Symbolic Bayesian Inference by Lazy Partial Evaluation

Chung-chieh Shan
Indiana University
ccshan@indiana.edu

Norman Ramsey
Tufts University
nr@cs.tufts.edu

Abstract

Bayesian inference, of posterior knowledge based on prior knowledge and observed evidence, is typically implemented by applying Bayes's theorem, solving an equation in which the posterior multiplied by the probability of an observation equals a joint probability. But when we observe a value of a continuous variable, the observation usually has probability zero, and Bayes's theorem says only that zero times the unknown is zero. To infer a posterior distribution from a zero-probability observation, we turn to the statistical technique of *disintegration*. The classic formulation of disintegration tells us only what constitutes a posterior distribution, not how to compute it. But by representing all distributions and observations as terms of our probabilistic language, core Hakaru, we have developed the first constructive method of computing disintegrations, solving the problem of drawing inferences from zero-probability observations. Our method uses a lazy partial evaluator to transform terms of core Hakaru, and we argue its correctness by a semantics of core Hakaru in which monadic terms denote measures. The method, which has been implemented in a larger system, is useful not only on its own but also in composition with sampling and other inference methods commonly used in machine learning.

1. Introduction

A core technique used in reasoning about uncertainty is *probabilistic inference*. In the Bayesian approach, one begins with a probability distribution representing *prior* knowledge or belief about the world. Then, given an *observation* of the world, one uses *inference* to compute or approximate a *posterior* probability distribution that represents one's new beliefs in light of the evidence. In applications, inference algorithms are often specialized to particular distributions; to make such algorithms easier to reuse, many researchers have embedded them in *probabilistic programming languages*.

A probabilistic programming language addresses the inference problem in two stages. First, the prior distribution, which is different in every application, is specified by a *generative model*. The model looks like a program, and it is called *generative* because it is written as if running it generated the state of the world, by making random choices. In practice, generative models are not run; they are used only to draw inferences. Second, the generative model is given to an inference algorithm which, like an interpreter or compiler, is reused across applications.

This paper contributes a new algorithm that computes a posterior distribution by symbolic manipulation of the prior distribution and an observation. Our *automatic disintegrator* is based on the statistical technique of *disintegration* (Chang and Pollard 1997). Automatic disintegration solves a longstanding problem in probabilistic programming: in a way that is firmly grounded in the mathematics of probability, it can draw inferences from observations of zero-probability events. Such observations are common in practice; for example, they include any observation of the *value* of a variable described by a continuous distribution, such as a uniform or normal distribution. In addition, our disintegrator has a number of other desirable properties:

- Not only the prior and the observation, but also the *posterior*, are represented as terms in our probabilistic language, core Hakaru (pronounced Hah-KAH-roo). The disintegrator is a purely syntactic transformation which combines two terms—one that represents the prior and one that describes the quantity to be observed—into one open term that represents a function from observed value to posterior distribution.
- The semantics of core Hakaru is stated in terms of the measure monad (an extension of the probability monad). Our disintegrator preserves this semantics.
- A term of core Hakaru also has a semantics as an *importance sampler*, so in full Hakaru we can combine disintegration with sampling-based inference techniques as well as exact ones.

To use our disintegrator, a programmer typically represents a prior as a sequence of bindings, each of which names a probabilistic choice or a deterministic computation. The prior is then extended with a quantity to be observed, which is written as an expression in terms of the bound variables. The disintegrator transforms the extended prior so that the observed quantity is bound *first*. From the syntactic form of the transformed extended prior, we "read off" a function from observation to posterior.

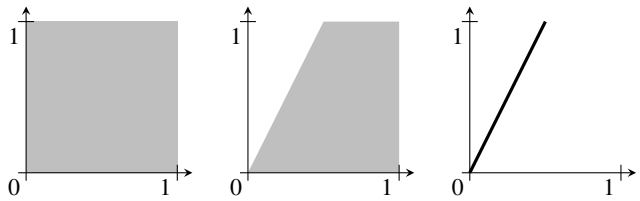
Like a partial evaluator, the disintegrator produces a residual program. Like a lazy evaluator, it puts bindings on a heap and orders their evaluation as determined by demand, not by source code. As needed to preserve semantics, it rewrites bindings' right-hand sides, which incur effects in the measure monad. This rewriting requires just enough computer algebra to handle changes of variables in simple integrals. The simplicity of the computer algebra does limit the system, but in experiments with a variety of distributions expressed in the full Hakaru language, the disintegrator works—both by itself and in conjunction with other sophisticated inference techniques, including symbolic simplification, Metropolis-Hastings sampling, and Gibbs sampling.

2. The idea of our contribution

To explain our contribution, it's best to postpone the semantics of core Hakaru and the workings of our disintegration algorithm. We begin instead with the problem: we introduce observation and inference, we show the difficulties created by observation of a zero-probability event, we show how disintegration addresses the difficulty, and we introduce core Hakaru by example.

2.1 Observation, inference, and query in core Hakaru

Probabilistic programmers don't only infer posterior distributions; we also ask questions about distributions. A question, often called a "query," can be posed either to a prior or to a posterior distribution; popular queries include asking for the expected value of some variable or function, asking for the most likely outcome in the distribution, and asking for samples drawn from the distribution. In this



(a) Prior distribution m_{1a}: uniform distribution on the unit square — $E(x) = 1/2$

(b) Posterior distribution m_{1b} after observing $y \le 2 \cdot x$ — $E(x) = 11/18$

(c) Posterior distribution m_{1c} after observing $y = 2 \cdot x$ — $E(x) \stackrel{?}{=} 1/4$

Figure 1. Examples of observation and inference

section we study distributions over pairs (x,y) of real numbers: one prior distribution and two posterior distributions (Figure 1). To each distribution we pose the same query: the expected value of x, written *informally* as $E(x)$. This notation is widely used and can promote intuition, but for precision work, it's too informal—for example, it doesn't say with respect to what distribution we are taking the expectation. We write commonly used but informal notations in gray.

Part (a) of Figure 1 shows a prior distribution of pairs (x,y) distributed uniformly over the unit square. This distribution can be specified by the following generative model written in core Hakaru:

$$m_{1a} \equiv \textbf{do}\ \{x \leftsquigarrow \textbf{uniform}\ 0\ 1;\ y \leftsquigarrow \textbf{uniform}\ 0\ 1;\ \textbf{return}\ (x,y)\} \quad (1)$$

The notation is meant to evoke Haskell's **do** notation; the unit (**return**) and bind ($\leftsquigarrow$) operations are the standard operations of the monad of probability distributions (Giry 1982; Ramsey and Pfeffer 2002). The term **uniform** 0 1 denotes the uniform distribution of real numbers between 0 and 1.

Even before doing observation or inference, we can ask for the expected value of x under model m_{1a}. To make it crystal clear that we are asking for the expectation of the function $\lambda(x,y).x$ under the distribution denoted by model m_{1a}, we write not the informal $E(x)$ but the more precise $E_{m_{1a}}(\lambda(x,y).x)$. The expectation of a function f is the ratio of two integrals: the integral of f and the integral of the constant 1 function. Expectation is defined whenever the integral of the constant 1 function is finite and nonzero.

$$E_{m_{1a}}(\lambda(x,y).x) = \frac{\int_{[0,1]\times[0,1]} x\,d(x,y)}{\int_{[0,1]\times[0,1]} 1\,d(x,y)} = \frac{1/2}{1} = 1/2. \quad (2)$$

Part (b) of Figure 1 introduces observation and inference: if we observe[1] $y \le 2 \cdot x$, we must infer the posterior distribution represented by the shaded trapezoid in Figure 1(b). Again, the expected value of x is the ratio of two integrals: of the function $\lambda(x,y).x$ and the constant 1 function. To calculate the integrals, we split the trapezoid into a triangle and a rectangle:

$$\begin{aligned} E_{m_{1b}}(\lambda(x,y).x) &= \frac{\int_{\{(x,y)\in[0,1]\times[0,1]\mid y\le 2\cdot x\}} x\,d(x,y)}{\int_{\{(x,y)\in[0,1]\times[0,1]\mid y\le 2\cdot x\}} 1\,d(x,y)} \\ &= \frac{1/12+3/8}{1/4+1/2} = \frac{11/24}{3/4} = 11/18. \end{aligned} \quad (3)$$

[1] While we must use notation from integral calculus, we avoid its convention that juxtaposition means multiplication. As in programming languages, we write multiplication using an infix operator, but not the infix operator $\times$, which we reserve to refer to product types and product domains. To multiply numbers, we write an infix $\cdot$, as in $2 \cdot x$.

