```

It elegantly captures the properties we're trying to express: a Markov transition with stationary distribution P is either a literal (over some primitive transition operator definition), a concatenation of two transitions, or a distribution over transitions. All expressions in this language preserve both the Markov property and the stationary distribution, and can be used to drive a Markov chain. As a contrived example, we could define a moderately complex transition operator as follows: with probability p_0 sample a transition operator from the distribution P_{T_0} (defined abstractly over arbitrary other transitions) and with probability p_1 choose the operator defined by concatenating the operators $T_1 T_2$.

The result is a simple language for composing transition operators, and it has an immediate potential benefit: it allows the construction of transition operators that balance exploratory sampling power with computational expense. For example, we could construct a transition operator that with probability 0.9 performs a computationally-inexpensive spherical Metropolis transition, and with probability 0.1 performs a comparatively-demanding gradient-based transition, such as that used by Hamiltonian Monte Carlo [Neal, 2011]. The resulting transition would occasionally make gradient-based jumps, but spend most of its time making cheap moves based on a simple Gaussian proposal distribution. The desirable properties of the resulting operator (stationary distribution-preserving, reversibility, Markov) are preserved.

The motivation for this language is fruitfully compared with a similar problem found in parallel computing research. A perennial dream in parallel computation is the development of a 'sufficiently smart compiler' that could automatically parallelize algorithms across arbitrary compute units. But implementing a sufficiently smart compiler is a notoriously difficult problem as achieving perfor-

mance gains from parallelism can be a thorny matter in practice: aside from the theoretical limitations on parallel computation governed by Amdahl's Law [Amdahl, 1967], *runtime* or *environment* play a significant role in determining parallel speedups. Parallelism is typically accomplished by invoking numerous threads of control to execute smaller individual tasks independently, and coordinating these threads requires some overhead. The coordination problem, however, is usually dwarfed by environmental factors; memory access and transfer speeds in shared-memory environments, network latency in distributed systems, etc. are major bottlenecks to achieving parallel speedups in practice. Achieving speedups from parallelism is thus often a matter of trial and error — highly dependent on the algorithm being parallelized, and often subject to hard-to-estimate runtime penalties [Kirk and Hwu, 2010, Marlow, 2013].

Inference in Bayesian statistics is afflicted by similar difficulties. Samplers that work particularly well for some models may work poorly on others, and the primary time that this problem may be observed is during sampling itself. A 'sufficiently smart' inference system — like the sufficiently smart parallelizing compiler — would automatically tailor efficient inference algorithms to any model.

As an intermediate step towards a sufficiently smart parallelizing compiler, the Haskell community has advocated two main approaches for managing parallel computation. The first is the use of a monad for deterministic parallelism, in which lower-level details of parallelizing a computation are handled by immutable primitives 'under the hood' [Marlow et al., 2011, Kuper et al., 2014]. The second is the use of *evaluation strategies*, whereby a user of the language annotates how a function should be evaluated by means of certain strategy combinators [Trinder et al., 1998]. The strategy approach has the advantage of being composable and minimally invasive on the code being parallelized; primitive strategies can be combined together to form more elaborate strategies, without ever changing the sequential semantics of the program being evaluated. The strategy system allows a programmer to distinguish *what* should be evaluated

from *how* to evaluate it.

In the absence of a sufficiently smart inference system, an embedded language for building custom Markov transition operators could be useful for much the same reason. Customized transition operators that declare 'how something should be sampled' can be built from familiar sampling primitives, while also being minimally intrusive on the structure of the model under consideration.

The following sections illustrate an embedding for the simple language described above. As the BNF suggests, this language is very simple — we can get a lot of mileage out of the two abstract terms 'concatT' and 'sampleT' alone. But what's more is that unlike the free monad encoding developed in Chapter 4, we don't require a deep embedding for this language since we don't need to examine the structure of its expressions — we can get away without defining the above grammar at all, instead exploiting existing functionality from our host language to produce a shallow, monadic embedding.

5.3 The Structure of Markov Transitions

5.3.1 Markov Chains

It is worth saying something about the mathematical specification of Markov transitions. This section provides theoretical justification for the deterministic and random concatenation operations described in the previous section, mostly following Geyer [2005].

The distribution of a transition operator T between points in a general state space X is described by a *Markov kernel*. One takes a σ-algebra over X and considers the measurable space $(X, \mathcal{X})$ — a Markov kernel $P : X \times \mathcal{X} \to [0, 1]$ then has

the form:

$$P(x, A) = \Pr(X_{n+1} \in A \,|\, X_n = x) \tag{5.1}$$

such that $P(x, \cdot)$ is a probability measure for fixed $x \in X$ and $P(\cdot, A)$ is a measurable function for fixed $A \in \mathcal{X}$. The distribution of any given transition from x to some other point in the state space is thus established by the Markov kernel $P(x, \cdot)$, so that transitions $T_1, T_2, \ldots, T_n$ can be characterized probabilistically by their respective Markov kernels $P_1, P_2, \ldots, P_n$.

Markov kernels can be composed according to the rule:

$$(P_1 P_2)(x, A) = \int P_1(y, A) P_2(x, dy). \tag{5.2}$$

So here $P_1(\cdot, A)$ is a measurable function being integrated against the probability measure $P_2(x, \cdot)$. One can appeal to the conditional Fubini theorem [Applebaum, 2009] to establish that kernel composition is associative, viz. $(P_1 P_2) P_3 = P_1 (P_2 P_3)$.

A Markov chain has a *stationary distribution* P^* if, for any kernel P, composition with P^* results in P^*. That is, that:

$$P^* P = P^*.$$

Here the transition with distribution P is said to 'preserve the stationary distribution' of the chain. From here we can establish both the deterministic and random concatenation operations described in Section 5.2, each of which preserves the stationary distribution as required. For deterministic concatenation, i.e. the operation corresponding to 'concatT', if transitions $T_1, T_2, \ldots, T_n$ each preserve the stationary distribution of a chain, then:

$$\begin{aligned}
P^* P_1 P_2 \cdots P_n &= (P^* P_1) P_2 \cdots P_n && \text{(associativity)} \\
&= P^* P_2 \cdots P_n && \text{(stationary distribution)} \\
&= P^* P_n && \text{(induction)} \\
&= P^*.
\end{aligned}$$

For random concatenation, i.e. the operation corresponding to 'sampleT', we can show that transition operators are closed under convex combinations. That is: for coefficients $a_1, \ldots, a_n$ such that $a_i \geq 0$ and $\sum_i^n a_i = 1$, and transitions $T_1, \ldots, T_n$, we have that:

$$\begin{aligned}
P^* \left(\sum_i^n a_i P_i \right) &= \sum_i^n a_i P^* P_i && \text{(distributivity)} \\
&= \sum_i^n a_i P^* && \text{(stationary distribution)} \\
&= \left(\sum_i^n a_i \right) P^* && \text{(distributivity)} \\
&= P^*. && \text{(convexity)}
\end{aligned}$$

Since $\sum_i^n a_i P_i$ is simply the expected value of the probability distribution over some P_i's, we have that random concatenation also preserves the stationary distribution as required.

Note that we can take the approach from Chapter 3 and define kernels in terms of measurable *functions* instead of measurable *sets*. Take $(X, \mathcal{X})$ a measurable space and $f : X \to \mathbb{R}$ a measurable function. We can translate the kernel in Equation 5.1 into:

$$P(x, f) = \int f dP_x$$

for some appropriate probability measure P_x. Using a currying construction, we can characterize its type via:

$$P : X \to (X \to \mathbb{R}) \to \mathbb{R}.$$

Note that $(X \to \mathbb{R}) \to \mathbb{R}$ was the type we used to encode the Giry monad as a continuation. Generalizing the type of the continuation to another measurable space $(Y, \mathcal{Y})$, the type of the kernel can be rewritten as:

$$P : X \to \mathcal{P}(Y)$$

for $\mathcal{P}$ the Giry functor.

The class of Markov kernels $P : X \to \mathcal{P}(Y)$ can thus be identified as the morphisms of the so-called *Kleisli category* of the Giry monad, a fact noted by Panangaden [1999]. Here the objects are the same as in **Meas**, the identity morphisms are the Dirac morphisms $\delta : X \to \mathcal{P}(X)$, and composition is defined as in Equation 5.2.

We won't need to actually deal with any integration in this chapter. We simply use Markov kernels to denote 'canonical' probabilistic semantics for our transition operators in the same way that we used probability measures to denote semantics for sampling functions. The point is that transition operators can safely be composed in an associative way (using either deterministic or random concatenation) without invalidating any important probabilistic properties.

5.3.2 The State Monad

The primary tool we'll use to embed this language is the *state monad* — a data struture that can be used to denote stateful computation. Unlike probability monads — which are still probably considered somewhat exotic — the state monad is a run-of-the-mill monad familiar to just about any Haskell programmer. It can be defined as follows:

```
data State s a = State { run :: s -> (a, s) }

instance Functor (State s) where
  fmap f (State g) = State (\s ->
    let (a, s') = g s
    in  (f a, s'))

instance Applicative (State s) where
```

```
  pure  = return
  (<*>) = ap

instance Monad (State s) where
  return x      = State (\s -> (x, s))
  State g >>= f = State (\s ->
    let (a, s') = g s
    in  run (f a) s')
```

It is easy to verify it satisfies the functor and monad laws. For functor, note that:

```
fmap id (State g)
  = State (\s ->
      let (a, s') = g s
      in  (id a, s'))

  = State (\s ->
      let (a, s') = g s
      in  (a, s'))

  = State g
```

so that 'fmap id = id'. We can also show that:

```
fmap (f . h) (State g)
  = State (\s ->
      let (a, s') = g s
      in  ((f . h) a, s'))

  = State (\s ->
      let (a, s') = g s
      in  (f (h a), s'))
```

```
= fmap f (State (\s ->
    let (a, s') = g s
    in  (h a, s')))

= fmap f (fmap h (State g))
```

so that 'fmap (f . h) = fmap f . fmap h', as required. We omit a treatment of the monad laws here for brevity, but they can be verified in similar fashion (we will prove a virtually identical analogue of them below).

In practice we don't actually need to define the state monad machinery manually as it's a fundamental part of the Haskell ecosystem. We can just import 'Control.Monad.Trans.State.Strict' from the *transformers* library and have it at our disposal. It comes with a number of useful convenience functions for working with the underlying state.

The state monad and its importance can be illustrated simply by the following functions that use it to add up the contents of a list:

```
increase :: Int -> State Int ()
increase x = modify (+ x)

statefulSum :: [Int] -> Int
statefulSum xs = execState (for xs increase) 0
```

The function 'execState' executes a stateful action by running the monad. The 'increase' function — which modifies some existing state by adding an amount to it — is applied to each element of an input list using the 'for' combinator. 'execState' requires an initial state to work from, so we provide it with 0. The 'modify' function here is one that instances of the state monad get for free: it simply examines the state and modifies it according to the provided function.

The above example is more interesting than it looks. Consider the 'increase' function in particular: it modifies some state by adding a provided integer to it. But note that it doesn't 'know' what state it's going to receive — it will work for any provided state. Also, the type system restricts that the state must be of type 'Int', indicating that it can't be passed (for example) a list of integers representing the history of states received.

In other words, '**increase' describes a way to transition from an arbitrary point in some state space to another, and the value that it transitions to does not depend on the history of states that it has visited**. It is exactly a Markov transition operator. As mentioned in the previous section, 'increase' is an example of a so-called *Kleisli arrow* for a given monad — a function of type $a \to m\, b$ for monad m where it is evident from the type alone that the input is a pure value, independent of any history.

However, it's not yet a transition operator that's useful for MCMC — it has no stationary distribution, nor is it reversible. To get there, we can reuse the sampling function-based probability monad from Chapter 4 in order to make non-deterministic transitions, layering a state monad transformer on top of it. The 'StateT' monad transformer is defined as follows:

```
newtype StateT s m a = StateT { runStateT :: s -> m (a, s) }

instance (Functor m) => Functor (StateT s m) where
  fmap f (StateT g) = StateT (\s ->
    let (a, s') = g s
    in  (f a, s'))

instance (Functor m, Monad m) => Applicative (StateT s m) where
  pure  = return
  (<*>) = ap
```

```
instance (Functor m, Monad m) => Monad (StateT s m) where
  return a = StateT (\s -> return (a, s))
  m >>= k  = StateT (\s -> do
    (a, s') <- runStateT m s
    runStateT (k a) s')
```

It operates exactly like the state monad described previously, except it can be parameterized by another monad as well. Define the following type for transition operators, using the 'StateT' monad transformer over our existing probability monad:

```
type Transition s m a = StateT s (Prob m) a
```

Recall that the sampling monad from Chapter 4 uses a primitive state monad under-the-hood in order to handle the state of a PRNG. So here we've created a monad transformer stack with three 'layers', each providing different functionality. From the top down, we have:

- An explicit state-passing layer denoted by 'StateT s' for some state of type 's', allowing us to transition between points of some user-defined state space.
- A probability monad layer denoted by 'Prob m', allowing us to make random choices.
- A primitive state-passing layer denoted by the type 'm' that handles passing a PRNG.

Note that the transition operator monad – being a simple monad transformer stack — satisfies the functor and monad laws so long as each of its layers does. We will take as given that the primitive state monad at the bottom satisfies the

monad laws and simply verify below that the monad laws hold for the topmost state layer — the functor laws can be verified in the same manner we did for the plain state monad.

For left-identity, we have:

```
return x >>= f
  = StateT (\s -> do
      (a, s') <- runStateT (return x) s
      runStateT (f a) s')

  = StateT (\s -> do
      (a, s') <- runStateT (StateT (\t -> return (x, t))) s
      runStateT (f a) s')

  = StateT (\s -> do
      (a, s') <- return (x, s)
      runStateT (f a) s')

  = StateT (\s -> runStateT (f x) s)

  = StateT (runStateT (f x))

  = f x
```

And right-identity follows similarly:

```
m >>= return
  = StateT (\s -> do
      (a, s') <- runStateT m s
      runStateT (return a) s')
```

```
  = StateT (\s -> do
      (a, s') <- runStateT m s
      runStateT (StateT (\t -> return (a, t))) s')

  = StateT (\s -> do
      (a, s') <- runStateT m s
      return (a, s'))

  = StateT (\s -> runStateT m s)

  = StateT (runStateT m)

  = m
```

For associativity, note first that

```
(StateT f >>= g) >>= h
  = StateT (\s -> do
      (a, s') <- runStateT (StateT f) s
      runStateT (g a) s') >>= h

  = StateT (\s -> do
      (a, s') <- f s
      runStateT (g a) s') >>= h

  = StateT (\s -> do
      let i t = do
        (b, t') <- f t
        runStateT (g b) t'

      (a, s') <- i s
      runStateT (h a) s')
```

```
    = StateT (\s -> do
        (a, s') <- f s
        (b, s'') <- runStateT (g a) s'
        runStateT (h b) s'')
```

and then that

```
  StateT f >>= \x -> (g x >>= h)
    = StateT f >>= \x -> StateT (\s -> do
        (a, s') <- runStateT (g x) s
        runStateT (h a) s')

    = StateT (\s -> do
        (a, s') <- f s
        runStateT (\x -> StateT (\t -> do
          (b, t') <- runStateT (g x) t
          runStateT (h b) t') a) s')