To represent the posterior distribution in core Hakaru, we add the observation to our generative model:

$$\begin{aligned} m_{1b} = \textbf{do}\ \{&x \leftsquigarrow \textbf{uniform}\ 0\ 1; \\ &y \leftsquigarrow \textbf{uniform}\ 0\ 1; \\ &\textbf{observe}\ y \le 2 \cdot x; \\ &\textbf{return}\ (x,y)\} \end{aligned} \quad (4)$$

The new line **observe** $y \le 2 \cdot x$ restricts the domain of (x,y) to where the predicate $y \le 2 \cdot x$ holds; the same predicate appears in (3).

So far, so good. But in Figure 1(c) this happy story falls apart. To help you understand what goes wrong, we draw an analogy between the diagrams, our calculations of expectations, and this classic equation for conditional probability:

$$\Pr(A \cap B) = \Pr(A) \cdot \Pr(B \mid A). \quad (5)$$

Here's the analogy:

- A and B are sets of points in the unit square. Set A represents an *observation*. In Figure 1(b), A is the set $\{(x,y) \mid y \le 2 \cdot x\}$.

  Set B represents a *query*; for example, we might ask for the probability that $x > 2/3$ or for the probability that the point (x,y) falls within the circle inscribed within the unit square. To keep the analogy simple, we ask only *probability* queries; the probability of a set B is the expectation of the set's characteristic function χ_B, which is 1 on points in B and 0 on points outside of B.
- $\Pr(\cdots)$ with no vertical bar represents a query against the prior distribution. For example in Figure 1(a), $\Pr(x > 2/3) = 1/3$, and $\Pr(A) = 3/4$. In particular, $\Pr(A \cap B)$ is just another query against the prior.
- $\Pr(B \mid A)$ represents a query against the posterior; it's the unknown we're trying to compute. To compute it, we solve (5); the solution gives $\Pr(B \mid A)$ in terms of queries against the prior. In Figure 1(b), the solution is the ratio of integrals in (3).

Now in Figure 1(c), we observe $y = 2 \cdot x$. Let's see what goes wrong. Set $A = \{(x,y) \mid y = 2 \cdot x\}$. A line has no area, so $\Pr(A) = 0$, and for any B, the probability $\Pr(A \cap B) = 0$. Equation (5) tells us only that $0 = 0 \cdot \Pr(B \mid A)$, so we can't solve for $\Pr(B \mid A)$. A precise calculation of the expectation of $\lambda(x,y).x$ is no better: in the ratio of integrals, both numerator and denominator are zero.

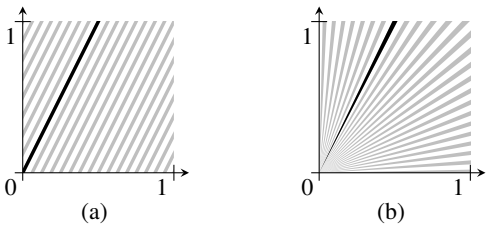
But the line segment in Figure 1(c) looks so reasonable! We started with a uniform distribution over the square, so shouldn't the posterior be a uniform distribution over that line segment? Isn't $E(x) = 1/4$? Not necessarily. We crafted Figure 1(c) from a paradox discussed by Bertrand (1889), Borel (1909), and Kolmogorov (1933), and there is no single expected value of x.

2.2 Observation of measure-zero sets is paradoxical

Why doesn't the line segment in Figure 1(c) determine the expected value of x? Aside from the obvious remark that $0/0$ isn't defined, can we say anything else? Yes. We can start with a tactic popular in probabilistic-programming circles: define the expected value of x as the result of a limiting process. The question is, *which* limiting process. There's more than one limit that converges to the line segment in Figure 1(c); two such limits are depicted in Figure 2.

In Figure 2(a), we imagine a limiting process in which we compute the expected value of x over a sequence of ever-narrower "strips" around $y = 2 \cdot x$:

$$\begin{aligned} E(x) &= \lim_{\varepsilon\to 0} \frac{\int_{\{(x,y)\in[0,1]\times[0,1]\mid y-2\cdot x\in[-\varepsilon,\varepsilon]\}} x\,d(x,y)}{\int_{\{(x,y)\in[0,1]\times[0,1]\mid y-2\cdot x\in[-\varepsilon,\varepsilon]\}} 1\,d(x,y)} \\ &= \lim_{\varepsilon\to 0} \frac{\varepsilon/4+\varepsilon^3/24}{\varepsilon-\varepsilon^2/4} = 1/4. \end{aligned} \quad (6)$$



Observing that $y - 2 \cdot x$ is 0. The center of mass of the dark strip is $(1/4, 1/2)$. $E(x) = 1/4$

Observing that y/x is 2. The center of mass of the dark wedge is $(1/3, 2/3)$. $E(x) = 1/3$

Figure 2. A Borel paradox: Two ways to observe $y = 2 \cdot x$

But in Figure 2(b), we compute the expected value of x over a sequence of ever-narrower "wedges" around $y = 2 \cdot x$:

$$\begin{aligned} E(x) &= \lim_{\varepsilon \to 0} \frac{\int_{\{(x,y) \in [0,1] \times [0,1] | y/x \in [2-\varepsilon, 2+\varepsilon]\}} x \, d(x,y)}{\int_{\{(x,y) \in [0,1] \times [0,1] | y/x \in [2-\varepsilon, 2+\varepsilon]\}} 1 \, d(x,y)} \\ &= \lim_{\varepsilon \to 0} \frac{(4/3) \cdot (\varepsilon/(4-\varepsilon^2)^2)}{\varepsilon/(4-\varepsilon^2)} = 1/3. \end{aligned} \tag{7}$$

Neither of these answers is "right"; the unpleasant truth is that we can't simply say **observe** $y = 2 \cdot x$ and hope it is unambiguous. We must say not only *what* zero-probability set we wish to observe, but also *how* to observe it. To say how, we don't calculate limits; instead we use an established technique from statistics: *disintegration* (Chang and Pollard 1997).

2.3 Resolving the paradox via disintegration

As uses of disintegration go, the example from Figures 1(c) and 2 is moderately sophisticated, because we observe the value of an expression that depends on two variables. To introduce disintegration, we step back and first discuss the general case. We then relate the general case to the two limits from Figure 2. Finally, we show disintegration in core Hakaru.

Disintegration decomposes measures on product spaces Disintegration may apply to any distribution ξ over a product of spaces α and β; we write $\xi \in \mathbb{M}(\alpha \times \beta)$, where $\mathbb{M}$ stands for *measures*. If we observe $a \in \alpha$, what is the posterior distribution over β? To answer this question, we decompose ξ into a measure $\mu \in \mathbb{M}\,\alpha$ and a *measure kernel* $\kappa \in (\alpha \to \mathbb{M}\,\beta)$, where the arrow stands for *measurable* functions. For example, when ξ is the uniform distribution over the unit square, one possible decomposition is for μ to be the uniform distribution over the unit interval and for κ to be the constant function that returns the uniform distribution over the unit interval.

A decomposition of ξ into μ and κ is called a *disintegration* of ξ; to reconstruct ξ from μ and κ, we write $\xi = \mu \otimes \kappa$. To a probabilist, the decomposition should look familiar: μ represents a distribution $\Pr(A)$, and κ represents the conditional probability $\Pr(B \,|\, A)$. Multiply them and you get the joint distribution (5).

Another way to think about disintegration is that the measure ξ is decomposed into a family of measures $\{\kappa(a) \mid a \in \alpha\}$ indexed by the observed outcome a. To infer a posterior distribution from the observation a, it's enough to apply κ to a, but to reconstruct ξ, we also need μ, which tells us how to weight the members of the family κ. The idea of a family may help illuminate the paradox in Figure 2: In Figure 2(a), we decompose the area measure into a family of measures along line segments each with slope 2, but at different y-intercepts. In Figure 2(b), we decompose the area measure into a family of measures along wedges each with origin $(0,0)$, but at different slopes.

Disintegrating our paradoxical example To use disintegration to compute a posterior distribution corresponding to Figure 1(c), we change the coordinate system so that the observation $y = 2 \cdot x$ is expressed as an observation of a single variable. (A change of variables requires more work than a Hakaru programmer needs to do, but the technique is familiar and helps illustrate the ideas.) Different coordinate systems lead to different answers.

For example, we can rotate the coordinate system, changing (x,y) coordinates to (t,u) coordinates:

$$t = y - 2 \cdot x \qquad\qquad u = 2 \cdot y + x \tag{8}$$

If we disintegrate the measure over (t,u) and apply the resulting measure kernel to the value 0 for t, we get back a measure that tells us $E(x) = 1/4$, as in Figure 2(a).

Or we can change rectangular (x,y) coordinates to polar (θ, r) coordinates:

$$\theta = \arctan \frac{y}{x} \qquad\qquad r = \sqrt{x^2 + y^2} \tag{9}$$

If we disintegrate the measure over (θ, r) and apply the resulting measure kernel to the value $\arctan 2$ for θ, we get back a measure that tells us $E(x) = 1/3$, as in Figure 2(b).

2.4 Using disintegration in core Hakaru

It is tedious to change coordinate systems and transform a measure to the new coordinates. Our new program transformation eliminates the tedium. In core Hakaru, all you do is write the *one* variable you wish to observe, using the variables you already have. You then extend your existing model by adding the new variable on the left. For example, you don't *change* (x,y) to (t,u); instead you keep (x,y) and *extend* it with t, winding up with $(t,(x,y))$.

In general, the new, extended model denotes a distribution over a pair space $\alpha \times \beta$ where α describes the variable you want to observe (so typically $\alpha = \mathbb{R}$) and β has whatever your original model had. Here's what the extended model, with $(t,(x,y))$, looks like for Figure 2(a):

$$\begin{aligned} \overleftarrow{m}_{2a} \equiv \mathbf{do}\ \{ & x \leftsquigarrow \mathbf{uniform}\ 0\ 1; \\ & y \leftsquigarrow \mathbf{uniform}\ 0\ 1; \\ & \mathbf{let}\ t = y - 2 \cdot x; \\ & \mathbf{return}\ (t,(x,y))\} \end{aligned} \tag{10}$$

For a model in this form, we use the name $\overleftarrow{m}$ (pronounced "right-to-left m"); the arrow points from right to left because the model **return**s a pair in which t, on the left, depends on (x,y), on the right. Our automatic disintegrator converts this term to an equivalent term in this form:

$$\overrightarrow{m}_{2a} \equiv \mathbf{do}\ \{t \leftsquigarrow m';\ p \leftsquigarrow k';\ \mathbf{return}\ (t,p)\} \tag{11}$$

The subterms m' and k' are found by the disintegrator. Crucially, m' is independent of the variables of the model; no variable appears free in it, and it denotes a measure on t. By contrast, k' typically depends on t, and it denotes a function from t to a measure—that is, a kernel. Hence for a model in this form, we use the name $\overrightarrow{m}$ (pronounced "left-to-right m"); the arrow points from left to right because the model **return**s a pair in which p, on the right, depends on t, on the left.

The disintegrator guarantees that $\overrightarrow{m}_{2a}$ is equivalent to $\overleftarrow{m}_{2a}$, which is to say that the measure denoted by m' and the kernel denoted by k' together disintegrate the measure denoted by $\overleftarrow{m}_{2a}$. To observe $t = 0$, we substitute 0 for t in k' and drop t from the **return**, yielding the posterior $m_{2a} \equiv k'\{t \mapsto 0\}$.

Let's see how this story plays out in the examples from Figure 2.

Disintegrating our example using the rotated coordinate t For Figure 2(a), our automatic disintegrator finds this equivalent prior:

$$\begin{aligned}\overleftarrow{m}_{2\mathrm{a}} \equiv \mathbf{do}\ \{t \leftsquigarrow \mathbf{lebesgue};\\ p \leftsquigarrow \boxed{\begin{array}{l}\mathbf{do}\ \{x \leftsquigarrow \mathbf{uniform}\ 0\ 1;\\ \quad \mathbf{observe}\ 0 \le t + 2 \cdot x \le 1;\\ \quad \mathbf{return}\ (x, t + 2 \cdot x)\}\end{array}};\\ \mathbf{return}\ (t,p)\}\end{aligned} \tag{12}$$

In this model, $m' \equiv \mathbf{lebesgue}$ is the Lebesgue measure on the real line, and k' is the boxed term in equation (12). To get the posterior, we substitute 0 for t in k', drop t, and simplify:

$$\begin{aligned}m'_{2\mathrm{a}} \equiv \mathbf{do}\ \{&x \leftsquigarrow \mathbf{uniform}\ 0\ 1;\\ &\mathbf{observe}\ 0 \le 2 \cdot x \le 1;\\ &\mathbf{return}\ (x, 2 \cdot x)\}\end{aligned} \tag{13}$$

Unlike the predicate $y = 2 \cdot x$, the predicate $0 \le 2 \cdot x \le 1$ is useful to **observe**, because it describes a set of nonzero measure. Therefore, when we use the posterior to integrate $\lambda(x,y).x$ and the constant 1 function, we calculate the expectation $E_{m_{2\mathrm{a}}}(\lambda(x,y).x) = 1/4$ as a ratio of nonzero numbers. This answer is consistent with geometric reasoning about "strips"; the intersection of the unit square and an infinitesimally thin strip around the line $y = 2 \cdot x$ is a region whose midpoint is $(1/4, 1/2)$.

Disintegrating our example using the polar coordinate θ For Figure 2(b), we extend the prior using only the ratio y/x, not its arc tangent, for simplicity.

$$\begin{aligned}\overleftarrow{m}_{2\mathrm{b}} \equiv \mathbf{do}\ \{&x \leftsquigarrow \mathbf{uniform}\ 0\ 1;\\ &y \leftsquigarrow \mathbf{uniform}\ 0\ 1;\\ &\mathbf{let}\ s = y/x;\\ &\mathbf{return}\ (s, (x,y))\}\end{aligned} \tag{14}$$

The automatic disintegrator emits the equivalent term

$$\begin{aligned}\overleftarrow{m}_{2\mathrm{b}} \equiv \mathbf{do}\ \{s \leftsquigarrow \mathbf{lebesgue};\\ p \leftsquigarrow \boxed{\begin{array}{l}\mathbf{do}\ \{x \leftsquigarrow \mathbf{uniform}\ 0\ 1;\\ \quad \mathbf{observe}\ 0 \le s \cdot x \le 1;\\ \quad \mathbf{factor}\ x;\\ \quad \mathbf{return}\ (x, s \cdot x)\}\end{array}};\\ \mathbf{return}\ (s,p)\}\end{aligned} \tag{15}$$

The **factor** x in the boxed term k' weights the probability distribution by x; it makes the probability mass proportional to x. Think of **factor** as multiplying by a probability density; in Figure 2(b), as we move away from the origin the wedges become thicker, and the thickness is proportional to x. To infer a posterior from the observation that y/x is 2, we substitute 2 for s in k' and drop s, yielding the posterior distribution

$$\begin{aligned}m_{2\mathrm{b}} \equiv \mathbf{do}\ \{&x \leftsquigarrow \mathbf{uniform}\ 0\ 1;\\ &\mathbf{observe}\ 0 \le 2 \cdot x \le 1;\\ &\mathbf{factor}\ x;\\ &\mathbf{return}\ (x, 2 \cdot x)\}.\end{aligned} \tag{16}$$

Using this posterior to integrate $\lambda(x,y).x$, we calculate the expectation $E_{m_{2\mathrm{b}}}(\lambda(x,y).x) = 1/3$. This answer is consistent with geometric reasoning about wedges; the intersection of the unit square and an infinitesimally thin wedge around the line $y = 2 \cdot x$ is a region whose midpoint is $(1/3, 2/3)$.

2.5 From examples to algorithm

From these examples we can work up to our disintegration algorithm. The algorithm preserves the semantics of terms, so before presenting the algorithm, we describe first the semantic foundations and then the syntax and semantics of core Hakaru.

3. Foundations

Our contributions rest on a foundation of measure theory, integration, and real analysis. We review just enough to make the semantics of core Hakaru understandable.

- We begin with *measure*, and in particular, a *measure function*. "Measure" generalizes "probability," so it is closest to the intuition of probabilistic programmers.
- We move to *integrators*. An integrator is a higher-order function that returns the integral of a function. Integrators can be used to compute expectation; the expectation of a function f is the ratio of applying an integrator to f and to the constant 1 function.

  An integrator may feel more powerful than a measure function, and perhaps less familiar, but they are equivalent (Section 3.2). We define disintegration using integrators, and in the following sections, we explain core Hakaru and our disintegration algorithm in terms of integrators, not measure functions.
- We discuss the properties of disintegrations that justify using **lebesgue** to disintegrate every extended model that has a continuously distributed real observable.

3.1 Measures and measure functions

A *measure function* maps sets to nonnegative real numbers. In standard terminology, a measure function is called just a *measure*, and the result of applying a measure function to a set S is called the *measure* of S. To explain how measures are constructed and how they are used in probability and inference, we pretend for this section that a measure and a measure function are not the same thing—please think of "measure" as an abstract datatype, and think of "measure function" as a possible implementation.

When a set S represents an observation, the measure of S is proportional to the probability of that observation. Here are two examples:

- In core Hakaru, **uniform** 0 1 denotes the *uniform probability measure* on the interval $[0,1]$. The corresponding measure function, which we'll call $[\![\mathbf{uniform}\ 0\ 1]\!]_M$, assigns a measure to many sets, including every interval on the real line. Under $[\![\mathbf{uniform}\ 0\ 1]\!]_M$, the measure of any interval is the length of the part of the interval that intersects with $[0,1]$. For example, $[\![\mathbf{uniform}\ 0\ 1]\!]_M[2/3, 2] = 1/3$, and indeed, if we choose a real variable x uniformly between 0 and 1, the probability of it lying between $2/3$ and 2 is $1/3$.
- In **uniform** 0 1 the measure, or the probability mass, is spread out evenly over an interval. But it is also possible to concentrate probability mass at a single point. In core Hakaru, **return** a denotes the *Dirac measure at a*: the measure of a set S is 1 if S contains a and 0 otherwise. Any discrete distribution can be represented as a countable linear combination of Dirac measures.