    = StateT (\s -> do
        (a, s') <- f s
        runStateT (StateT (\t -> do
          (b, t') <- runStateT (g a) t
          runStateT (h b) t')) s')

    = StateT (\s -> do
        (a, s') <- f s
        (b, s'') <- runStateT (g a) s'
        runStateT (h b) s'')
```

so that evaluating in any order yields the same result, as required.

5.4 A Shallowly Embedded Language For Transitions

The 'Transition' type contains all the functionality required to characterize reversible Markov transition operators, as we'll see in Section 5.5. Now what we'd like to do is implement analogues for the 'concatT' and 'sampleT' terms defined in Section 5.2. These will provide the ability to compose transition operators together, either by straight concatenation or by sampling them from some distribution.

The structure of the state monad used to implement these transition operators is such that existing functionality in Haskell makes this embedded language embarrassingly easy to implement.

To start, deterministic concatenation of operators is provided by the monadic '>>' operator that all instances of Monad get automatically. It is defined as follows:

```
(>>) :: Monad m => m a -> m b -> m b
m >> k = m >>= const k
```

Since we will only be concerned about chains over a single parameter space, we can specialize the types to define 'concatT' as an alias, as below:

```
concatT
  :: Monad m
  => Transition s m a
  -> Transition s m a
  -> Transition s m a
concatT t0 t1 = t0 >> t1
```

We can provide another convenience function for concatenating lists of transition operators by simply folding the list together with >>:

```
concatAll
  :: Monad m
  => [Transition s m a]
  -> Transition s m a
concatAll = foldl1 (>>)
```

Defining the 'sampleT' analogue is not much more difficult. Here we make use of the probability monad encoded in the 'Transition' type to make a random choice, using the 'lift' function that allows us to work with monad transformer stacks:

```
sampleT
  :: PrimMonad m
  => Transition s m a
  -> Transition s m a
  -> Transition s m a
sampleT t0 t1 = do
  heads <- lift (bernoulli 0.5)
  if   heads
  then t0
  else t1
```

It's important to note that, for primitive transition operators T_0 and T_1, a transition operator $T_{0,1}$ built using 'sampleT' is not always one of T_0 or T_1. Rather, every time $T_{0,1}$ is used to transition a state it will randomly choose one of T_0 or T_1 to do the dirty work. A resulting Markov chain would be formally driven by $T_{0,1}$, but any sequence of 'raw' transitions used might look like

$$T_0T_0T_1T_0T_1T_1T_1T_0\ldots$$

We can create other convenience functions for composing transition operators probabilistically. 'oneOf' uses one element from the provided list of transition operators at random:

```
oneOf
  :: PrimMonad m
  => [Transition s m a]
  -> Transition s m a
oneOf ts = do
  j <- lift (uniformR (0, length ts - 1)
  ts !! j
```

Here the '!!' operator simply retrieves the value of a list at the provided index.

We can define 'firstWithProb' to use the first of two supplied transition operators with probability p. This is just a generalization of 'sampleT' with an additional argument for the success probability — note that 'sampleT' is 'firstWithProb 0.5':

```
firstWithProb
  :: PrimMonad m
  => Double
  -> Transition s m a
  -> Transition s m a
  -> Transition s m a
firstWithProb p t0 t1 = do
  heads <- lift (bernoulli p)
  if heads then t0 else t1
```

And finally we can define the function 'frequency' to choose a transition operator according to a supplied frequency distribution[1]:

```
frequency
  :: PrimMonad m
  => [(Int, Transition s m a)]
```

[1]This idea and code is almost verbatim from the Haskell library *quickcheck* for randomized property-based testing.

```
  -> Transition m a
frequency xs = lift (uniformR (1, tot)) >>= (`pick` xs) where
  tot = (sum . map fst) xs
  pick n ((k, v):vs)
    | n <= k = v
    | otherwise = pick (n - k) vs
```

The 'Transition' type combined with the above functions for working with it constitute another lightweight embedded DSL, and we can use it to create Markov chains that sample from target functions that are proportional to probability densities.

And that's it. The above combinators are *all that is needed* for an embedded language that allows us to build complex transition operators from a set of law-abiding primitives.

The rest of this chapter illustrates this embedded language using a number of implementations of primitive transitions, and is available on Github as the *declarative* library. Implementations for the individual primitive transitions are also available in the *mighty-metropolis*, *speedy-slice*, *hasty-hamiltonian*, *lazy-langevin*, and *hnuts* libraries respectively. In the following sections we'll define our primitive transition operators at a high level; for low-level details of the implementations, please see the respective libraries.

5.5 Primitive Transition Operators

In the case of the 'Transition s m a' type, the type system enforces the Markov property by preventing us from 'cheating' and looking at the history of a chain. But here the type system can only help us so much: it can't enforce similar invariants about reversibility or stationarity. These properties are important if MCMC is to work as expected.

To be able to make some guarantees about the validity of the transitions expressible in the embedded language, we'd want to prohibit users from defining their own *primitive* transitions, and instead insist that they build their custom transitions from a fixed set of known primitives that we provide.

In the following sections we implement five popular and useful transition operators that constitute a set of 'known good' primitives. Each is independently reversible and stationary distribution-preserving, so we can guarantee that these properties will be preserved for compound transitions created from them. We'll want to use these to sample from a space with probability proportional to some *target function*. This is typically something proportional to the posterior density of a conditioned probabilistic model, but the presentation is kept general in this chapter.

5.5.1 A Concrete State Type

Up until now we've kept the type of our state space abstract, denoting it by 's' in the type 'Transition s m a'. To decrease type signature clutter we'll make this type concrete in the following implementations, instead just using a type 'Transition m a' for some concrete 's'.

We'll restrict ourselves to running Markov chains over homogeneously-typed spaces; that is, spaces that have the same type on each dimension. This includes $\mathbb{R}^n$ or even $\mathbb{Z}^n$, for example, but disallows spaces like $\mathbb{R} \times \mathbb{Z}$. Note that this can be done — we'd just have to transition across dimensions in blocks — but is unnecessary for the scope of this dissertation. This means that in all cases a point in parameter space can be represented by an n-dimensional vector. The simplest concrete type for our state space is thus 'Vector a' for some type 'a'. In practice, though, we'll also want to cache a few more things in the state, rather than just the parameter space position. We can do this without violating the Markov property, however.

To avoid repeating computation unnecessarily, we'll cache the value of the target function at the most recent point that we've visited in parameter space. We'll also include with the state a supplementary store of information needed by various primitive transitions that again only depend on the latest state of the chain — these are the typical 'tuning parameters' used by many MCMC algorithms.

More controversially, we'll also cache the target function itself. This is benign so long as we're disciplined when it comes to using it, and in fact, none of the primitives described in this section actually modify it — using it only as a read-only value. In some applications it can however be useful to change the state of the target function; an example of this is illustrated later, in Section 5.8.

The target function is encoded by the following type:

```
data Target a = Target {
    logObjective :: Vector a -> Double
  , gradient     :: Maybe (Vector a -> Vector a)
  }
```

Notice first that the target is expected to be a *log*-target from a point in parameter space to the reals. It is potentially accompanied by its gradient; this is not mandatory in general, but is required to use gradient-based transitions. The following helper functions are useful for creating targets:

```
createTargetWithoutGradient :: (Vector a -> Double) -> Target a
createTargetWithoutGradient f = Target f Nothing

createTargetWithGradient
  :: (Vector a -> Double)
  -> (Vector a -> Vector a)
  -> Target a
createTargetWithGradient f g = Target f (Just g)
```

The concrete state of our chains becomes the following product type, called 'Chain':

```
data Chain a = Chain {
    parameterSpacePosition :: Vector a
  , objectiveFunction      :: Target a
  , objectiveValue         :: Double
  , optionalStore          :: OptionalStore
  }
```

We'll also adapt the return type of the transition to be the most recently visited point of the chain, leading us to the following transition type

```
type Transition m a = Transition (Chain a) (Prob m) (Vector a)
```

With these types in place we can define the following 'mcmc' function that traces a Markov chain for n iterations according to a particular transition operator:

```
mcmc
  :: Monad m
  => Int
  -> Transition m a
  -> Chain a
  -> Gen (PrimState m)
  -> m [Vector a]
mcmc n t o = sample (replicateM n t `evalStateT` o)
```

To illustrate its use from a user's perspective, consider something proportional to a simple 1D standard Gaussian log-density as the target. The following code initializes a value of type 'Chain Double' at the point 0, from which we can run a Markov chain:

```
logTarget :: Double -> Double
logTarget x = negate (x ^ 2 / 2)

initialPosition :: Chain Double
initialPosition = Chain position target value empty where
  position = V.fromList [0.0]
  target   = createTargetWithoutGradient logTarget
  value    = logObjective target position
```

We could then run a chain using an abstract transition operator called 'trans' using the 'mcmc' function:

```
> prng <- create
> mcmc 1000 trans initialPosition prng
```

In the following sections we'll define some primitive transition operators that we could use to take the place of the 'trans' function. We will simply describe the primitive transitions here: for the corresponding code, see Appendix B.

Note that one effect of restricting the parameter space type to 'Vector a' is that we exclude the use of *ensemble* samplers as primitives. This is by necessity, as a transition operator for a single particle is incompatible with that of an ensemble. This rules out useful algorithms for MCMC, such as the famous affine-invariant ensemble sampler of Goodman and Weare [2010]. An existing Haskell implementation of this algorithm is available on Github as *flat-mcmc* [Tobin, 2012], but we're unable to make use of it here.

5.5.2 Metropolis-Hastings (MH)

Metropolis-Hastings [Metropolis et al., 1953, Hastings, 1970] remains the reliable workhorse of MCMC as it is easy to implement and computationally inexpen-

sive. But it does encounter various well-known drawbacks on some problems — namely those where variables are highly correlated, creating narrow, twisting regions of probability in parameter space that are difficult to move through unless customized local proposals are used.

As a primitive transition operator for the *declarative* framework, the MH transition represents a safe baseline that one may wish to use 'most of the time', possibly by composing it probabilistically with some more computationally expensive transition operator like HMC. It has a very standard implementation — the current position of the chain is perturbed according to a proposal distribution, and the proposed move is accepted or rejected according to the standard criterion.

The Metropolis-Hastings transition operator is represented in the sequel via the following monadic function:

```
metropolisHastings :: PrimMonad m => Maybe Double -> Transition m Double
```

5.5.3 Slice Sampling

Slice sampling [Neal, 2003] is a simple and powerful method for doing MCMC. It roughly works as an approximation to Gibbs sampling [Geman and Geman, 1984], making incremental local moves across one parameter at a time. However it is less demanding than Gibbs sampling in that full conditional distributions are not required. A slice sampling implementation iterates through each dimension of the parameter space and perturbs it, creating a final transition across all dimensions. The algorithm makes use of rejection sampling internally, in its 'inner loop'.

Slice sampling fits roughly the same role as the MH transition in the *declarative* library: it is a computationally inexpensive workhorse that can be occasionally

mixed with other, more expensive transitions.

The slice sampling transition operator is represented in the sequel via the following monadic function:

```
slice :: PrimMonad m => Double -> Transition m Double
```

5.5.4 Hamiltonian Monte Carlo (HMC)

Hamiltonian Monte Carlo (HMC) [Neal, 2011] or *Hybrid Monte Carlo* is a gradient-based method that seeks to avoid random walk behaviour while wandering through parameter space. It is conceptually simple to understand and implement but can be difficult to use in practice, as its performance is particularly sensitive to the values of two tuning parameters. With good settings for these parameters HMC is known to perform notably well [Girolami and Calderhead, 2011].

The HMC transition operator is represented in the sequel via the following monadic function:

```
hamiltonian :: PrimMonad m =>
  Maybe Double -> Maybe Int -> Transition m Double
```

5.5.5 Metropolis-adjusted Langevin Diffusion (MALA)

The Metropolis-adjusted Langevin Diffusion [Girolami and Calderhead, 2011] is a gradient-based MCMC algorithm in which the proposal process is a discretized Langevin diffusion with drift. It is easy to use, requiring only a single tuning parameter that is used to determine the magnitude of the proposal. It is also relatively simple to implement.

The MALA transition is more or less a drop-in replacement for HMC in the case that one only wants to specify a single tuning parameter. Girolami and Calderhead [2011] found comparable performance between the two algorithms, with HMC slightly outperforming MALA on their test cases.

The MALA transition operator is represented in the sequel via the following monadic function:

```
mala :: PrimMonad m => Maybe Double -> Transition m Double
```

5.5.6 No U-Turn Sampler (NUTS)

Hoffman and Gelman [2011] developed the No U-Turn Sampler (NUTS) as a way to automatically find acceptable values for the tuning parameters of HMC. It is an implementation of HMC that uses a number of heuristics to determine how these parameters are to be set. NUTS is famously used as the backend for Stan [Stan, 2013], and is also included in libraries like PyMC3 [Salvatier et al., 2016].

Hoffman and Gelman [2011] also specify some extensions to basic NUTS — such as the incorporation of *dual averaging* [Nesterov, 2009] — that make use of a burn-in period to help find good values for parameters. This is awkward to incorporate in our framework since transition operators must not make use of the history of the chain, and so we can't easily define transitions as taking place in a burn-in phase or not. This extension is doable — it would require adding a burn-in phase indicator to the 'Chain' type, but remains unimplemented here.

The NUTS transition is the most heavyweight of the primitive operators discussed here. It can perhaps be used sparingly as a supporting transition to more-frequently-used MH or slice sampling transitions when good settings for HMC's tuning parameters are difficult to come up with.

The NUTS transition operator is represented in the sequel via the following monadic function:

```
nuts :: PrimMonad m => Transition m Double
```

5.6 Composite Transition Operators

Writing composite transitions is very simple, following the simple combinators presented in Section 5.4. Consider a progressive Metropolis transition that deterministically iterates through several Metropolis transitions of increasing step size; we simply map a 'metropolisHastings' transition-creating function over a list of step sizes and concat the results:

```
mhProgressiveTransition :: PrimMonad m => Transition m Double
mhProgressiveTransition =
  let sizes = [0.1, 0.5, 1.0, 2.0, 2.5]
  in  concatAll (fmap (metropolisHastings . Just) sizes)
```

Similarly we can create a progressive slice sampler that deterministically slice samples with step sizes 0.5 and 1.0, and then randomly chooses another slice sampler with step size 4.0 or 10.0:

```
sliceTransition :: PrimMonad m => Transition m Double
sliceTransition = do
  slice 0.5
  slice 1.0
  oneOf [slice 4.0, slice 10.0]
```

We can create a relatively heavyweight transition operator that deterministically concatenates three transitions together: the first being a Metropolis transition

with probability 0.8 and a HMC transition with probability 0.2, and the remaining two a slice sampling transition and NUTS transition respectively:

```
customTransition :: PrimMonad m => Transition m Double
customTransition = do
  firstWithProb 0.8
    (metropolisHastings (Just 3.0))
    (hamiltonian (Just 0.05) (Just 20))
  slice 3.0
  nuts
```

And as a final example we can create a transition operator based on a distribution over transitions; a Metropolis transition is chosen with probability 1/2, a slice sampler with probability 2/5, and NUTS transition with probability 1/10:

```
randomTransition :: PrimMonad m => Transition m Double
randomTransition = frequency
  [ (5, metropolisHastings (Just 1.5))
  , (4, slice 1.0)
  , (1, nuts)
  ]
```

These composite transitions are also composable. Consider this contrived one, just for illustration:

```
probablyOverkill :: PrimMonad m => Transition m Double
probablyOverkill = oneOf [
    mhProgressiveTransition
  , sliceTransition
  , randomTransition
  , firstWithProb 0.6 customTransition (slice 0.1)
  ]
```

In the following section we'll demonstrate some of these transitions over a collection of test targets.

5.7 Simulations

5.7.1 Overview

Here we'll use a variety of low-dimensional popular test functions for optimization and sampling in order to test-drive some transition operators; they each have various interesting properties (for example correlated variables and disparate modes of probability) and can be conveniently visualized. These are useful for testing transitions because we're not concerned about their stationary distribution-preserving properties — those have already been established for the primitive transitions we're using, and we are assured that composite transitions remain well-behaved as well. By using a set of low-dimensional test functions we can easily visualize a run of a given chain and roughly judge 'how it has done', which is difficult to do in larger problems.

We'll also use the effective sample size metric from R's *coda* library [Plummer, 2015b] to roughly measure the efficiency of the various chains under different transitions. But it will be easy to see that this metric — like others — can't really tell the whole story when it comes to MCMC, which is why visualizing a chain's trace can be useful.

A rigorous study of the merits of various composite transition operators is beyond the scope of this thesis, which proposes only that a novel language for expressing such compound operators can be embedded in a suitable host. The reason for this is that the search space of the problem is large: one would want to develop a set of useful composite transitions and then characterize the classes of targets for which each of them might be useful. Since we're typically interested

in sampling over a posterior, it is then another significant step to characterize what kind of model and data correspond to an identified class of target.

Quantifying the tradeoff between computational cost and exploratory power for any given transition operator also requires serious preparation and effort; the implementations of the primitive transitions would each have to be carefully optimized, and careful attention would need to be applied to benchmarks to ensure that they really measured the performance of relevant parts of the transitions (rather than, say, the speed of memory allocation or access to the disk).

What is useful here is to merely demonstrate that a given compound transition *can* be useful on a given problem when compared with primitive transitions.

5.7.2 Target Densities

To test the various transition operators we'll use four simple yet somewhat pathological targets mostly gathered from a Wikipedia collection of functions commonly used to test optimization algorithms [Wikipedia, 2015].

The Rosenbrock density

The first is the two-dimensional *Rosenbrock density*, defined as follows:

$$f(x, y) = \exp\{-100(y - x^2)^2 - (1 - x)^2\}. \tag{5.3}$$

The Rosenbrock density is *anisotropic* or 'banana-shaped', containing long, narrow, twisting canyons of probability about a mode (see Figure 5.1). It was notably used by Goodman and Weare [2010] to test their AIEMCMC sampler, due to the strong correlation between its variables. We can wrap the log-Rosenbrock density up as a 'Target' as follows:

```
rosenbrock :: Target Double
rosenbrock = createTargetWithGradient lRosenbrock glRosenbrock where
  lRosenbrock :: Vector Double -> Double
  lRosenbrock xs =
    let [x, y] = V.toList xs
    in  (-1) * (5 * (y - x ^ 2) ^ 2 + 0.05 * (1 - x) ^ 2)

  glRosenbrock :: Vector Double -> Vector Double
  glRosenbrock xs =
    let [x, y] = V.toList xs
        dx = 20 * x * (y - x ^ 2) + 0.1 * (1 - x)
        dy = -10 * (y - x ^ 2)
    in  V.fromList [dx, dy]
```

Note that our log-target only needs to be *proportional* to the actual log-density of interest, so we've elided a factor of 20 in the code when compared to Equation 5.3.

The Himmelblau density

The second function is the *Himmelblau density*, defined by

$$f(x, y) = \exp\{-((x^2 + y - 11)^2 + (x + y^2 - 7)^2)\}.$$

The Himmelblau density contains four distinct regions of probability, two of which are closer together than the others (see Figure 5.2). It can be encoded as follows:

```
himmelblau :: Target Double
himmelblau = createTargetWithGradient lHimmelblau glHimmelblau where
  lHimmelblau :: Vector Double -> Double
  lHimmelblau xs =
```

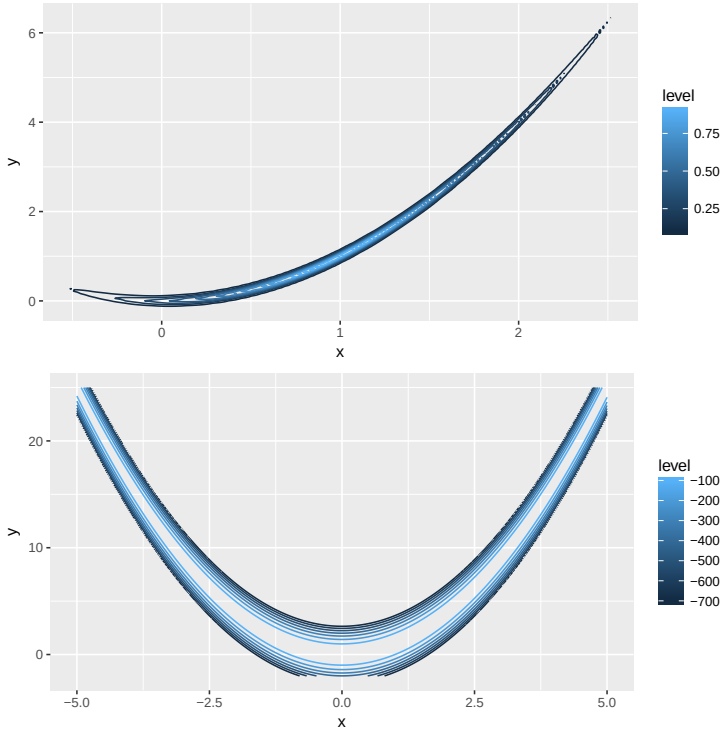


Figure 5.1: Contours of the Rosenbrock and log-Rosenbrock densities. The density is anisotropic — most of the probability is concentrated along a skinny, banana-shaped canyon.

```
  let [x, y] = V.toList xs
  in  (-1) * ((x * x + y - 11) ^ 2 + (x + y * y - 7) ^ 2)

glHimmelblau :: Vector Double -> Vector Double
glHimmelblau xs =
  let [x, y] = V.toList xs
      quadFactor0 = x * x + y - 11
      quadFactor1 = x + y * y - 7
      dx = (-2) * (2 * quadFactor0 * x + quadFactor1)
      dy = (-2) * (quadFactor0 + 2 * quadFactor1 * y)
  in  V.fromList [dx, dy]
```

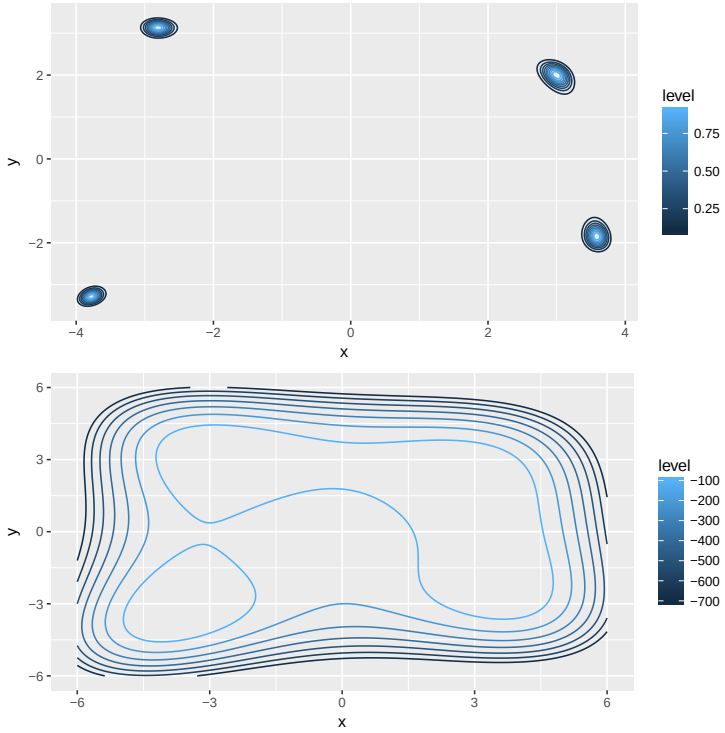


Figure 5.2: Contours of the Himmelblau and log-Himmelblau densities. The density has four distinct modes of probability that are well-separated from each other.

The Beale density

The third is the *Beale density*, defined by

$$f(x, y) = \exp\left\{-((1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2)\right\}.$$

The Beale density is particularly tricky; it contains two canyons of probability, one which contains significantly more probability than the other (see Figure 5.3). We can write it in Haskell as:

```
beale = createTargetWithGradient lBeale glBeale where
  lBeale :: Vector Double -> Double
  lBeale xs
```

```
    | and [x >= -4.5, x <= 4.5, y >= -4.5, y <= 4.5]
        = negate ((1.5   - x + x * y) ^ 2
                  + (2.25  - x + x * y ^ 2) ^ 2
                  + (2.625 - x + x * y ^ 3) ^ 2)
    | otherwise = - (1 / 0)
  where
    [x, y] = V.toList xs

glBeale :: Vector Double -> Vector Double
glBeale xs =
  let [x, y] = V.toList xs
      dx = negate (2 * (1.5 - x + x * y) * ((-1) + y)
          + 2.25  * 2 * (2.25 - x + x * y ^ 2) * ((-1) + y ^ 2)
          + 2.625 * 2 * (2.2625 - x + x * y ^ 3) * ((-1) + y ^ 3))
      dy = negate (2 * (1.5 - x + x * y) * x
                  + 2 * (2.25 - x + x * y ^ 2) * 2 * x * y
                  + 2 * (2.625 - x + x * y ^ 3) * 3 * x * y ^ 2)
  in  V.fromList [dx, dy]
```

The BNN density

Finally there is the *BNN density*, defined as

$$f(x, y) = \exp\left\{-\frac{1}{2}(x^2y^2 + x^2 + y^2 - 8x - 8y)\right\}.$$

It contains two large mounds of probability over a correlated parameter space (see Figure 5.4), and has the following encoding:

```
bnn :: Target Double
bnn = createTargetWithGradient lBnn glBnn where
  lBnn :: Vector Double -> Double
```

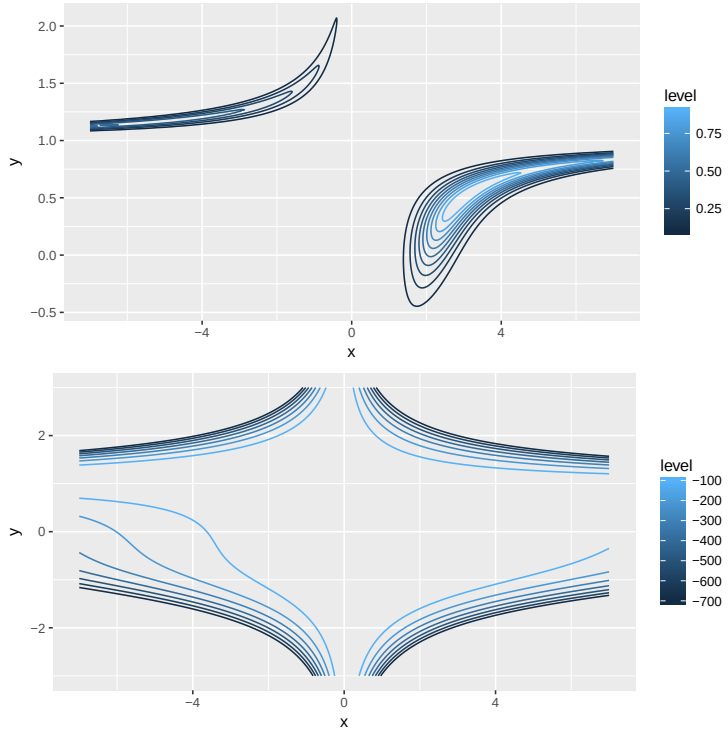


Figure 5.3: Contours of the Beale and log-Beale densities. The density has two distinct modes of probability, each bending sharply along two asymptotes.

```
lBnn xs =
  let [x, y] = V.toList xs
  in  -0.5 * (x ^ 2 * y ^ 2 + x ^ 2 + y ^ 2 - 8 * x - 8 * y)

glBnn :: Vector Double -> Vector Double
glBnn xs =
  let [x, y] = V.toList xs
      dx = -0.5 * (2 * x * y * y + 2 * x - 8)
      dy = -0.5 * (2 * x * x * y + 2 * y - 8)
  in  V.fromList [dx, dy]
```

The BNN density has the most interesting namesake: is named as such because

the author found it somewhere years ago, defined some Haskell code to work with it under the nondescriptive name 'bnn', and years later forgot both where he had found it and what it was supposed to represent. If you happen to know who to credit for it, please don't hesitate to share.

It's worth noting that while we've manually calculated the gradients for the targets in the examples above, we could have equally used an automatic differentiation library (for example Kmett [2010]) to calculate these for us.

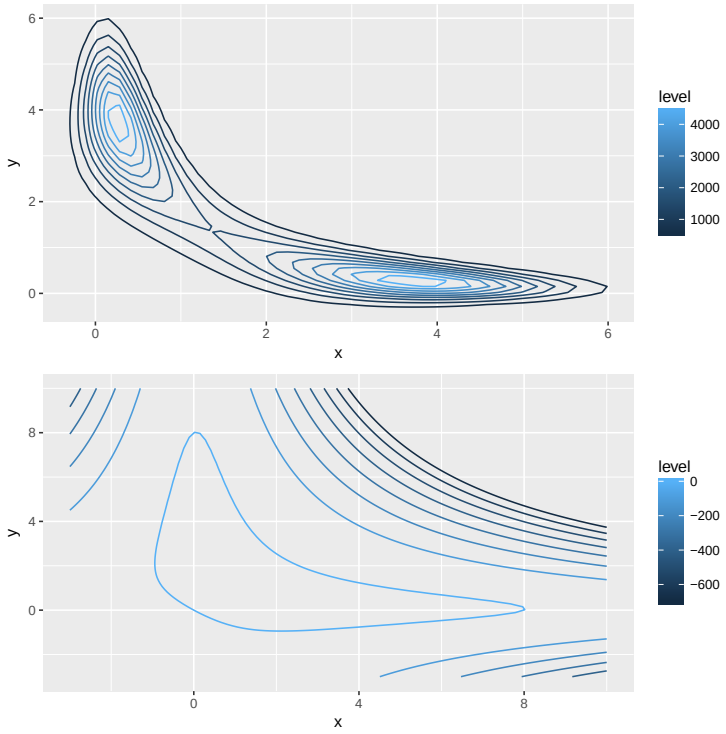


Figure 5.4: Contours of the BNN and log-BNN densities. The density has two large modes of probability over a correlated parameter space.

5.7.3 Configuration

We'll use six transition operators for running chains on these test targets, assembled more or less at random. The first are three primitive operators corresponding to basic MH, HMC, and NUTS transitions where the MH and HMC step sizes are set to 1 and 0.05 respectively, and the HMC leapfrog-steps parameter is set to 10. The second three are composite transitions corresponding to the 'mhProgressiveTransition', 'customTransition', and 'randomTransition' operators defined in Section 5.6. We can use the following Haskell function to drive the chains — each of which is capped at approximately 10000 total primitive transitions in order to keep the total computation constant. Also note that same random seed is used for each chain, and each is also initialized at the point $(1, 1)$.

```
tracer
  :: PrimMonad m
  => [(String, Int, Chain Double, Transition m Double)]
tracer =
    [ (sl ++ "-" ++ cl, n, chain, trans)
    | (cl, chain) <- chains
    , (sl, n, trans) <- transitions
    ]
  where
    transitions = [
        ("mh" , 10000, mhBaseTransition)
      , ("hmc", 10000, hmcBaseTransition)
      , ("nuts", 10000, nutsBaseTransition)
      , ("mh-prog", 2000, mhProgressiveTransition)
      , ("custom", 3333, customTransition)
      , ("random", 10000, randomTransition)
      ]
    chains = [
        ("rosenbrock", rosenbrockChain)
```

```
  , ("beale", bealeChain)
  , ("himmelblau", himmelblauChain)
  , ("bnn", bnnChain)
  ]
```

To run the chains assembled in 'tracer' we just map a function across them, pairing each target with each transition and executing the transition for the specified number of iterations.

5.7.4 Results

The four target functions and the six transitions used to sample from them result in 24 total chains, each of which presents some useful information. The floor-truncated effective sample sizes (ESS) for each chain (as reported by *coda*'s 'effectiveSize' function) is summarized in Table 5.5.

We thus have two metrics to evaluate the chains on: an informal visual measure of how well the chain appears to have traversed the space, and an ESS calculation that attempts to measure the efficiency of the chain. The ESS performance is a mixed-bag, providing some useful information in places but failing to capture important information about the chains' performance.

The traces over the Rosenbrock are displayed in Figure 5.6. They all appear to capture most of the mass of the distribution, but can be distinguished by how well they traverse into the tails. The HMC transition performs particularly well here, while the MH transition operator has the most difficulty moving into low-probability regions of the space. It's worth noting that the 'random' and 'custom' compound transitions appear to have performed comparably well to the NUTS transition. This is not surprising for 'custom' — which performs a primitive NUTS transition on every iteration — but the 'random' strategy only uses a

gradient calculation with probability 0.1, opting for MH or slice transitions otherwise. In terms of ESS, the NUTS and 'custom' transitions dominate, with MH also performing strongly.

The chains all appear to cover the high-probability areas of the BNN density well, with the MH, 'random', and 'custom' transitions possibly eking out the competition. However the 'custom' and 'random' operators yield the highest ESS, followed by MH and 'mh-progressive'. It's interesting that the solely gradient-based proposals perform worst in terms of ESS, whereas the 'random' and 'custom' operators — which also make use of a gradient-based primitive transition, either certainly or with some probability — are buoyed. It's difficult to tell if this is the primitive slice transition speaking, however.

Things are more interesting on the Beale density, in which two of the primitive operators — HMC and NUTS — completely fail to jump into to the secondary area of high probability. The other transitions all seem to perform comparably. The HMC and NUTS chains report high ESS numbers despite failing to locate an appreciable area of probability; the other chains all perform comparable to each other.

Finally there is the Himmelblau density containing four distinct and disparate modes of probability. Here, the 'custom' transition is the *only* one that successfully finds more than one mode; the rest remain stuck in a single mode (not always the same mode) and stay there for the course of the trace. The ESS figures can be misleading with respect to their performance: the 'custom' transition has the lowest ESS by far, while all the others report ESS numbers far in excess of it. HMC in particular reports impossibly high numbers (far in excess of the trace size).

Again, the purpose of these simulations was to simply illustrate a mix of transition operators and look at how they performed. But it seems that 'custom' transition indeed performed admirably throughout. It consists of a random choice be-

tween a Metropolis and Hamiltonian transition (strongly favoring the Metropolis transition), followed by a deterministic sequence of a slice sample and a NUTS transition.

By blending a number of strong transitions together it seems like it may be possible to hedge one's sampling risk, though more careful study would be required to quantify this in any serious way.

5.8 Extensions

5.8.1 Annealing

Recall from Section 5.5.1 that the target function we're sampling from is included in the 'Chain a' type that represents the state of our Markov chain. This has until now been technically unnecessarily as we could have provided the target as read-only information (using, say, the *reader monad*, not discussed here).

It's also possible to manipulate the target function in order to propose useful transitions. This must be done carefully, as recklessly altering the target function obviously changes the distribution being sampled (consider replacing the log-objective function with 'const 0', for example). But with disciplined use it allows one to incorporate, for example, an *annealing schedule* that warps the target according to some temperature. This can make it easier for a Markov chain to visit some distant region of probability that it would normally have difficulty transitioning to. Annealing allows one to 'flatten' or 'stretch' the target distribution in order to make it easier to move about in.

We can implement a function that takes a base transition operator and uses it to transition the state over some annealed space, by manipulating the target function cached in the state. Such a 'transition operator transformer' can be defined

as follows:

```
  anneal
    :: PrimMonad m
    => Double
    -> Transition m Double
    -> Transition m Double
  anneal invTemp baseTransition
    | invTemp < 0 = error "anneal: invalid temperture"
    | otherwise = do
        originalTarget <- gets objectiveFunction
        let annealedTarget = annealer invTemp originalTarget
        modify (useTarget annealedTarget)
        baseTransition
        modify (useTarget originalTarget)
        gets parameterSpacePosition

  annealer :: Double -> Target Double -> Target Double
  annealer invTemp target = Target annealedL annealedG where
    annealedL xs = invTemp * logObjective target xs
    annealedG    =
      case gradient target of
        Nothing -> Nothing
        Just g  -> Just (V.map (* invTemp) . g)

  useTarget :: Target a -> Chain a -> Chain a
  useTarget newTarget (Chain current _ _ store) =
    Chain current newTarget (logObjective newTarget current) store
```

An annealed transition operator is defined by swapping in the annealed target, performing the base transition on the annealed space, and then swapping the original target back in.

The 'anneal' combinator allows us to write transition operators that operate one or more times in the annealed space. The following operator reuses the 'randomTransition' operator from earlier in the chapter, setting it to an annealing schedule based on the provided inverse temperatures:

```
annealingTransition :: PrimMonad m => Transition m Double
annealingTransition = do
  anneal 0.70 randomTransition
  anneal 0.05 randomTransition
  anneal 0.05 randomTransition
  anneal 0.70 randomTransition
  randomTransition
```

The 'annealingTransition' performs four 'randomTransitions' on the annealed space before executing a fifth on the original space.

The resulting transition operator performs well on the Himmelblau density from before; on a trace of 2000 iterations it finds all four regions of probability, displayed in Figure 5.10.

5.8.2 Coupled Chains and Tempering

The structure of our transition operator type is also amenable to MCMC techniques that rely on *coupled* chains, such as *parallel tempering* [Neal, 1996]. We don't explore this topic any further, but note that the type of transition over coupled chains is easy to express:

```
type CoupledTransition m a =
  StateT (Chain a, Chain a) (Prob m) (Vector a)
```

Indeed for n coupled chains this could be replaced by:

```
type CoupledTransition m a = StateT [Chain a] (Prob m) (Vector a)
```

5.9 Summary and Comparison to Other Work

A *declarative*-style language is not a full probabilistic programming language, with support for denoting models, sampling from them, and so on. It is a simple embedded language for building Markov transition operators in a type-safe fashion, such that the resulting transitions can be used to drive a Markov chain over a log-density function. The means by which that log-density function is gotten is kept abstract, such that we can treat it as being supplied by some arbitrary frontend. *declarative* can then be used to customize how inference is performed in the backend.

A similar feature for authoring custom transitions exists in PyMC3 [Salvatier et al., 2016] under a system of 'step methods'. As is typical in Python, this system is implemented using a class called MCMC. An immediate advantage of the present implementation by virtue of being embedded in Haskell is that custom transition operators are stitched together in a way that gets checked at compile time. This may seem academic (when do transition operators get *that* unwieldy) but being able to check transitions at compile time is useful as a development aid. If custom transitions in *declarative* pass a typechecker, they at least 'work' in a weak sense, saving the researcher the hassle of actually running his or her Markov chain to spot the error.

Ścibior et al. [2017] discussed a similar, in-development system for denoting and manipulating transitions such that they can be used as simple building blocks towards arbitrarily more complex ones. The described system is more ambitious in scope than that presented here. We focus on a language for building transition operators, while Ścibior et al. [2017] seeks to mix and match entire inference algorithms under a formalized denotational semantics. Narayanan and

Shan [2014] developed a combinator library for MCMC sampling that is similar in scope to *declarative*, though the extent of its features is not clear.

5.10 Conclusion

The monadic language presented here falls out naturally from characterizing Markov transitions as values with type in the state monad. The resulting combinators are extraordinarily easy to implement — the only remotely tricky part is composing operators by mixing, which requires randomness from an underlying probability monad.

A language for mixing and matching transition operators in this fashion is desirable even as a low-cost way to explore the characteristics of various MCMC algorithms on targets of one's choice. One can start with a composite transition consisting of various strong primitive transitions (in order to 'hedge operator risk') and then refine it to some more concise operator for the problem via trial and error. The embedded language makes it trivial to explore alternate transition operators at almost no cost.

There is a large expanse here for future work: adding new primitive transition operators, characterizing scenarios where specific compound transition operators could prove useful, and so on. It would be particularly desirable to find a kind of 'killer app' for this kind of work, i.e. a case in which some compound transition built from flexible primitives soundly beats any primitive transition on its own.

An expanded version of this work including support for both continuous and discrete parameters was implemented towards an open-source inference backend for the (now hibernating) *Baysig* project, a standalone Haskell-inspired language with similar aims to BUGS or Stan but focusing on increased expressiveness.

A major avenue of future work would be to use this language as a backend onto an embedded probabilistic programming language like that developed in Chapter 4. Since the presentation here has been kept general, there is no technical hurdle to doing this: given a free monad-captured AST, we could write an interpreter that evaluates a posterior when provided concrete values for the parameters.

Label	ESS(x)	ESS(y)
mh-rosenbrock	104	91
mh-progressive-rosenbrock	16	11
hmc-rosenbrock	14	16
nuts-rosenbrock	285	500
custom-rosenbrock	260	261
random-rosenbrock	57	33
mh-himmelblau	306	286
mh-progressive-himmelblau	799	389
hmc-himmelblau	16678	41450
nuts-himmelblau	5657	3478
custom-himmelblau	4	9
random-himmelblau	2572	3267
mh-beale	12	47
mh-progressive-beale	26	60
hmc-beale	489	663
nuts-beale	437	749
custom-beale	31	58
random-beale	21	64
mh-bnn	115	115
mh-progressive-bnn	126	111
hmc-bnn	45	39
nuts-bnn	52	54
custom-bnn	197	234
random-bnn	298	295

Figure 5.5: Floor-truncated effective sample size (ESS) by coordinate for chains driven by various transition operators over the Rosenbrock, Himmelblau, Beale, and BNN densities.

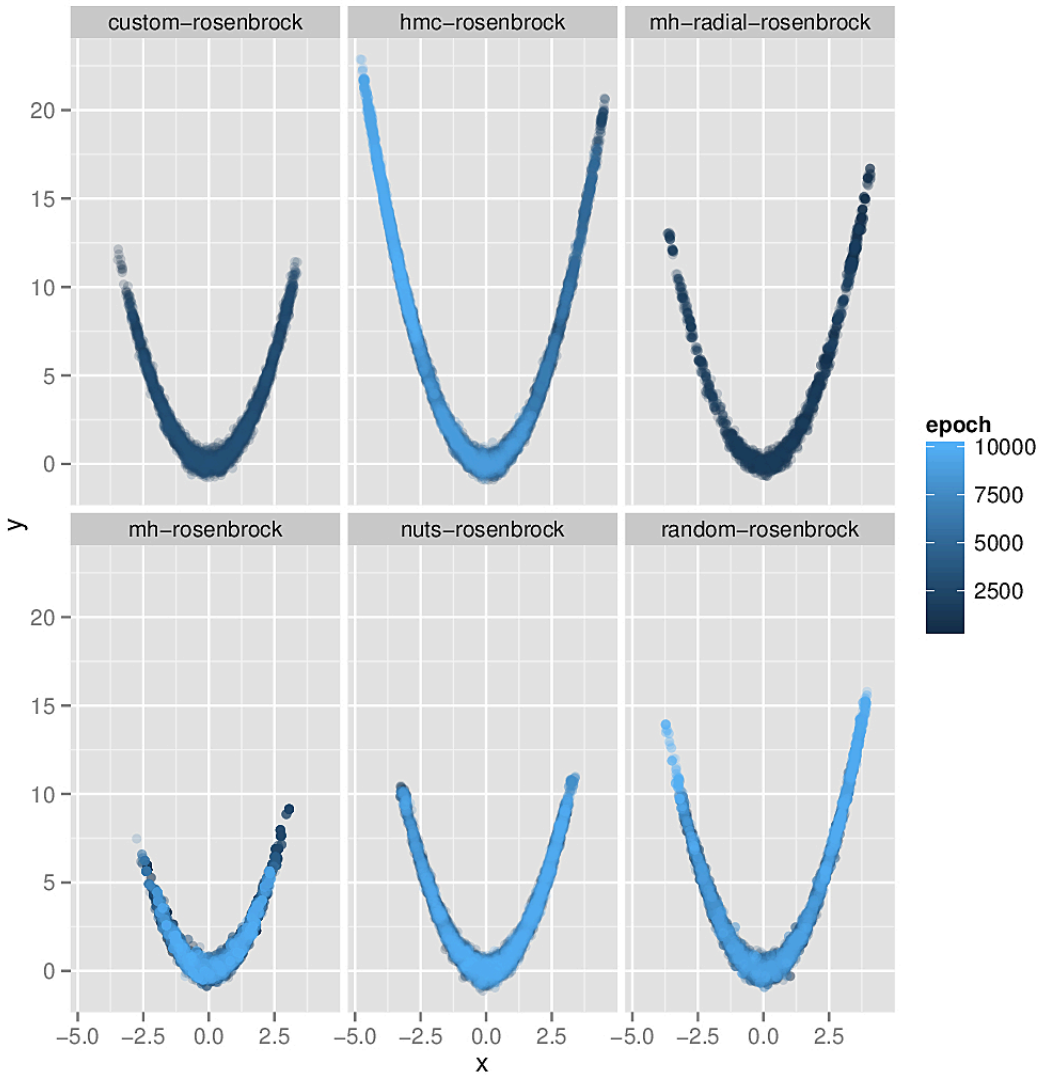


Figure 5.6: Traces of Markov chains over the log-Rosenbrock density. Each trace was taken for a total of 10000 primitive transitions, so the 'mh-progressive' and 'custom' traces contain less points (2000 and 3333 respectively).

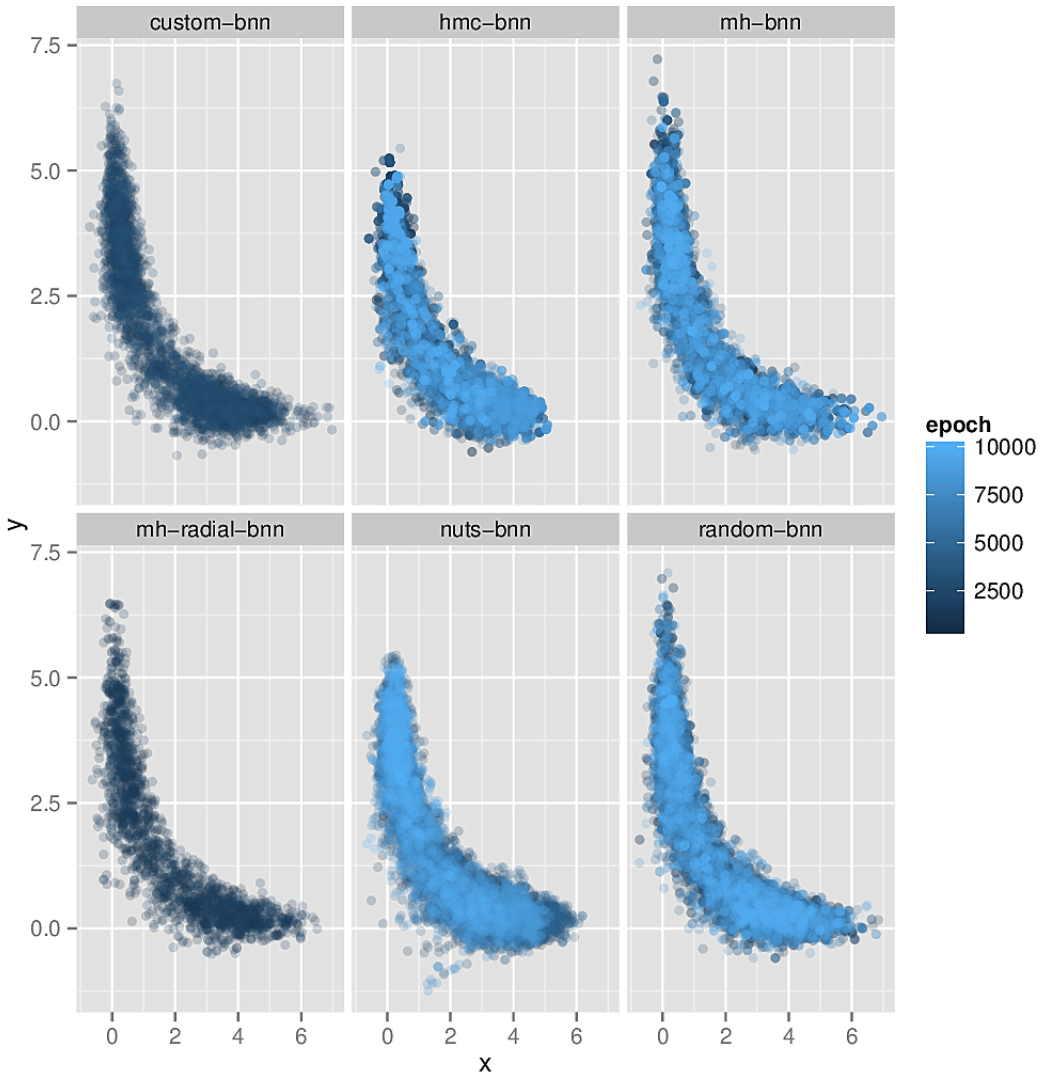


Figure 5.7: Traces of Markov chains over the log-BNN density. Each trace was taken for a total of 10000 primitive transitions, so the 'mh-progressive' and 'custom' traces contain less points (2000 and 3333 respectively).

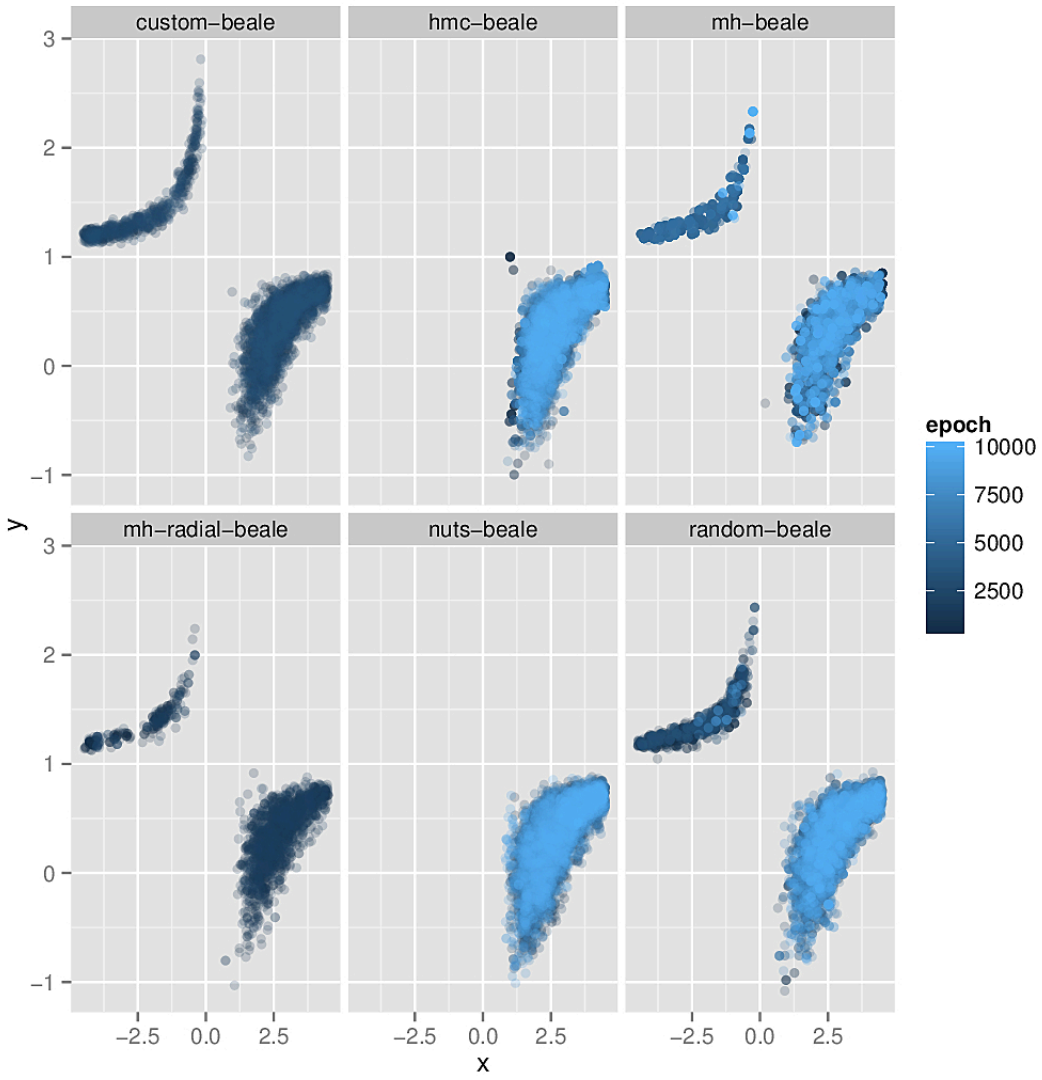


Figure 5.8: Traces of Markov chains over the log-Beale density. Each trace was taken for a total of 10000 primitive transitions, so the 'mh-progressive' and 'custom' traces contain less points (2000 and 3333 respectively). Note that the pure gradient-based algorithms fail to locate the topmost mode of probability.

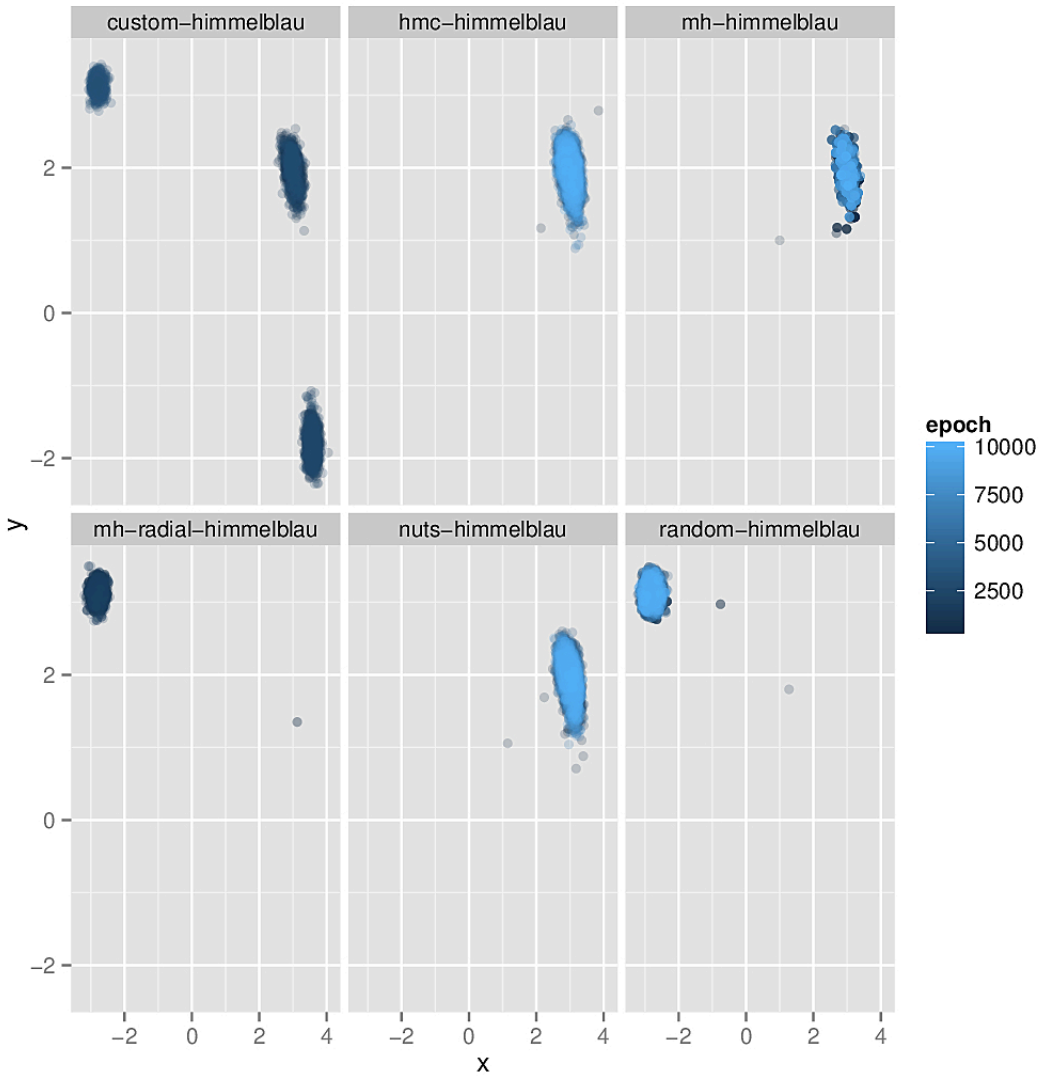


Figure 5.9: Traces of Markov chains over the log-Himmelblau density. Each trace was taken for a total of 10000 primitive transitions, so the 'mh-progressive' and 'custom' traces contain less points (2000 and 3333 respectively). Note that all but one of the algorithms fail to find more than one mode of probability, with the 'custom' transition alone locating three.

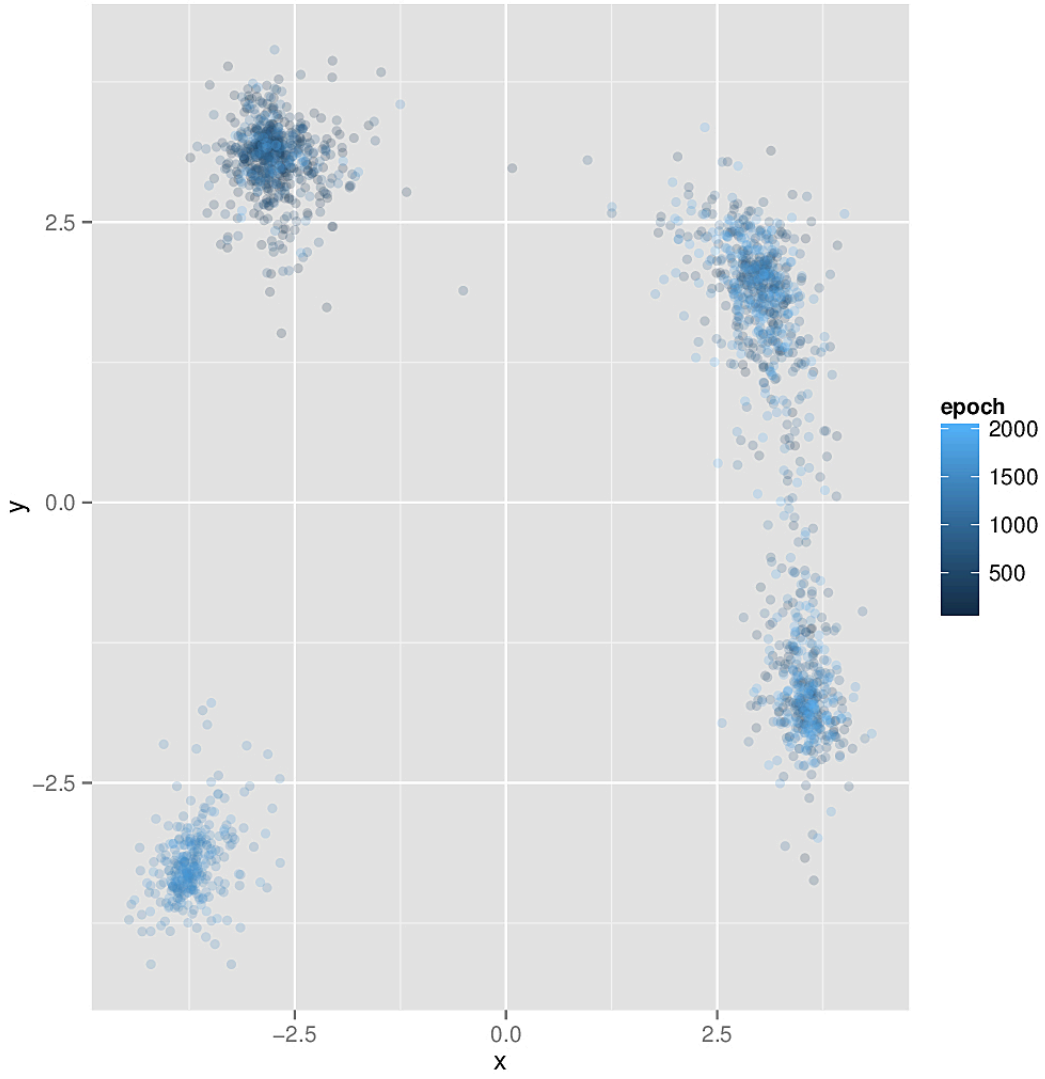


Figure 5.10: A trace of 2000 iterations of the annealingTransition operator on the log-Himmelblau density. Note that it successfully finds all four modes of probability.

Chapter 6

Conclusion

This dissertation has supported the thesis that **novel and useful domain-specific languages for solving problems in Bayesian statistics can be embedded in statically-typed, purely-functional programming languages**.

In this chapter we'll summarize the primary contributions of the dissertation and demonstrate that they defend the thesis.

6.1 Representing Probability Distributions

Chapter 3 provided a large tour of the Giry monad, and how it can be implemented as a shallowly-embedded DSL for integration.

We formally characterized the functor, applicative, and monadic structure of the Giry monad using probabilistic semantics. In particular, the functorial structure was shown to encapsulate the probabilistic concept of pushforward measure, while the applicative structure was shown to encode product measure and independence of measurable functions.

We then implemented the Giry monad as a shallowly-embedded DSL and characterized the implementation via a restricted continuation monad. Measures were represented in the embedded language as integration procedures. We demonstrated various functions for building, manipulating, and querying measures (with special note to the measure convolution, moment/cumulant-generating function, and cumulative distribution function queries), but noted that the measure representation was too computationally demanding to be useful in practice.

6.2 Representing Structured Probabilistic Models

Chapter 4 presented a novel and useful way to preserve the structure of a probabilistic model. We detailed a procedure by which a typed, monadic probabilistic programming language can be deeply-embedded in a statically-typed functional host by exploiting the properties of the free monad over a probabilistic base functor.

We then presented a a comonadic MCMC algorithm that used the cofree comonad to represent execution traces of probabilistic programs. The free applicative was also demonstrated to be useful for encoding conditional independence statically.

6.3 Declarative Markov Chains

Chapter 5 presented a novel and useful shallowly-embedded language for building Markov transition operators. Markov transition operators were demonstrated to be composable either by deterministic concatenation or by probabilistic mixing (or combinations of the two), and a simple grammar was presented that described how complex transition operators could be built from primitive ones via either of these operations.

We demonstrated that the grammar could be implemented as a shallowly-embedded language based on monadic combinators, using a state monad transformer over a probability monad. We then presented implementations of a number of popular primitive transition operators in Haskell and used them to build a handful of composite operators, which were then demonstrated in Markov chains over a number of example spaces.

We also demonstrated that the language can be extended to support other popular MCMC methods, such as annealing and tempering.

6.4 Commentary and Future Work

The work contained in this dissertation emphasizes the themes of *composition*, *abstraction*, and *structure* — important concepts in statically-typed purely functional languages. Emphasis on these ideas naturally leads to the construction of languages, which are nothing more than collections of composable terms bound together by a structured grammar. This pattern of language engineering is a fruitful way to build software in a number of domains, and statistics proves to be one of them.

Embedding these simple languages dodges many of the hairier parts of implementing programming languages and building compilers, since we can hijack much of the more complicated and burdensome-to-implement features from our host language. What's more, one can often get pretty far just by using a shallow embedding, in which the terms of the embedded language are defined directly by their semantics. In only one case — to support a structured probabilistic programming language — did we need to use a deep embedding that preserved abstract syntax.

Most of the features developed throughout this dissertation rely on important

underlying features of statically-typed, purely functional languages. Monads — surely the prime driver of innovation — are difficult to express faithfully without a strong type system and functional purity. Newcomers to languages like Haskell are often put off by the apparent restrictions placed on them by monads, but closer inspection quickly illustrates that they can often encapsulate some kind of important law-preserving structure that can be exploited to implement some desirable feature.

More generally, the idea of exploring and exploiting algebraic and categorical structures like functors, monoids, applicatives, monads, and what have you is surely relatively new to statistics and machine learning research. And yet there seems to be a lot of low-hanging fruit in this area, both in terms of enabling practical software implementations and by illuminating similarities or shared structure with other well-understood problems.

There is surely an endless amount of work to explore where this dissertation leaves off. Probabilistic programming is magnificent in the number of interesting technical areas it touches, and modern approaches beyond the imperative, direct style of the venerable BUGS software have really only been investigated within the last ten years. There are few other domains that this author is aware of where statistics, computer science, and pure mathematics are connected so intimately.

One of the most important avenues of future work would also be to make the various pieces of software developed during the course of this research *production-ready*. These implementations necessarily have a proof-of-concept feel about them; they are designed to explore new areas and possibilities, rather than to immediately take the place of existing FORTRAN code. Considerable software engineering would be required to make these embedded languages synchronized, user-friendly, featureful, and performant. But the beauty is that this can certainly be done — once one knows the tricks, domain specific languages can be embedded in languages like Haskell with much greater ease (and much fewer lines of code) than it takes to implement a standalone language like Stan, for example.

The BUGS and Stan projects were large pieces of institutional-supported software involving multiple developers over many years, but it's not hard to imagine a finely-tuned set of open source packages embedded directly in a popular programming language becoming a serious competitor in the future — there are many attractive features of Haskell-like languages that simply don't exist anywhere amongst the competition.

Appendices

Appendix A

Cofree Encoding and MCMC Code

Below is the full code for the comonadic inference algorithm. Note that it consists of a few functions that must be pattern matched against every branch of an underlying base functor; this makes the code somewhat repetitive, but it is isolated well. It is further motivation for keeping the set of probabilistic primitives minimal.