Measurable spaces and measurable sets We've been coy about what sets $[\![\mathbf{uniform}\ 0\ 1]\!]_M$ can be applied to. When we ask about the measure of a set S, or the probability of a point landing in set S, we are asking "how big is set S?" If we could ask the question about any set of real numbers, life would be grand. But the real numbers refuse to cooperate. To explain the issue, and to lay the foundation for the way we address it in core Hakaru, we recapitulate some of the development of the Lebesgue measure.

The nineteenth-century mathematicians who wanted to know how big a set was were hoping for a definition with four properties (Royden 1988, Chapter 3):

1. The size of an interval should be its length.
2. The size of a set should be invariant under translation.

3. The size of a union of disjoint sets should be the sum of the individual sizes.
4. Any subset of the real line should have a size.

Not all four properties can be satisfied simultaneously. But if we limit our attention to *measurable subsets* of the real line, we can establish the first three properties. The measurable subsets are the smallest collection of sets of reals that contains all the intervals and is closed under complement, countable intersection, and countable union. The one and only function on the measurable sets that has the first three properties is the *Lebesgue measure*. We write it as Λ.

The theory of measure functions, and the corresponding techniques of integration developed by Lebesgue, are not limited to real numbers: measure theory and abstract integration deal with *measurable spaces*. A measurable space α is a set $\alpha_{\mathcal{S}}$ together with a collection of its subsets $\alpha_{\mathcal{M}}$ that are deemed *measurable*. The collection $\alpha_{\mathcal{M}}$ must be closed under complement, countable intersection, and countable union. Such a collection $\alpha_{\mathcal{M}}$ is called a σ*-algebra* on the set $\alpha_{\mathcal{S}}$. Any collection M of subsets of $\alpha_{\mathcal{S}}$ *generates* a σ-algebra $\sigma(M)$ on $\alpha_{\mathcal{S}}$; $\sigma(M)$ is the smallest σ-algebra of which M is a subset.

Having defined measurable spaces, we can now explain measure functions more precisely. A *measure function* (or simply *measure*) μ on a measurable space α is a function from $\alpha_{\mathcal{M}}$ to $\mathbb{R}^+$. We write $\mathbb{R}^+$ to include the nonnegative real numbers as well as (positive) infinity, so in other words, μ must map each set in $\alpha_{\mathcal{S}}$ either to a nonnegative real number or to ∞. Moreover, it must be *countably additive*, which means that for any countable collection of pairwise-disjoint measurable sets $\{S_1, S_2, \dots\} \subseteq \alpha_{\mathcal{M}}$,

$$\mu(S_1 \cup S_2 \cup \cdots) = \mu(S_1) + \mu(S_2) + \cdots. \tag{17}$$

For notational convenience, we typically write α not only to stand for a measurable space but also to stand for its underlying set $\alpha_{\mathcal{S}}$. This convention is less confusing than it might be, because although probabilistic programs use many different measures, they use relatively few measurable spaces. These measurable spaces correspond with base types and type constructors used in many programming languages. We defer the details to Section 4, which explains the syntax and semantics of core Hakaru.

3.2 Integrators

A measure is often identified with its measure function, but a measure can do more than just measure sets; it can also integrate functions. The connection between measure and integration is used by anyone who represents a probability measure as a probability-density function. To see how it works, we *define* the integral of a function f with respect to a measure μ.

What functions can be integrated? A function f from one measurable space α to another β is *measurable* iff the inverse image of every measurable set is a measurable set. Crucially, when β is $\mathbb{R}^+$ (whose measurable subsets $\mathbb{R}^+_{\mathcal{M}}$ are the σ-algebra generated by the intervals), the measurable functions form a complete partial order: $f \sqsubseteq g$ iff for all $a \in \alpha$, $f(a) \le g(a)$. Any measurable function f from α to $\mathbb{R}^+$ can be integrated with respect to any measure μ on α. You can find the details in Royden (1988, Chapters 4 and 11); here we review only the essentials.

The integral of a function f with respect to a measure μ is often written $\int f\, d\mu$, or if it is convenient to introduce a bound variable x, $\int f(x)\, d\mu(x)$. But these traditional notations are awkward to work with; reasoning about integrals is far easier if we treat the measure μ as an *integrator function* and write the integral simply as $\mu(f)$ (de Finetti 1974; Pollard 2001). When it is useful to distinguish μ as a measure function from μ as an integrator, we write the measure function as $[\![\mu]\!]_M$, the integrator as $[\![\mu]\!]_I$, and the integral of f as $[\![\mu]\!]_I(f)$. In this section, where we define $[\![\mu]\!]_I$ in terms of $[\![\mu]\!]_M$, we maintain this distinction.

We begin by defining the integral of the *characteristic function* χ_S of a measurable set S, where

$$\chi_S(a) = \begin{cases} 1 & \text{if } a \in S \\ 0 & \text{if } a \notin S \end{cases} \tag{18}$$

Because S is measurable, χ_S is also measurable, and its integral is

$$[\![\mu]\!]_I(\chi_S) = [\![\mu]\!]_M(S). \tag{19}$$

As always, integration is linear, so for any real r and measurable functions f and g,

$$[\![\mu]\!]_I(\lambda a.\, r \cdot f(a)) = r \cdot [\![\mu]\!]_I(f), \tag{20}$$

$$[\![\mu]\!]_I(\lambda a.\, f(a) + g(a)) = [\![\mu]\!]_I(f) + [\![\mu]\!]_I(g). \tag{21}$$

These three equations define the integral of every *simple function*, where a simple function is a linear combination of characteristic functions. To integrate the remaining measurable functions, we use limits; given a monotone sequence of functions $f_1 \sqsubseteq f_2 \sqsubseteq \cdots$, the integral of the limit is the limit of the integrals:

$$[\![\mu]\!]_I(\lambda a.\, \lim_{n\to\infty} f_n(a)) = \lim_{n\to\infty} [\![\mu]\!]_I(f_n) \tag{22}$$

Every measurable function can be approximated from below by a sequence of simple functions, and as in Scott's domain theory, equations (18) to (22) define $[\![\mu]\!]_I$ uniquely; given $[\![\mu]\!]_M$, there is one and only one function $[\![\mu]\!]_I$ satisfying these equations. Finally, integrators and measure functions are in one-to-one correspondence; to define $[\![\mu]\!]_M$ in terms of $[\![\mu]\!]_I$, simply use equation (19).

Many useful measures are most easily defined by specifying their corresponding integrators. For example, the uniform measure is defined by Lebesgue integration with respect to the Lebesgue measure, and the Dirac measure is defined by function application:

$$[\![\textbf{uniform}\ 0\ 1]\!]_I f = \int_{[0,1]} f(x)\, dx \tag{23}$$

$$[\![\textbf{return}\ a]\!]_I f = f(a) \tag{24}$$

As another example, we express disintegration using integrators. If $\xi = \mu \otimes \kappa$, where $\mu \in \mathbb{M}\,\alpha$ and $\kappa \in (\alpha \to \mathbb{M}\,\beta)$, then ξ is defined by an iterated integral:

$$[\![\xi]\!]_I f = [\![\mu]\!]_I(\lambda a.\, [\![\kappa\, a]\!]_I(\lambda b.\, f(a,b))). \tag{25}$$

Here the integrand f is an arbitrary measurable function from $\alpha \times \beta$ to $\mathbb{R}^+$. The same equation can be written using traditional integral notation, which is more familiar but is cumbersome to manipulate:

$$\int f\, d\xi = \iint f(a,b)\, d(\kappa(a))(b)\, d\mu(a). \tag{26}$$

3.3 Disintegrations and the Lebesgue measure

An extended prior defines a measure ξ over a product space $\alpha \times \beta$. A disintegration $\xi = \mu \otimes \kappa$ may not always exist, but when there is one, there are infinitely many. For example, we can obtain $\xi = \mu' \otimes \kappa'$ by scaling μ and κ. But the properties we are interested in, such as expectation or most likely outcome, are invariant under scaling, so this form of non-uniqueness doesn't matter. In fact, when $\alpha = \mathbb{R}$, we exploit the non-uniqueness to choose a particularly advantageous μ.