```
import Control.Comonad
import Control.Comonad.Cofree
import Control.Monad.Free
import Control.Monad.ST
import Data.Bits
import Data.Dynamic
import Data.Maybe
import Data.Monoid
import qualified Data.Vector as V
import Data.Void
import Data.Word
import qualified System.Random.MWC as MWC
```

```
import System.Random.MWC.Probability (Prob)
import qualified System.Random.MWC.Probability as Prob

import Control.Monad.Trans.Cont

-- types ----------------------------------------------------------------------

data ModelF a r =
    BernoulliF !Double (Bool -> r)
  | BetaF !Double !Double (Double -> r)
  | NormalF !Double !Double (Double -> r)
  | DiracF a
  deriving Functor

data Dist =
    Bernoulli !Double
  | Beta !Double !Double
  | Normal !Double !Double
  | Dirac
  deriving (Eq, Show)

type Distribution a = Free (ModelF a)

type Model b = forall a. Distribution a b

type Terminating a = Distribution a Void

type Execution a = Cofree (ModelF a) Node

data Node = Node {
    nodeScore   :: !Double
  , nodeValue   :: !Dynamic
```

```
  , nodeSeed    :: !MWC.Seed
  , nodeHistory :: [Dynamic]
  , nodeDist    :: !Dist
  } deriving Show

-- primitive terms -------------------------------------------------------

beta :: Double -> Double -> Distribution a Double
beta a b = liftF (BetaF a b id)

bernoulli :: Double -> Distribution a Bool
bernoulli p = liftF (BernoulliF p id)

normal :: Double -> Double -> Distribution a Double
normal m s = liftF (NormalF m s id)

dirac :: a -> Distribution a b
dirac x = liftF (DiracF x)

-- additional distributions ----------------------------------------------

geometric :: Double -> Distribution a Int
geometric p = loop where
  loop = do
    accept <- bernoulli p
    if   accept
    then return 1
    else fmap succ loop

uniform :: Distribution a Double
uniform = beta 1 1
```

```haskell
-- sampling ------------------------------------------------------------------

toSampler :: Distribution a a -> Prob IO a
toSampler = iterM $ \case
  BernoulliF p f -> Prob.bernoulli p >>= f
  BetaF a b f    -> Prob.beta a b >>= f
  NormalF m s f  -> Prob.normal m s >>= f
  DiracF x       -> return x

simulate :: Prob IO a -> IO a
simulate model = MWC.withSystemRandom . MWC.asGenIO $ Prob.sample model

-- densities -----------------------------------------------------------------

logDensityBernoulli :: Double -> Bool -> Double
logDensityBernoulli p x
    | p < 0 || p > 1 = log 0
    | otherwise      = b * log p + (1 - b) * log (1 - p)
  where
    b = if x then 1 else 0

logDensityBeta :: Double -> Double -> Double -> Double
logDensityBeta a b x
  | x <= 0 || x >= 1 = log 0
  | a < 0 || b < 0   = log 0
  | otherwise        = (a - 1) * log x + (b - 1) * log (1 - x)

logDensityNormal :: Double -> Double -> Double -> Double
logDensityNormal m s x
  | s <= 0    = log 0
  | otherwise = negate (log s) - (x - m) ^ 2 / (2 * s ^ 2)
```

```
logDensityDirac :: Eq a => a -> a -> Double
logDensityDirac a x
  | a == x    = 0
  | otherwise = negate (1 / 0)

-- execution: initializing ---------------------------------------------------

execute :: Typeable a => Terminating a -> Execution a
execute = executeGeneric (42, 108512)

executeGeneric
  :: Typeable a => (Word32, Word32) -> Terminating a -> Execution a
executeGeneric = annotate where
  annotate seeds term = case term of
    Pure r    -> absurd r
    Free cons ->
      let (nextSeeds, genGenerator) = xorshift seeds
          seed  = MWC.toSeed (V.singleton genGenerator)
          ann   = initialize seed cons
      in  ann :< fmap (annotate nextSeeds) cons

samplePurely
  :: Typeable a => Prob (ST s) a -> Prob.Seed -> ST s (Dynamic, Prob.Seed)
samplePurely prog seed = do
  prng     <- MWC.restore seed
  value    <- MWC.asGenST (Prob.sample prog) prng
  nodeSeed <- MWC.save prng
  if   seed == nodeSeed
  then error "a generator failed to step!"
  else return (toDyn value, nodeSeed)

initialize :: Typeable a => MWC.Seed -> ModelF a b -> Node
```

```
initialize seed = \case
  BernoulliF p _ -> runST $ do
    (nodeValue, nodeSeed) <- samplePurely (Prob.bernoulli p) seed
    let nodeScore   = logDensityBernoulli p (unsafeFromDyn nodeValue)
        nodeHistory = mempty
        nodeDist    = Bernoulli p
    return Node {..}

  BetaF a b _ -> runST $ do
    (nodeValue, nodeSeed) <- samplePurely (Prob.beta a b) seed
    let nodeScore   = logDensityBeta a b (unsafeFromDyn nodeValue)
        nodeHistory = mempty
        nodeDist    = Beta a b
    return Node {..}

  NormalF m s _ -> runST $ do
    (nodeValue, nodeSeed) <- samplePurely (Prob.normal m s) seed
    let nodeScore   = logDensityNormal m s (unsafeFromDyn nodeValue)
        nodeHistory = mempty
        nodeDist    = Normal m s
    return Node {..}

  DiracF a -> Node 0 (toDyn a) seed mempty Dirac

-- execution: scoring and running ---------------------------------------------

scoreWithModel
  :: (Typeable a, Eq a)
  => Terminating a -> Dynamic
  -> Double
scoreWithModel model x = case model of
  Pure r    -> absurd r
```

```
  Free cons -> case cons of
    BernoulliF p _ -> logDensityBernoulli p (unsafeFromDyn x)
    BetaF a b _    -> logDensityBeta a b (unsafeFromDyn x)
    NormalF m s _  -> logDensityNormal m s (unsafeFromDyn x)
    DiracF a       -> logDensityDirac a (unsafeFromDyn x)

score :: Execution a -> Double
score = loop 0 where
  loop !acc (Node {..} :< cons) = case cons of
    BernoulliF _ k -> loop (acc + nodeScore) (k (unsafeFromDyn nodeValue))
    BetaF _ _ k    -> loop (acc + nodeScore) (k (unsafeFromDyn nodeValue))
    NormalF _ _ k  -> loop (acc + nodeScore) (k (unsafeFromDyn nodeValue))
    DiracF _       -> acc

depth :: Execution a -> Int
depth = loop 0 where
  loop !acc (Node {..} :< cons) = case cons of
    BernoulliF _ k -> loop (succ acc) (k (unsafeFromDyn nodeValue))
    BetaF _ _ k    -> loop (succ acc) (k (unsafeFromDyn nodeValue))
    NormalF _ _ k  -> loop (succ acc) (k (unsafeFromDyn nodeValue))
    DiracF _       -> succ acc

step :: Typeable a => Execution a -> Execution a
step prog@(Node {..} :< _) = stepWithInput nodeValue prog

stepWithInput :: Typeable a => Dynamic -> Execution a -> Execution a
stepWithInput value prog = case unwrap prog of
  BernoulliF _ k -> k (unsafeFromDyn value)
  BetaF _ _ k    -> k (unsafeFromDyn value)
  NormalF _ _ k  -> k (unsafeFromDyn value)
  DiracF _       -> prog
```

```
run :: Typeable a => Execution a -> a
run prog = case unwrap prog of
  DiracF a -> a
  _        -> run (step prog)

runWithInput :: Typeable a => Dynamic -> Execution a -> a
runWithInput value = run . stepWithInput value

stepGenerators :: Functor f => Cofree f Node -> Cofree f Node
stepGenerators = extend stepGenerator

stepGenerator :: Cofree f Node -> Node
stepGenerator (Node {..} :< cons) = runST $ do
  (_, nseed) <- samplePurely (Prob.beta 1 1) nodeSeed
  return Node {nodeSeed = nseed, ..}

-- mcmc: perturb ---------------------------------------------------------------

perturb :: Execution a -> Execution a
perturb = extend perturbNode

perturbNode :: Execution a -> Node
perturbNode (node@Node {..} :< cons) = case cons of
  BernoulliF p _ -> runST $ do
    (nvalue, nseed) <- samplePurely (Prob.bernoulli p) nodeSeed
    let nscore   = logDensityBernoulli p (unsafeFromDyn nvalue)
        ndist    = Bernoulli p
    return $! Node nscore nvalue nseed nodeHistory ndist

  BetaF a b _ -> runST $ do
    (nvalue, nseed) <- samplePurely (Prob.beta a b) nodeSeed
    let nscore   = logDensityBeta a b (unsafeFromDyn nvalue)
```

```
        ndist    = Beta a b
    return $! Node nscore nvalue nseed nodeHistory ndist

  NormalF m s _ -> runST $ do
    (nvalue, nseed) <- samplePurely (Prob.normal m s) nodeSeed
    let nscore   = logDensityNormal m s (unsafeFromDyn nvalue)
        ndist    = Normal m s
    return $! Node nscore nvalue nseed nodeHistory ndist

  DiracF a -> node

-- mcmc: markov chain ------------------------------------------------------

invert
  :: (Eq a, Typeable a, Typeable b)
  => Int -> [a] -> Model b -> (b -> a -> Double)
  -> Model (Execution b)
invert epochs obs prior ll = loop epochs (execute (prior >>= dirac)) where
  loop n current
    | n == 0    = return current
    | otherwise = do
        let proposal = perturb current

            valueAtCurrent       = run current
            valueAtProposal      = run proposal
            currentLl            = ll valueAtCurrent
            proposalLl           = ll valueAtProposal

            currentContribution  = sum (fmap currentLl obs)
            proposalContribution = sum (fmap proposalLl obs)

            currentScore         = score current + currentContribution
```

```
            proposalScore        = score proposal + proposalContribution

            fw = negate (log (fromIntegral (depth current))) + score proposal
            bw = negate (log (fromIntegral (depth proposal))) + score current

            prob = moveProbability currentScore proposalScore bw fw

        accept <- bernoulli prob
        let next = if accept then proposal else stepGenerators current
        loop (pred n) (snapshot next)

moveProbability :: Double -> Double -> Double -> Double -> Double
moveProbability current proposal bw fw =
    whenNaN 0 (exp (min 0 (proposal - current + bw - fw)))
  where
    whenNaN val x
      | isNaN x   = val
      | otherwise = x

-- Record the present value of every node in its history.
snapshot :: Functor f => Cofree f Node -> Cofree f Node
snapshot = extend snapshotValue

snapshotValue :: Cofree f Node -> Node
snapshotValue (Node {..} :< _) = Node { nodeHistory = history, .. } where
  history = nodeValue : nodeHistory

-- Data.Bits.Extended ------------------------------------------------------

-- | A pure xorshift implementation.
--
--   See: https://en.wikipedia.org/wiki/Xorshift.
```