Given $\xi = \mu \otimes \kappa$, if μ is *absolutely continuous* with respect to the Lebesgue measure Λ, then the Radon-Nikodym theorem assures the existence of κ' such that $\xi = \Lambda \otimes \kappa'$, and any two such κ's are equal almost everywhere. Absolute continuity means just that μ assigns

Variables	x
Real numbers	$r \in \mathbb{R}$
Atomic terms	$u ::= x$ (not bound in the heap) $\mid -u \mid u^{-1} \mid u+u \mid u+r \mid r+u \mid u \cdot u \mid u \cdot r \mid r \cdot u \mid u \leq u \mid u \leq r \mid r \leq u \mid \mathbf{fst}\ u \mid \mathbf{snd}\ u$
Bindings (guards)	$g ::= x \leftsquigarrow e \mid \mathbf{let\ inl}\ x = e \mid \mathbf{let\ inr}\ x = e \mid \mathbf{factor}\ e$
Head normal forms	$v ::= u \mid \mathbf{do}\ \{g; e\} \mid \mathbf{return}\ e \mid \mathbf{fail} \mid \mathbf{mplus}\ e_1\ e_2 \mid r \mid () \mid (e_1, e_2) \mid \mathbf{inl}\ e \mid \mathbf{inr}\ e \mid \mathbf{lebesgue} \mid \mathbf{uniform}\ r_1\ r_2$
Terms	$e, m ::= v \mid x \mid -e \mid e^{-1} \mid e+e \mid e \cdot e \mid e \leq e \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e$

Figure 3. Syntactic forms of core Hakaru

$\boxed{\Gamma \vdash e : \alpha}$

$$\frac{\Gamma \vdash e : \mathbb{M}\,\alpha \quad \Gamma, x{:}\alpha \vdash e' : \mathbb{M}\,\beta}{\Gamma \vdash \mathbf{do}\ \{x \leftsquigarrow e; e'\} : \mathbb{M}\,\beta}$$

$$\frac{\Gamma \vdash e : \alpha + \beta \quad \Gamma, x{:}\alpha \vdash e' : \mathbb{M}\,\gamma}{\Gamma \vdash \mathbf{do}\ \{\mathbf{let\ inl}\ x = e; e'\} : \mathbb{M}\,\gamma} \qquad \frac{\Gamma \vdash e : \alpha + \beta \quad \Gamma, x{:}\beta \vdash e' : \mathbb{M}\,\gamma}{\Gamma \vdash \mathbf{do}\ \{\mathbf{let\ inr}\ x = e; e'\} : \mathbb{M}\,\gamma}$$

$$\frac{\Gamma \vdash e : \mathbb{R} \quad \Gamma \vdash e' : \mathbb{M}\,\alpha}{\Gamma \vdash \mathbf{do}\ \{\mathbf{factor}\ e; e'\} : \mathbb{M}\,\alpha} \qquad \frac{\Gamma \vdash e : \alpha}{\Gamma \vdash \mathbf{return}\ e : \mathbb{M}\,\alpha}$$

$$\frac{}{\Gamma \vdash \mathbf{fail} : \mathbb{M}\,\alpha} \qquad \frac{\Gamma \vdash e_1 : \mathbb{M}\,\alpha \quad \Gamma \vdash e_2 : \mathbb{M}\,\alpha}{\Gamma \vdash \mathbf{mplus}\ e_1\ e_2 : \mathbb{M}\,\alpha}$$

$$\frac{}{\Gamma \vdash \mathbf{lebesgue} : \mathbb{M}\,\mathbb{R}} \qquad \frac{r_1 < r_2}{\Gamma \vdash \mathbf{uniform}\ r_1\ r_2 : \mathbb{M}\,\mathbb{R}}$$

Figure 4. Typing rules for terms of measure type

$$\begin{aligned}
[\![\mathbf{do}\ \{x \leftsquigarrow e; e'\}]\!]_I \rho f &= [\![e]\!]_I \rho (\lambda a.\, [\![e']\!]_I (\rho\{x \mapsto a\}) f) \\
[\![\mathbf{do}\ \{\mathbf{let\ inl}\ x = e; e'\}]\!]_I \rho f &= [\![e']\!]_I (\rho\{x \mapsto a\}) f && \text{if } [\![e]\!]\rho = \mathbf{inl}\ a \\
[\![\mathbf{do}\ \{\mathbf{let\ inl}\ x = e; e'\}]\!]_I \rho f &= 0 && \text{if } [\![e]\!]\rho = \mathbf{inr}\ b \\
[\![\mathbf{do}\ \{\mathbf{let\ inr}\ x = e; e'\}]\!]_I \rho f &= [\![e']\!]_I (\rho\{x \mapsto a\}) f && \text{if } [\![e]\!]\rho = \mathbf{inr}\ a \\
[\![\mathbf{do}\ \{\mathbf{let\ inr}\ x = e; e'\}]\!]_I \rho f &= 0 && \text{if } [\![e]\!]\rho = \mathbf{inl}\ b \\
[\![\mathbf{do}\ \{\mathbf{factor}\ e; e'\}]\!]_I \rho f &= [\![e]\!]\rho \cdot [\![e']\!]_I \rho f && \text{if } [\![e]\!]\rho \geq 0 \\
[\![\mathbf{return}\ e]\!]_I \rho f &= f([\![e]\!]\rho) \\
[\![\mathbf{fail}]\!]_I \rho f &= 0 \\
[\![\mathbf{mplus}\ e_1\ e_2]\!]_I \rho f &= [\![e_1]\!]_I \rho f + [\![e_2]\!]_I \rho f \\
[\![\mathbf{lebesgue}]\!]_I \rho f &= \textstyle\int_{\mathbb{R}} f(x)\,dx \\
[\![\mathbf{uniform}\ r_1\ r_2]\!]_I \rho f &= \tfrac{1}{r_2 - r_1} \cdot \textstyle\int_{[r_1, r_2]} f(x)\,dx
\end{aligned}$$

Figure 5. Denotations of terms of measure type

zero measure to every set of Lebesgue measure zero; it is equivalent to saying that μ is described by a probability-density function. Because so many observation distributions μ over $\mathbb{R}$ are absolutely continuous with respect to Λ, our disintegrator simply *assumes* $\mu = \Lambda$ whenever $\alpha = \mathbb{R}$. That is why throughout Section 2.4, the automatically found m' is **lebesgue**. More general observation distributions are discussed briefly in Section 8.

4. Syntax, types, and semantics of core Hakaru

The syntax, typing rules, and semantics of core Hakaru are presented in Figures 3, 4, and 5. Core Hakaru has standard unit ($\mathbb{1}$), pair ($\times$), and sum ($+$) types; for reasoning about probability, it also has a real-number type ($\mathbb{R}$) and measure types ($\mathbb{M}$). We write types as $\alpha, \beta, \gamma, \ldots$, where

$$\alpha ::= \mathbb{1} \mid \alpha \times \beta \mid \alpha + \beta \mid \mathbb{R} \mid \mathbb{M}\,\alpha. \tag{27}$$

Types as measurable spaces Each type of core Hakaru corresponds to a measurable space. We recognize the distinction between a type (a means of classifying terms) and a measurable space (a semantic object), but notating the distinction adds a lot of ink and seems to make the presentation less clear. So in this paper we conflate types with spaces.

The unit space $\mathbb{1}$ is the usual singleton set $\mathbb{1}_{\mathcal{S}} = \{()\}$, equipped with its only σ-algebra $\mathbb{1}_{\mathcal{M}} = \{\{\}, \{()\}\}$. The Cartesian product space $\alpha \times \beta$ is the usual Cartesian product set $(\alpha \times \beta)_{\mathcal{S}} = \alpha_{\mathcal{S}} \times \beta_{\mathcal{S}}$, equipped with the σ-algebra generated by its *measurable rectangles*:

$$(\alpha \times \beta)_{\mathcal{M}} = \sigma\big(\{A \times B \mid A \in \alpha_{\mathcal{M}},\, B \in \beta_{\mathcal{M}}\}\big). \tag{28}$$

The disjoint-union space $\alpha + \beta$ is the usual disjoint-union set, equipped with the σ-algebra generated by the measurable subsets of α and β. The measurable subsets of the real numbers $\mathbb{R}$ are the σ-algebra generated by the intervals.

Finally, the space $\mathbb{M}\,\alpha$ is the set of measures on the space α, equipped with the smallest σ-algebra that makes the function $\lambda\mu.\, [\![\mu]\!]_I(f)$ from $\mathbb{M}\,\alpha$ to $\mathbb{R}^+$ measurable for every integrand f:

$$(\mathbb{M}\,\alpha)_{\mathcal{M}} = \sigma\big(\{\, \{\mu \mid [\![\mu]\!]_I(f) \in S\} \mid f \in (\alpha \to \mathbb{R}^+),\, S \in \mathbb{R}^+_{\mathcal{M}} \,\}\big) \tag{29}$$

This construction $\mathbb{M}$ is a monad on the category of measurable spaces (Giry 1982). Types of the form $\mathbb{M}\,\alpha$ correspond to effectful computations, where the effect is an extended version of probabilistic choice. A morphism from α to β in the Kleisli category is a measurable function from α to $\mathbb{M}\,\beta$, or a *kernel* from α to β.