```
xorshift :: (Bits t, Num t) => (t, t) -> ((t, t), t)
xorshift (s0, s1) = ((s1, s11), s11 + s1) where
  x = s0 `xor` shiftL s0 23
  s11 = x `xor` s1 `xor` (shiftR x 17) `xor` (shiftR s1 26)

-- Data.Dynamic.Extended ------------------------------------------------------

unsafeFromDyn :: Typeable a => Dynamic -> a
unsafeFromDyn = fromJust . fromDynamic
```

Appendix B

Primitive Markov Transition Code

Below is the full code corresponding to the various primitive transition operators described in Chapter 5. Note that the implementations depend on a number of third-party libraries; see the repositories listed in Chapter 1 for details on these.

Each primitive transition below depends on the 'Declarative.Core' and 'Declarative.Types' modules which contain the code included in Chapter 5, plus the following simple construction for the 'OptionalStore' parameter:

```
type OptionalStore = HashMap Algorithm OptionalInfo

data Algorithm =
    MH
  | HMC
  | MALA
  | Slice
  | NUTS
  deriving (Eq, Show)

instance Hashable Algorithm where
```

```
    hashWithSalt n = hashWithSalt n . show

  data OptionalInfo =
      ODouble Double
    | OInt Int
    | OPair (OptionalInfo, OptionalInfo)
    deriving (Eq, Show)
```

B.1 Metropolis-Hastings

```
{-# LANGUAGE RankNTypes #-}

import Control.Monad.Primitive
import Control.Monad.State.Strict
import Data.HashMap.Strict (HashMap)
import qualified Data.HashMap.Strict as HashMap
import Data.Vector.Unboxed (Vector)
import qualified Data.Vector.Unboxed as V
import Declarative.Core
import Declarative.Types
import Statistics.Distribution
import Statistics.Distribution.Normal

metropolisHastings :: PrimMonad m => Maybe Double -> Transition m Double
metropolisHastings e = do
  Chain current target _ store <- get
  let sd = getStandardDeviation e store
  proposed <- lift $ perturb current sd
  zc       <- lift unit
  let next     = nextState target current proposed sd zc
      newStore = updateStandardDeviation sd store
```

```
  put $ Chain next target (logObjective target next) newStore
  return next

sphericalGaussian :: Vector Double -> Vector Double -> Double -> Double
sphericalGaussian xs mu sd = product $ zipWith density normalDists xsAsList
  where
    xsAsList    = V.toList xs
    muAsList    = V.toList mu
    normalDists = map (`normalDistr` sd) muAsList

perturb
  :: PrimMonad m
  => Vector Double
  -> Double
  -> Prob m (Vector Double)
perturb q sd = V.mapM (`normal` sd) q

acceptRatio
  :: Target Double -> Vector Double -> Vector Double -> Double -> Double
acceptRatio target current proposed sd = exp . min 0 $
    logObjective target proposed + log (sphericalGaussian current proposed sd)
  - logObjective target current  - log (sphericalGaussian proposed current sd)

nextState
  :: Target Double
  -> Vector Double
  -> Vector Double
  -> Double
  -> Double
  -> Vector Double
nextState target current proposed sd z
    | z < acceptProb = proposed
```

```
    | otherwise      = current
  where
    ratio = acceptRatio target current proposed sd
    acceptProb | isNaN ratio = 0
               | otherwise   = ratio

getStandardDeviation :: Maybe Double -> OptionalStore -> Double
getStandardDeviation (Just sd) _   = sd
getStandardDeviation Nothing store = sd where
  (ODouble sd) = HashMap.lookupDefault (ODouble 1.0) MH store

updateStandardDeviation :: Double -> OptionalStore -> OptionalStore
updateStandardDeviation sd = HashMap.insert MH (ODouble sd)
```

B.2 Slice Sampling

```
{-# OPTIONS_GHC -Wall #-}
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE DoAndIfThenElse #-}
{-# LANGUAGE RankNTypes #-}

import Control.Monad
import Control.Monad.Primitive
import Control.Monad.State.Strict
import Data.Vector.Unboxed (Vector, Unbox)
import qualified Data.Vector.Unboxed as V
import qualified Data.Vector.Unboxed.Mutable as V hiding (length)
import Declarative.Core
import Declarative.Types
import System.Random.MWC hiding (uniform)
```

```
slice :: PrimMonad m => Double -> Transition m Double
slice e = do
  Chain position target _ _ <- get
  let n = V.length position
  forM_ [0..n - 1] $ \j -> do
    Chain q _ _ store <- get
    height  <- liftM log $ lift $ uniform (0, exp $ logObjective target q)
    bracket <- lift . lift $ findBracket (logObjective target) j e height q
    next    <- lift $ rejection (logObjective target) j bracket height q
    put $ Chain next target (logObjective target next) store

  Chain q _ _ _ <- get
  return q

findBracket :: (Num b, Ord a, PrimMonad m, Unbox b)
  => (Vector b -> a)
  -> Int
  -> b
  -> a
  -> Vector b
  -> m (b, b)
findBracket f j step height xs = go step xs xs where
  go !e !bl !br
    | f bl <  height && f br <  height =
        return (bl `V.unsafeIndex` j , br `V.unsafeIndex` j)
    | f bl <  height && f br >= height = do
        br0 <- expandBracketRight j e br
        go (2 * e) bl br0
    | f bl >= height && f br <  height = do
        bl0 <- expandBracketLeft j e bl
        go (2 * e) bl0 br
    | otherwise = do
```

```
        bl0 <- expandBracketLeft  j e bl
        br0 <- expandBracketRight j e br
        go (2 * e) bl0 br0

expandBracketBy
  :: (PrimMonad m, Unbox a)
  => (a -> a -> a) -> Int -> a -> Vector a -> m (Vector a)
expandBracketBy f j e xs = do
  v  <- V.thaw xs
  xj <- V.unsafeRead v j
  V.unsafeWrite v j (f xj e)
  V.freeze v

expandBracketRight
  :: (Num a, Unbox a, PrimMonad m)
  => Int -> a -> Vector a -> m (Vector a)
expandBracketRight = expandBracketBy (+)

expandBracketLeft
  :: (Unbox a, Num a, PrimMonad m)
  => Int -> a -> Vector a -> m (Vector a)
expandBracketLeft  = expandBracketBy (-)

rejection
  :: (Ord b, Unbox a, PrimMonad m, Variate a)
  => (Vector a -> b)
  -> Int
  -> (a, a)
  -> b
  -> Vector a
  -> Prob m (Vector a)
rejection f j bracket height = go where
```

```
  go zs = do
    u    <- uniform bracket
    v    <- lift $ V.thaw zs
    lift $ V.unsafeWrite v j u
    cool <- lift $ V.freeze v
    if   f cool < height
    then go cool
    else return cool
```

B.3 Metropolis-Adjusted Langevin Diffusion

```
import Control.Monad
import Control.Monad.Primitive
import Control.Monad.State.Strict
import Declarative.Core
import Declarative.Types
import Data.HashMap.Strict (HashMap)
import qualified Data.HashMap.Strict as HashMap
import Data.Vector.Unboxed (Vector)
import qualified Data.Vector.Unboxed as V
import Statistics.Distribution
import Statistics.Distribution.Normal

mala :: PrimMonad m => Maybe Double -> Transition m Double
mala e = do
  Chain current target _ store <- get
  let step = getStepSize e store
  proposal <- lift $ perturb target current step
  zc       <- lift unit

  let cMean    = localMean target current step
```

```
      pMean    = localMean target proposal step
      next     = nextState target (current, cMean) (proposal, pMean) step zc
      newStore = updateStepSize step store

  put $ Chain next target (logObjective target next) newStore
  return next

sphereGauss :: Vector Double -> Vector Double -> Double -> Double
sphereGauss xs mu sd = product $ zipWith density normalDists xsAsList where
  xsAsList = V.toList xs
  muAsList = V.toList mu
  normalDists = map (`normalDistr` sd) muAsList

(.*) :: Double -> Vector Double -> Vector Double
z .* xs = V.map (* z) xs

(.+) :: Vector Double -> Vector Double -> Vector Double
xs .+ ys = V.zipWith (+) xs ys

localMean :: Target Double -> Vector Double -> Double -> Vector Double
localMean target position e = position .+ scaledGradient position where
  grad = handleGradient (gradient target)
  scaledGradient p = (0.5 * e * e) .* grad p

perturb
  :: PrimMonad m
  => Target Double
  -> Vector Double
  -> Double
  -> Prob m (Vector Double)
perturb target position e = do
  zs <- V.replicateM (V.length position) standardNormal
```

```
  return $ localMean target position e .+ (e .* zs)

getStepSize :: Maybe Double -> OptionalStore -> Double
getStepSize (Just step) _ = step
getStepSize Nothing store = step where
  (ODouble step) = HashMap.lookupDefault (ODouble 1.0) MALA store

updateStepSize :: Double -> OptionalStore -> OptionalStore
updateStepSize step = HashMap.insert MALA (ODouble step)

nextState
  :: Target Double
  -> (Vector Double, Vector Double)
  -> (Vector Double, Vector Double)
  -> Double
  -> Double
  -> Vector Double
nextState target (current, cMean) (proposal, pMean) e z
    | z < acceptProb = proposal
    | otherwise      = current
  where
    ratio = acceptRatio target (current, cMean) (proposal, pMean) e
    acceptProb | isNaN ratio = 0
               | otherwise   = ratio

acceptRatio
  :: Target Double
  -> (Vector Double, Vector Double)
  -> (Vector Double, Vector Double)
  -> Double
  -> Double
acceptRatio target (current, cMean) (proposal, pMean) e = exp . min 0 $
```

```
    logObjective target proposal + log (sphereGauss current pMean e)
  - logObjective target current  - log (sphereGauss proposal cMean e)
```

B.4 Hamiltonian Monte Carlo

```
{-# LANGUAGE RankNTypes #-}

import Control.Monad
import Control.Monad.Primitive
import Control.Monad.State.Strict
import Control.Monad.Trans
import qualified Data.HashMap.Strict as HashMap
import Data.Vector.Unboxed (Vector, Unbox)
import qualified Data.Vector.Unboxed as V
import Declarative.Core
import Declarative.Types

hamiltonian :: PrimMonad m => Maybe Double -> Maybe Int -> Transition m Double
hamiltonian e l = do
  Chain current target _ store <- get
  let (stepSize, nDisc) = getParameters e l store
      q0                = current
  r0 <- V.replicateM (V.length q0) (lift standardNormal)
  zc <- lift unit
  let (q, r)   = leapfrogIntegrator target q0 r0 stepSize nDisc
      next     = nextState zc target q0 q r0 r
      newStore = updateParameters stepSize nDisc store
  put $ Chain next target (logObjective target next) newStore
  return next

leapfrog
```

```
  :: Target Double
  -> Vector Double
  -> Vector Double
  -> Double
  -> (Vector Double, Vector Double)
leapfrog target q r e = (qf, rf) where
  rm       = adjustMomentum glTarget e q r
  qf       = adjustPosition e rm q
  rf       = adjustMomentum glTarget e qf rm
  glTarget = handleGradient $ gradient target

leapfrogIntegrator
  :: (Enum a, Eq a, Num a)
  => Target Double
  -> Vector Double
  -> Vector Double
  -> Double
  -> a
  -> (Vector Double, Vector Double)
leapfrogIntegrator target q0 r0 e = go q0 r0 where
  go q r 0 = (q, r)
  go q r n = let (q1, r1) = leapfrog target q r e
             in  go q1 r1 (pred n)

adjustMomentum
  :: (t -> Vector Double)
  -> Double
  -> t
  -> Vector Double
  -> Vector Double
adjustMomentum glTarget e q r = r .+ ((0.5 * e) .* glTarget q)
```

```haskell
adjustPosition :: (Unbox a, Num a) => a -> Vector a -> Vector a -> Vector a
adjustPosition e r q = q .+ (e .* r)

acceptanceRatio
  :: (Unbox a, Floating a)
  => (t -> a)
  -> t
  -> t
  -> Vector a
  -> Vector a -> a
acceptanceRatio lTarget q0 q1 r0 r1 =
  auxilliaryTarget lTarget q1 r1 - auxilliaryTarget lTarget q0 r0

auxilliaryTarget :: (Unbox a, Floating a) => (t -> a) -> t -> Vector a -> a
auxilliaryTarget lTarget q r = lTarget q - 0.5 * innerProduct r r

innerProduct :: (Unbox a, Num a) => Vector a -> Vector a -> a
innerProduct xs ys = V.sum $ V.zipWith (*) xs ys

(.*) :: (Unbox a, Num a) => a -> Vector a -> Vector a
z .* xs = V.map (* z) xs

(.-) :: (Unbox a, Num a) => Vector a -> Vector a -> Vector a
xs .- ys = V.zipWith (-) xs ys

(.+) :: (Unbox a, Num a) => Vector a -> Vector a -> Vector a
xs .+ ys = V.zipWith (+) xs ys

nextState
  :: Unbox a
  => Double
  -> Target a
```

```
  -> Vector a
  -> Vector a
  -> Vector Double
  -> Vector Double
  -> Vector a
nextState z target q0 q1 r0 r1
    | z < min 1 ratio = q1
    | otherwise       = q0
  where
    ratio = exp $ acceptanceRatio (logObjective target) q0 q1 r0 r1

getParameters :: Maybe Double -> Maybe Int -> OptionalStore -> (Double, Int)
getParameters (Just e) (Just l) _ = (e, l)
getParameters _  _          store = (e, l) where
  OPair (ODouble e, OInt l) =
    HashMap.lookupDefault (OPair (ODouble 0.05, OInt 20)) HMC store

updateParameters :: Double -> Int -> OptionalStore -> OptionalStore
updateParameters e l = HashMap.insert HMC (OPair (ODouble e, OInt l))
```

B.5 No U-Turn Sampler

```
{-# LANGUAGE DoAndIfThenElse #-}

import Control.Monad
import Control.Monad.Primitive
import Control.Monad.Trans
import Control.Monad.Trans.State.Strict
import Data.HashMap.Strict (HashMap)
import qualified Data.HashMap.Strict as HashMap
import Data.Vector.Unboxed (Vector, Unbox)
```

```
import qualified Data.Vector.Unboxed as V
import Declarative.Core
import Declarative.Types

type Parameters = Vector Double
type Gradient   = Parameters -> Parameters
type Particle   = (Parameters, Parameters)

-- | The NUTS transition kernel.
nuts :: PrimMonad m => Transition m Double
nuts = do
    Chain t target _ store <- get
    r0          <- V.replicateM (V.length t) (lift $ normal 0 1)
    z0          <- lift $ exponential 1
    let logu     = log (auxilliaryTarget lTarget t r0) - z0
        lTarget  = logObjective target
        glTarget = handleGradient $ gradient target
        e        = getStepSize Nothing store

    let go (tn, tp, rn, rp, tm, j, n, s)
          | s == 1 = do
              vj <- lift $ categorical [-1, 1]
              z  <- lift unit

              (tnn, rnn, tpp, rpp, t1, n1, s1) <-
                if   vj == -1
                then do
                  (tnn', rnn', _, _, t1', n1', s1') <-
                    buildTree lTarget glTarget tn rn logu vj j e
                  return (tnn', rnn', tp, rp, t1', n1', s1')
                else do
                  (_, _, tpp', rpp', t1', n1', s1') <-
```