Because measurable functions do not themselves form a measurable space (Aumann 1961), functions in core Hakaru are *not* first-class types. Full Hakaru does have function types and λ-abstraction, but we avoid putting an arrow type under an $\mathbb{M}$ constructor.

Syntax The syntactic forms associated with $\mathbb{1}$, pairs, sums, and reals are mostly standard; the lone exception is the elimination form for sums. There is no primitive **case** expression; as shown below, **case** desugars to effectful elimination forms that treat pattern-matching failure as failure to make a probabilistic choice (a term that denotes the zero measure). Core Hakaru has to deal with failure regardless, and by encoding conditionals using failure, we reduce the number of syntactic forms the disintegrator has to handle.

The syntax of core Hakaru, as described in Figure 3, is set up to make it easy to explain our automatic disintegrator. An *atomic* term u, also called a neutral term (Dybjer and Filinski 2002), is one that the disintegrator cannot reason about or improve, because it mentions at least one variable whose value is fixed and unknown. (In partial-evaluation terms, the variable is *dynamic*.) One such variable is the variable t, introduced in Section 2.4 to stand for an observed value. Atomic terms include such variables, plus applications of strict functions to other atomic terms and to real literals r.

The next line in Figure 3 describes the *binding* forms or *guards* that can be used in core Hakaru's **do** notation. The binding form $x \leftsquigarrow e$ is the classic binding in the monad of measures (or the probability monad); as shown in Figure 4, e must have type $\mathbb{M}\ \alpha$ and x is bound with type α. The **let inl** and **let inr** forms are the elimination forms for sums; here e is a term of type $\alpha+\beta$, and depending on its value, the binding acts either as a **let** binding or as monadic failure. Finally, the **factor** form weights the measure by a real scaling factor. All four of these forms have effects in the monad of measures: probabilistic choice, failure, or weighting the measure.

The definition of terms is split into full terms and *head normal forms*. A head normal form v is an application of a known constructor, as is standard in lazy languages. Head normal forms include atomic terms; all the **do** forms; the standard constructors for a monad with zero and plus; real literals; and standard introduction forms for $\mathbb{1}$, pairs, and sums. They also include two forms specialized for probabilistic reasoning: **lebesgue** denotes the Lebesgue measure on the real line, and **uniform** $r_1\ r_2$ denotes the uniform distribution over the interval $[r_1, r_2]$. Full terms e include head normal forms, all variables, and applications of the same strict functions used in the definition of atomic terms.

Typing The typing rules of core Hakaru are intended to be unsurprising; Figure 4 simply formalizes what we say informally about the terms. Figure 4 shows rules only for terms of measure type; rules for terms of other types are standard and are omitted.

Semantics Core Hakaru is defined denotationally. The denotation of an expression is defined in the context of an environment ρ, which maps variables onto values drawn from the measurable spaces described above. Formally, if $x : \alpha$, then $\rho(x) \in \alpha_{\mathcal{S}}$. The denotation of a term e is written $[\![e]\!]\rho$, except if e has measure type then it denotes an integrator $[\![e]\!]_I\rho$. The semantic equations at non-measure types are standard and are omitted. The semantic equations at measure types are given in Figure 5, where we define each integrator by showing the result of applying it to an integrand f.

- A term **do** $\{x \leftsquigarrow e; e'\}$ denotes the integral, with respect to the measure denoted by e, of (roughly) the function $\lambda x.\, e'$.
- The sum-elimination forms are the most unusual; depending on the value of the right-hand side, a sum-elimination form acts either as a **let**-binding or as the zero measure.
- The next four forms are not so interesting: **factor** scales the integral by a real factor; **return** e integrates f by applying f to the value of e; **fail** denotes the zero measure; and **mplus** defines integration by a sum as the sum of the integrals.
- Finally, **lebesgue** and **uniform** $r_1\ r_2$ denote integrators over the real line and over the interval $[r_1, r_2]$ respectively.

For each e, the function from ρ to $[\![e]\!]\rho$ or to $[\![e]\!]_I\rho$ is measurable.

Syntactic sugar Hakaru is less impoverished than Figure 3 makes it appear. Here is some syntactic sugar:

$$\begin{array}{ll} \texttt{Bool} \equiv \mathbb{1}+\mathbb{1} & e_1 - e_2 \equiv e_1 + (-e_2) \\ \textbf{true} \equiv \textbf{inl}\ () & \textbf{observe}\ e \equiv \textbf{let inl}\ \_ = e \\ \textbf{false} \equiv \textbf{inr}\ () & \textbf{let}\ x = e \equiv x \leftsquigarrow \textbf{return}\ e\ \text{ or }\ \textbf{let inl}\ e = \textbf{inl}\ e \end{array} \tag{30}$$

Core Hakaru can provide syntactic sugar for **case** only when the term being desugared has measure type:

$$\begin{aligned} &\textbf{case}\ e\ \textbf{of inl}\ x_1 \Rightarrow e_1 \mid \textbf{inr}\ x_2 \Rightarrow e_2 \\ &\quad \equiv \textbf{mplus}\ (\textbf{do}\ \{\textbf{let inl}\ x_1 = e; e_1\})\ (\textbf{do}\ \{\textbf{let inr}\ x_2 = e; e_2\}) \end{aligned} \tag{31}$$

Full Hakaru desugars **case** in any context; desugaring lifts **case** into its evaluation context until it has measure type. Similar tactics apply to other syntactic sugar, such as the ternary comparison $e_1 \le e_2 \le e_3$ used in our examples.

We also extend core Hakaru's **do** notation to contain any number of binding forms. To help specify this extension, we define a *heap* to be a sequence of binding forms:

$$\text{Heaps} \qquad h ::= [g_1; g_2; \ldots; g_n] \tag{32}$$

We then define inductively

$$\textbf{do}\ \{[]; e\} \equiv e, \tag{33}$$

$$\textbf{do}\ \{[g_1; g_2; \ldots; g_n]; e\} \equiv \textbf{do}\ \{g_1; \textbf{do}\ \{[g_2; \ldots; g_n]; e\}\}. \tag{34}$$

As detailed in Section 5, heaps are central to our disintegrator.

We need no sugar to write discrete distributions; they are coded using **factor** and **mplus**. For example, here is a biased coin that comes up **true** with probability r:

$$\begin{aligned} \mathit{bernoulli}\ r \equiv \textbf{mplus}\ &(\textbf{do}\ \{\textbf{factor}\ r; \quad\ \textbf{return true}\}) \\ &(\textbf{do}\ \{\textbf{factor}\ (1-r); \textbf{return false}\}) \end{aligned} \tag{35}$$

5. Automating disintegration

Our automatic disintegrator takes a term $\vec{m} : \mathbb{M}\ (\alpha \times \beta)$ of core Hakaru and transforms it into an equivalent program of the form

$$\vec{m} \equiv \textbf{do}\ \{t \leftsquigarrow m'; p \leftsquigarrow k'\ t; \textbf{return}\ (t, p)\}, \tag{36}$$

where $m' : \mathbb{M}\ \alpha$ and $k' : \alpha \to \mathbb{M}\ \beta$. In this paper we focus on the continuous case where $\alpha = \mathbb{R}$ and $m' \equiv$ **lebesgue**. Other types α such as $\alpha = \mathbb{R} \times \mathbb{R}$ can be handled by successive disintegration. And as discussed in Section 3.3, $m' \equiv$ **lebesgue** is sufficient in the common case where the distribution of the observation is absolutely continuous with respect to the Lebesgue measure.

Before diving into automatic disintegration, let's first work out a disintegration by hand. We continue our example from Figure 2(a).

Disintegration by manipulating integrals We disintegrate this term, which is equivalent to $\vec{m}_{2\mathrm{a}}$ in equation (10):

$$\begin{aligned} \vec{m}'_{2\mathrm{a}} \equiv \textbf{do}\ \{&x \leftsquigarrow \textbf{uniform}\ 0\ 1; \\ &y \leftsquigarrow \textbf{uniform}\ 0\ 1; \\ &\textbf{return}\ (y - 2 \cdot x, (x, y))\} \end{aligned} \tag{37}$$

Our semantics gives this program the measure $[\![\vec{m}'_{2\mathrm{a}}]\!]_I$; because the term has two monadic bind operations, it defines an integrator with two integrals (Figure 5):

$$[\![\vec{m}'_{2\mathrm{a}}]\!]_I f = \int_{[0,1]} \int_{[0,1]} f(y - 2 \cdot x, (x, y))\, dy\, dx. \tag{38}$$

We calculate a disintegration by rewriting this integral, but first we cast it into integrator notation. The Lebesgue integral over a set S is defined by

$$\int_S g(a)\, da = \int_{\mathbb{R}} \chi_S(a) \cdot g(a)\, da, \tag{39}$$

and in integrator notation it is $[\![\Lambda]\!]_I(\lambda a.\, \chi_S(a) \cdot g(a))$, where Λ is the Lebesgue measure. We calculate a disintegration by using this definition to rewrite the integrals in equation (38):

$$\begin{aligned} &[\![\Lambda]\!]_I(\lambda x.\, \chi_{[0,1]}(x) \cdot [\![\Lambda]\!]_I(\lambda y.\, \chi_{[0,1]}(y) \cdot f(y - 2 \cdot x, (x, y)))) \\ &= \{\text{changing the integration variable from } y \text{ to } t = y - 2 \cdot x\} \qquad (40) \\ &\quad [\![\Lambda]\!]_I(\lambda x.\, \chi_{[0,1]}(x) \cdot [\![\Lambda]\!]_I(\lambda t.\, \chi_{[0,1]}(t + 2 \cdot x) \cdot f(t, (x, t + 2 \cdot x)))) \\ &= \{\text{exchanging integrals}\} \qquad (41) \\ &\quad [\![\Lambda]\!]_I(\lambda t.\, [\![\Lambda]\!]_I(\lambda x.\, \chi_{[0,1]}(x) \cdot \chi_{[0,1]}(t + 2 \cdot x) \cdot f(t, (x, t + 2 \cdot x)))) \end{aligned}$$

The final right-hand-side matches the right-hand-side of (25):

$$\mu = \Lambda, \tag{42}$$

$$[\![\kappa\, t]\!]_I\, f = [\![\Lambda]\!]_I(\lambda x.\, \chi_{[0,1]}(x) \cdot \chi_{[0,1]}(t + 2 \cdot x) \cdot f(x, t + 2 \cdot x)). \tag{43}$$

When we convert μ and κ back to terms of core Hakaru, they match the desired form (36). The result is exactly (12):

$$m' \equiv \textbf{lebesgue}, \quad k'\ t \equiv \textbf{do}\ \{x \leftsquigarrow \textbf{uniform}\ 0\ 1; \quad \textbf{observe}\ 0 \le t + 2 \cdot x \le 1; \quad \textbf{return}\ (x, t + 2 \cdot x)\}. \tag{44}$$

Derivations of this kind use two transformations again and again:

- *Move the observed variable to the outermost position* by exchanging order of integration (41). This reordering is justified by Tonelli's theorem.
- *Change the variable of integration*, which requires inverting a deterministic computation (40). Many of us haven't studied calculus for a long time, but if you want to revisit it, changing variables from y to x requires that $y = g(x)$ where g is invertible and differentiable. An integral over y is transformed into an integral over x using the inverse g^{-1} and the derivative g':

$$[\![\Lambda]\!]_I\big(\lambda y.\, f(g^{-1}(y), y)\big) = [\![\Lambda]\!]_I\big(\lambda x.\, |g'(x)| \cdot f(x, g(x))\big). \tag{45}$$

5.1 Disintegration by lazy partial evaluation

To exchange order of integration, we use lazy partial evaluation (Fischer et al. 2008). In the terminology of partial evaluation, the observed variable t, and indeed all atomic expressions of core Hakaru, are *dynamic*: the disintegrator treats them as opaque, constant values, which it does not try to reason about or manipulate. But variables that are $\leftsquigarrow$-bound or **let**-bound in the source code, as well as expressions that depend only on them, are *static*: the disintegrator is free to rewrite them and emit code for them in any way that preserves semantics. Bindings are rewritten and emitted using a list of bindings called a *heap*, as found in natural semantics of lazy languages (Launchbury 1993). To help explain how the disintegrator is structured and how it manipulates the heap, we appeal to a metaphor from *modes* in logic programming.

Imagine an abstract machine that runs core Hakaru by "making random choices." If $\mathcal{R}$ represents random choices and v represents a value, the outcome of running a term m of measure type might be described by a relation

$$m \overset{\mathcal{R}}{\rightsquigarrow} \textbf{return}\ v. \tag{46}$$

In our disintegrator, random choices $\mathcal{R}$ are represented by the heap, and the disintegrator operates in two modes: in *forward* mode, we are given m and we produce a value and extend the heap; in *backward* mode, we are given m and a value, and we rewrite the heap, constraining random choices, to ensure the outcome is the value we want. Randomized computation is lazy but not pure, and the evaluation of a term *not* of measure type can *also* constrain random choices on the heap. So these terms are also processed using the same two modes of computation.

Leaving the metaphor, we are not evaluating terms to produce values—we are *partially* evaluating terms to produce *code*. Manipulating terms m of measure type and terms e of any type, in two modes each, calls for four functions:

$\triangleright\, e$	Emit code to evaluate e.
$\gg m$	Put m's bindings on the heap and emit code to perform its final action.
$\triangleleft\, e\ v$	Constrain e to evaluate to the head normal form v by determining the outcome of a choice on the heap.
$\ll m\ v$	Put m's bindings on the heap and constrain its final action to produce v.

The functions are pronounced "evaluate" ($\triangleright$), "perform" ($\gg$), "constrain value" ($\triangleleft$), and "constrain outcome" ($\ll$). Each function may emit code and may update the heap by adding or changing a binding. When disintegration is complete, the final heap is rematerialized into the residual program.

We manage effects using continuation-passing style Updating the heap is not the only effect. From time to time the disintegrator decides on the next monadic action that the emitted code should take, such as $x \leftsquigarrow \textbf{uniform}\ 0\ 1$ in (44). In integrator terms, the disintegrator decides on the next outermost integral, such as $[\![\Lambda]\!]_I(\lambda x.\, \chi_{[0,1]}(x) \cdot \cdots)$ in (43). In this example, the action is a random choice that the disintegrator discharges from the heap; in other examples it could be a **factor** or **observe** emitted by $\triangleleft$ or $\ll$ to preserve semantics, or an **mplus** or **fail** copied from input to output. To emit all those actions while updating the heap, we use continuation-passing style (Bondorf 1992; Lawall and Danvy 1994; Danvy, Malmkjær, and Palsberg 1996).

The answer type of our continuation-passing style is the type of functions that take a heap and produce a residual program, or informally

$$ans = heap \to \lfloor \mathbb{M}\ \gamma \rfloor, \tag{47}$$

in which $\lfloor \mathbb{M}\ \gamma \rfloor$ means head normal forms of type $\mathbb{M}\ \gamma$. Using this answer type, we can now state the types of the four functions that make up our disintegrator. Writing $\lceil \alpha \rceil$ for terms of type α and $\lfloor \alpha \rfloor$ for head normal forms of type α, we have

$$\gg \text{("perform")} : \lceil \mathbb{M}\ \alpha \rceil \to (\lfloor \alpha \rfloor \to ans) \to ans \tag{48}$$

$$\triangleright \text{("evaluate")} : \lceil \alpha \rceil \to (\lfloor \alpha \rfloor \to ans) \to ans \tag{49}$$

$$\ll \text{("constrain outcome")} : \lceil \mathbb{M}\ \alpha \rceil \to \lfloor \alpha \rfloor \to ans \to ans \tag{50}$$

$$\triangleleft \text{("constrain value")} : \lceil \alpha \rceil \to \lfloor \alpha \rfloor \to ans \to ans \tag{51}$$

The disintegrator itself takes a term $\tilde{m} : \lceil \mathbb{M}\ (\alpha \times \beta) \rceil$ which stands for the extended prior model, plus a variable name $t : \lfloor \alpha \rfloor$ which stands for the observable. To disintegrate $\tilde{m}$, we invoke

$$k' = \gg \tilde{m}\ (\lambda v. \triangleleft\ (fst\ v)\ t\ (\lambda h'.\, \textbf{do}\ \{h'; \textbf{return}\ (snd\ v)\}))\ []. \tag{52}$$

This continuation-passing invocation starts with an empty initial heap and proceeds in three steps:

1. Perform $\tilde{m}$ and name the resulting pair v.
2. Constrain $fst\ v$ to equal t.
3. Form a residual program by combining the final heap h' with the final action **return** $(snd\ v)$.

The residual program k' represents the posterior distribution, which depends on free variable t.

5.2 Definition and example use of the disintegrator

The four functions that constitute our disintegrator are defined in Figure 6, where following Stoy (1977, Chapter 3), we use the classic $[\![\cdots]\!]$ quasiquotation brackets. Inside the brackets, juxtaposition stands for the construction of syntax; outside the brackets, juxtaposition stands for function application in the metalanguage. But as Mainland (2007) observes, quasiquotation is far more useful when used together with antiquotation. We have been unable to discover a standard notation for antiquotation, so from the POSIX shell language we have borrowed the $\$(\cdots)$ notation. Within $\$(\cdots)$, juxtaposition once again stands for function application in the metalanguage. And, again following Stoy, we quasiquote metalanguage variables in pattern matching and on right-hand sides.