```
                  buildTree lTarget glTarget tp rp logu vj j e
                return (tn, rn, tpp', rpp', t1', n1', s1')

            let accept = s1 == 1 && (min 1 (fi n1 / fi n :: Double)) > z

                n2 = n + n1
                s2 = s1 * stopCriterion tnn tpp rnn rpp
                j1 = succ j
                t2 | accept    = t1
                   | otherwise = tm

            go (tnn, tpp, rnn, rpp, t2, j1, n2, s2)

        | otherwise = do
            put $ Chain tm target (lTarget tm) (updateStepSize e store)
            return tm

    go (t, t, r0, r0, t, 0, 1, 1)

getStepSize :: Maybe Double -> OptionalStore -> Double
getStepSize (Just e) _     = e
getStepSize Nothing store = e where
  (ODouble e) = HashMap.lookupDefault (ODouble 0.1) NUTS store

updateStepSize :: Double -> OptionalStore -> OptionalStore
updateStepSize e = HashMap.insert NUTS (ODouble e)

buildTree lTarget glTarget t r logu v 0 e = do
  let (t0, r0) = leapfrog glTarget (t, r) (v * e)
      joint    = log $ auxilliaryTarget lTarget t0 r0
      n        = indicate (logu < joint)
      s        = indicate (logu - 1000 < joint)
```

```
  return (t0, r0, t0, r0, t0, n, s)

buildTree lTarget glTarget t r logu v j e = do
  z <- lift unit
  (tn, rn, tp, rp, t0, n0, s0) <-
    buildTree lTarget glTarget t r logu v (pred j) e

  if   s0 == 1
  then do
    (tnn, rnn, tpp, rpp, t1, n1, s1) <-
      if   v == -1
      then do
        (tnn', rnn', _, _, t1', n1', s1') <-
          buildTree lTarget glTarget tn rn logu v (pred j) e
        return (tnn', rnn', tp, rp, t1', n1', s1')
      else do
        (_, _, tpp', rpp', t1', n1', s1') <-
          buildTree lTarget glTarget tp rp logu v (pred j) e
        return (tn, rn, tpp', rpp', t1', n1', s1')

    let accept = (fi n1 / max (fi (n0 + n1)) 1) > (z :: Double)
        n2     = n0 + n1
        s2     = s0 * s1 * stopCriterion tnn tpp rnn rpp
        t2     | accept    = t1
               | otherwise = t0

    return (tnn, rnn, tpp, rpp, t2, n2, s2)
  else return (tn, rn, tp, rp, t0, n0, s0)

-- | Determine whether or not to stop doubling the tree of candidate states.
stopCriterion :: (Integral a, Num b, Ord b, Unbox b)
  => Vector b -> Vector b -> Vector b -> Vector b -> a
```

```
stopCriterion tn tp rn rp =
      indicate (positionDifference `innerProduct` rn >= 0)
    * indicate (positionDifference `innerProduct` rp >= 0)
  where
    positionDifference = tp .- tn

-- | Simulate a single step of Hamiltonian dynamics.
leapfrog :: Gradient -> Particle -> Double -> Particle
leapfrog glTarget (t, r) e = (tf, rf) where
  rm = adjustMomentum glTarget e t r
  tf = adjustPosition e rm t
  rf = adjustMomentum glTarget e tf rm

-- | Adjust momentum.
adjustMomentum :: (Fractional c, Unbox c)
  => (t -> Vector c) -> c -> t -> Vector c -> Vector c
adjustMomentum glTarget e t r = r .+ ((e / 2) .* glTarget t)

-- | Adjust position.
adjustPosition :: (Num c, Unbox c) => c -> Vector c -> Vector c -> Vector c
adjustPosition e r t = t .+ (e .* r)

-- | The MH acceptance ratio for a given proposal.
acceptanceRatio :: (Floating a, Unbox a)
  => (t -> a) -> t -> t -> Vector a -> Vector a -> a
acceptanceRatio lTarget t0 t1 r0 r1 = auxilliaryTarget lTarget t1 r1
                                    / auxilliaryTarget lTarget t0 r0

-- | The negative potential.
auxilliaryTarget :: (Floating a, Unbox a) => (t -> a) -> t -> Vector a -> a
auxilliaryTarget lTarget t r = exp (lTarget t - 0.5 * innerProduct r r)
```

```haskell
-- | Simple inner product.
innerProduct :: (Num a, Unbox a) => Vector a -> Vector a -> a
innerProduct xs ys = V.sum $ V.zipWith (*) xs ys

-- | Vectorized multiplication.
(.*) :: (Num a, Unbox a) => a -> Vector a -> Vector a
z .* xs = V.map (* z) xs

-- | Vectorized subtraction.
(.-) :: (Num a, Unbox a) => Vector a -> Vector a -> Vector a
xs .- ys = V.zipWith (-) xs ys

-- | Vectorized addition.
(.+) :: (Num a, Unbox a) => Vector a -> Vector a -> Vector a
xs .+ ys = V.zipWith (+) xs ys

-- | Indicator function.
indicate :: Integral a => Bool -> a
indicate True  = 1
indicate False = 0

-- | Alias for fromIntegral.
fi :: (Integral a, Num b) => a -> b
fi = fromIntegral
```

Bibliography

Charalambos Aliprantis and Kim Border. *Infinite Dimensional Analysis*. Springer-Verlag, Berlin, 2006.

Gene Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20 1967 Spring ACM Computer Conference*, pages 483–485, 1967.

Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle Markov Chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.

Heinrich Apfelmus. operational: Difficult monads made easy with operational semantics. https://hackage.haskell.org/package/operational, 2010.

David Applebaum. *Lévy processes and stochastic calculus*. Cambridge University Press, 2009.

Tom Avery. Codensity and the Giry monad. *Journal of Pure and Applied Algebra*, 220(3):1229–1251, 2016.

Steve Awodey. *Category Theory*. Oxford University Press, 2010.

Nils Bertschinger. Embedded probabilistic programming in Clojure. In *Proceedings of the 5th European Lisp Symposium*, 2012.

Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. Measure transformer semantics for Bayesian machine learning. In *Programming Languages and Systems*, pages 77–96. Springer, 2011.

Paolo Capriotti and Ambrus Kaposi. Free applicative functors. *arXiv preprint arXiv:1403.0749*, 2014.

Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, 2008.

Martin Erwig and Steve Kollmansberger. Functional Pearls: Probabilistic functional programming in Haskell. *Journal of Functional Programming*, 16(01): 21–34, 2006.

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

Charles Geyer. Markov Chain Monte Carlo lecture notes. `http://www.stat.umn.edu/geyer/f05/8931/n1998.pdf`, 2005.

Andy Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 117–128. ACM, 2009.

Andy Gill. Domain-specific languages and code synthesis using Haskell. *Queue*, 12(4):30, 2014.

Mark Girolami and Ben Calderhead. Riemann manifold Langevin and Hamiltonian Monte Carlo methods (with discussion). *J.R. Statist. Soc. B*, 73:123–214, 2011.

Michèle Giry. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, volume 915, pages 68–85, 1981.

Gabriel Gonzalez. foldl: Composable, streaming, and efficient left folds. https://hackage.haskell.org/package/foldl, 2013.

Jonathan Goodman and Jonathan Weare. Ensemble samplers with affine invariance. *Communications in Applied Mathematics and Computational Science*, 5 (1):65–80, 2010.

Noah D. Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org, 2014.

Noah D. Goodman, Vikash Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua Tenenbaum. Church: a language for generative models. In *Proceedings of Uncertainty in Artificial Intelligence*, 2008.

Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques - Second Edition*. Springer US, 2008.

Hakaru. Hakaru: A probabilistic programming embedded DSL. https://hakaru-dev.github.io, 2014.

W. Keith Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

Matthew D. Hoffman and Andrew Gelman. The No U-Turn Sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *arXiv preprint arXiv:1111.4246*, 2011.

David B. Kirk and Wen-mei Hwu. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

Oleg Kiselyov. Problems of the lightweight implementation of probabilistic programming. In *Proceedings of Workshop on Probabilistic Programming Semantics*, 2016.

Oleg Kiselyov and Chung-chieh Shan. Embedded probabilistic programming. In *Advances in Neural Information Processing Systems (NIPS) 2008*, 2008.

Edward Kmett. free: Monads for free. https://hackage.haskell.org/package/free, 2008a.

Edward Kmett. kan-extensions: Kan extensions, Kan lifts, various forms of the Yoneda lemma, and (co)density (co)monads. https://hackage.haskell.org/package/kan-extensions, 2008b.

Edward Kmett. ad: Automatic Differentiation in Haskell. https://hackage.haskell.org/package/ad, 2010.

Lindsey Kuper, Aaron Turon, Neelakantan Krishnaswami, and Ryan Newton. Freeze after writing. In *POPL '14*, 2014.

F. William Lawvere. The category of probabilistic mappings. https://github.com/mattearnshaw/lawvere/blob/master/pdfs/1962-the-category-of-probabilistic-mappings.pdf, 1962.

Tom Leinster. Codensity and the ultrafilter monad. *Theory and Applications of Categories*, 28(13):332–370, 2013.

Lei Li and Stuart J. Russell. The BLOG language reference. Technical report, Technical Report UCB/EECS-2013-51, EECS Department, University of California, Berkeley, 2013.

Miran Lipovača. *Learn You A Haskell for Great Good!* No Starch Press, 2011.

Andres Löh. Haskell for (E)DSLs. http://www.andres-loeh.de/HaskellForDSLs.pdf, 2012.

David J. Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. WinBUGS - a Bayesian modelling framework: Concepts, structure, and extensibility. *Statistics and Computing*, 10:325–337, 2000.

Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971.

Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled GPU functions in Haskell. *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell*, 45(11):67–78, 2010.

Vikash Mansinghka. *Natively Probabilistic Computation*. PhD thesis, MIT, 2009.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

Simon Marlow. *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming*. O'Reilly Media, 2013.

Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell*, 46(12):71–82, 2011.

Simon Marlow (editor). Haskell 2010 language report. https://www.haskell.org/onlinereport/haskell2010, 2010.

George Marsaglia and Arif Zaman. A new class of random number generators. *The Annals of Applied Probability*, pages 462–480, 1991.

George Marsaglia et al. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2008.

Andrew McCallum, Karl Schultz, and Sameer Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems*, pages 1249–1257, 2009.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

Praveen Narayanan and Chung-chieh Shan. A combinator library for MCMC sampling. In *3rd NIPS Workshop on Probabilistic Programming*, 2014.

Radford Neal. Sampling from multimodal distributions using tempered transitions. *Statistics and computing*, 6(4):353–366, 1996.

Radford Neal. Slice Sampling. *The Annals of Statistics*, 31(3):705–767, 2003.

Radford Neal. MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*, 2011.

Yurii Nesterov. Primal-dual subgradient methods for convex problems. *Mathematical programming*, 120(1):221–259, 2009.

Bruno C. d. S. Oliveira and Andres Löh. Abstract syntax graphs for domain specific languages. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*, pages 87–96. ACM, 2013.

Bryan O'Sullivan. mwc-random: Fast, high quality pseudo random number generation. https://hackage.haskell.org/package/mwc-random, 2009.

Prakash Panangaden. The category of Markov kernels. *Electronic Notes in Theoretical Computer Science*, 22:171–187, 1999.

Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based upon sampling functions. *ACM Transactions on Programming Languages and Systems*, 45(1):171–182, 2008.

Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, volume 23, pages 199–208. ACM, 1988.

Benjamin C. Pierce. *Types and Programming Languages*. MIT press, 2002.

Dan Piponi. Free, but not as in beer or speech. https://drive.google.com/file/d/0B51SFgxqMDS-NDBOX0ZDdW52dEE, 2014.

Martyn Plummer. coda: Output Analysis and Diagnostics for MCMC. https://cran.r-project.org/web/packages/coda/coda.pdf, 2015a.

Martyn Plummer. coda: Output Analysis and Diagnostics for MCMC. https://cran.r-project.org/web/packages/coda/coda.pdf, 2015b.

David Pollard. *A User's Guide to Measure Theoretic Probability*. Cambridge University Press, 2001.

Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *29th Annual ACM Symposium on Principles of Programming Languages (POPL '02)*, 2002.

G. Rodrigues. Categorifying measure theory: a roadmap. *arXiv preprint arXiv:0912.4914*, 2009.

P. Ruckdeschel, M. Kohl, T. Stabla, and F. Camphausen. S4 Classes for Distributions. *R News*, 6(2):2–6, May 2006. URL http://www.uni-bayreuth.de/departments/math/org/mathe7/DISTR/distr.pdf.

John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science*, 2:e55, 2016.

Taisuke Sato. Generative modeling by PRISM. In *International Conference on Logic Programming*, pages 24–35. Springer, 2009.

Adam Ścibior, Zoubin Ghahramani, and Andrew D. Gordon. Practical probabilistic programming with monads. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, pages 165–176. ACM, 2015.

Adam Ścibior, Yufei Cai, Klaus Ostermann, and Zoubin Ghahramani. Building inference algorithms from monad transformers. http://mlg.eng.cam.ac.uk/adam/pps2017.pdf, 2017.

Stan. Stan: A C++ Library for Probability and Sampling, version 2.1. http://mc-stan.org, 2013.

Kirk Sturtz. Categorical probability theory. *arXiv preprint arXiv:1406.6030*, 2014.

Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18 (04):423–436, 2008.

Hidetosi Takahasi and Masatake Mori. Double exponential formulas for numerical integration. *Publications of the Research Institute for Mathematical Sciences*, 9(3):721–741, 1974.

Jared Tobin. flat-mcmc: Painless general-purpose sampling. http://github.com/jtobin/flat-mcmc, 2012.

Jared Tobin. declarative: A library for building composable sampling strategies. http://github.com/jtobin/declarative, 2013a.

Jared Tobin. measurable: An embedded DSL for creating, composing, and using probability measures. http://github.com/jtobin/measurable, 2013b.

Jared Tobin. observable: An embedded language for probabilistic programming. http://github.com/jtobin/observable, 2013c.

Jared Tobin. mwc-probability: Sampling function-based probability distributions. http://github.com/jtobin/mwc-probability, 2014.

Jared Tobin. Practical Recursion Schemes. https://medium.com/@jaredtobin/practical-recursion-schemes-c10648ec1c29, 2015.

Jared Tobin. deanie: An embedded probabilistic programming language. http://github.com/jtobin/deanie, 2017.

Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016.

Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.

Tarmo Uustalu and Varmo Vene. Comonadic notions of computation. *Electronic Notes in Theoretical Computer Science*, 203(5):263–284, 2008.

Janis Voigtländer. Asymptotic improvement of computations over free monads. In *International Conference on Mathematics of Program Construction*, pages 388–403. Springer, 2008.

Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

Philip Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–55, 1994.

Wikipedia. Test functions for optimization. https://en.wikipedia.org/wiki/Test_functions_for_optimization, 2015.

David Williams. *Probability With Martingales*. Cambridge University Press, 1991.

David Wingate, Andreas Stuhlmuller, and Noah D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *AISTATS*, 2011.

Frank Wood, Jan Willem Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

Robert Zinkov. Why Probabilistic Programming Matters. http://zinkov.com/posts/2012-06-27-why-prob-programming-matters/, 2012.

Robert Zinkov and Chung-chieh Shan. Composing inference algorithms as program transformations. *arXiv preprint arXiv:1603.01882*, 2016.