In Figure 6, the metalanguage functions used most often are *smart constructors*. We use one smart constructor for each strict function in the object language. (These are the functions that define atomic terms in Figure 3.) For example, *fst* is a smart version of **fst**: when it gets a term of the form (e_1, e_2), it returns e_1; when given any other e, it returns **fst** e. As another example, smart $+$, when given

$\gg$ ("perform") : $\lceil \mathbb{M}\ \alpha \rceil \to (\lfloor \alpha \rfloor \to \mathit{heap} \to \lfloor \mathbb{M}\ \gamma \rfloor) \to \mathit{heap} \to \lfloor \mathbb{M}\ \gamma \rfloor$

$$\gg [\![u]\!]\ c\ h = [\![\textbf{do}\ \{x \leftsquigarrow u;\ \$(c\ x\ h)\}]\!] \quad \text{where } u \text{ is atomic, } x \text{ is fresh} \tag{53}$$
$$\gg [\![\textbf{lebesgue}]\!]\ c\ h = [\![\textbf{do}\ \{x \leftsquigarrow \textbf{lebesgue};\ \$(c\ x\ h)\}]\!] \quad \text{where } x \text{ is fresh} \tag{54}$$
$$\gg [\![\textbf{uniform}\ r_1\ r_2]\!]\ c\ h = [\![\textbf{do}\ \{x \leftsquigarrow \textbf{uniform}\ r_1\ r_2;\ \$(c\ x\ h)\}]\!] \quad \text{where } x \text{ is fresh} \tag{55}$$
$$\gg [\![\textbf{return}\ e]\!]\ c\ h = \triangleright\ [\![e]\!]\ c\ h \tag{56}$$
$$\gg [\![\textbf{do}\ \{g; e\}]\!]\ c\ h = \gg [\![e]\!]\ c\ [h; [\![g]\!]] \quad \text{unless } g \text{ binds a variable in } h \tag{57}$$
$$\gg [\![\textbf{fail}]\!]\ c\ h = [\![\textbf{fail}]\!] \tag{58}$$
$$\gg [\![\textbf{mplus}\ e_1\ e_2]\!]\ c\ h = [\![\textbf{mplus}\ \$(\gg [\![e_1]\!]\ c\ h)\ \$(\gg [\![e_2]\!]\ c\ h)]\!] \tag{59}$$
$$\gg [\![e]\!]\ c\ h = \triangleright\ [\![e]\!]\ (\lambda m.\ \gg m\ c)\ h \quad \text{where } e \text{ is not in head normal form} \tag{60}$$

$\triangleright$ ("evaluate") : $\lceil \alpha \rceil \to (\lfloor \alpha \rfloor \to \mathit{heap} \to \lfloor \mathbb{M}\ \gamma \rfloor) \to \mathit{heap} \to \lfloor \mathbb{M}\ \gamma \rfloor$

$$\triangleright [\![v]\!]\ c\ h = c\ [\![v]\!]\ h \quad \text{where } v \text{ is in head normal form} \tag{61}$$
$$\triangleright [\![\textbf{fst}\ e_0]\!]\ c\ h = \triangleright\ [\![e_0]\!]\ (\lambda v_0.\triangleright\ (\mathit{fst}\ v_0)\ c)\ h \quad \text{unless } e_0 \text{ is atomic} \tag{62}$$
$$\triangleright [\![\textbf{snd}\ e_0]\!]\ c\ h = \triangleright\ [\![e_0]\!]\ (\lambda v_0.\triangleright\ (\mathit{snd}\ v_0)\ c)\ h \quad \text{unless } e_0 \text{ is atomic} \tag{63}$$
$$\triangleright [\![-e_0]\!]\ c\ h = \triangleright\ [\![e_0]\!]\ (\lambda v_0.c\ (-v_0))\ h \tag{64}$$
$$\triangleright [\![e_0^{-\mathbf{1}}]\!]\ c\ h = \triangleright\ [\![e_0]\!]\ (\lambda v_0.c\ (v_0^{-1}))\ h \tag{65}$$
$$\triangleright [\![e_1 + e_2]\!]\ c\ h = \triangleright\ [\![e_1]\!]\ (\lambda v_1.\triangleright\ [\![e_2]\!]\ (\lambda v_2.c\ (v_1 + v_2)))\ h \tag{66}$$
$$\triangleright [\![e_1 \bullet e_2]\!]\ c\ h = \triangleright\ [\![e_1]\!]\ (\lambda v_1.\triangleright\ [\![e_2]\!]\ (\lambda v_2.c\ (v_1 \cdot v_2)))\ h \tag{67}$$
$$\triangleright [\![e_1 \leq e_2]\!]\ c\ h = \triangleright\ [\![e_1]\!]\ (\lambda v_1.\triangleright\ [\![e_2]\!]\ (\lambda v_2.c\ (v_1 \leq v_2)))\ h \tag{68}$$
$$\triangleright [\![x]\!]\ c\ [h_1; [\![x \leftsquigarrow e]\!]; h_2] = \gg [\![e]\!]\ (\lambda v.\lambda h_1'.c\ v\ [h_1'; [\![\textbf{let}\ x = v]\!]; h_2])\quad h_1 \tag{69}$$
$$\triangleright [\![x]\!]\ c\ [h_1; [\![\textbf{let inl}\ x = e_0]\!]; h_2] = \triangleright\ [\![e_0]\!]\ (\lambda v_0.\mathit{outl}\ v_0\ (\lambda e.\triangleright\ e\ (\lambda v.\lambda h_1'.c\ v\ [h_1'; [\![\textbf{let}\ x = v]\!]; h_2])))\ h_1 \tag{70}$$
$$\triangleright [\![x]\!]\ c\ [h_1; [\![\textbf{let inr}\ x = e_0]\!]; h_2] = \triangleright\ [\![e_0]\!]\ (\lambda v_0.\mathit{outr}\ v_0\ (\lambda e.\triangleright\ e\ (\lambda v.\lambda h_1'.c\ v\ [h_1'; [\![\textbf{let}\ x = v]\!]; h_2])))\ h_1 \tag{71}$$

$\ll$ ("constrain outcome") : $\lceil \mathbb{M}\ \mathbb{R} \rceil \to \lfloor \mathbb{R} \rfloor \to (\mathit{heap} \to \lfloor \mathbb{M}\ \gamma \rfloor) \to \mathit{heap} \to \lfloor \mathbb{M}\ \gamma \rfloor$

$$\ll [\![u]\!]\ v\ c\ h = \bot \quad \text{where } u \text{ is atomic} \tag{72}$$
$$\ll [\![\textbf{lebesgue}]\!]\ v\ c\ h = c\ h \tag{73}$$
$$\ll [\![\textbf{uniform}\ r_1\ r_2]\!]\ v\ c\ h = [\![\textbf{do}\ \{\textbf{observe}\ \$(r_1 \leq v \leq r_2); \textbf{factor}\ \$((r_2 - r_1)^{-1}); \$(c\ h)\}]\!] \tag{74}$$
$$\ll [\![\textbf{return}\ e]\!]\ v\ c\ h = \triangleleft\ [\![e]\!]\ v\ c\ h \tag{75}$$
$$\ll [\![\textbf{do}\ \{g; e\}]\!]\ v\ c\ h = \ll [\![e]\!]\ v\ c\ [h; [\![g]\!]] \quad \text{unless } g \text{ binds a variable in } h \tag{76}$$
$$\ll [\![\textbf{fail}]\!]\ v\ c\ h = [\![\textbf{fail}]\!] \tag{77}$$
$$\ll [\![\textbf{mplus}\ e_1\ e_2]\!]\ v\ c\ h = [\![\textbf{mplus}\ \$(\ll [\![e_1]\!]\ v\ c\ h)\ \$(\ll [\![e_2]\!]\ v\ c\ h)]\!] \tag{78}$$
$$\ll [\![e]\!]\ v\ c\ h = \triangleright\ [\![e]\!]\ (\lambda m.\ll m\ v\ c)\ h \quad \text{where } e \text{ is not in head normal form} \tag{79}$$

$\triangleleft$ ("constrain value") : $\lceil \mathbb{R} \rceil \to \lfloor \mathbb{R} \rfloor \to (\mathit{heap} \to \lfloor \mathbb{M}\ \gamma \rfloor) \to \mathit{heap} \to \lfloor \mathbb{M}\ \gamma \rfloor$

$$\triangleleft [\![u]\!]\ v\ c\ h = \bot \quad \text{where } u \text{ is atomic} \tag{80}$$
$$\triangleleft [\![r]\!]\ v\ c\ h = \bot \quad \text{where } r \text{ is a literal real number} \tag{81}$$
$$\triangleleft [\![\textbf{fst}\ e_0]\!]\ v\ c\ h = \triangleright\ [\![e_0]\!]\ (\lambda v_0.\triangleleft\ (\mathit{fst}\ v_0)\ v\ c)\ h \quad \text{unless } e_0 \text{ is atomic} \tag{82}$$
$$\triangleleft [\![\textbf{snd}\ e_0]\!]\ v\ c\ h = \triangleright\ [\![e_0]\!]\ (\lambda v_0.\triangleleft\ (\mathit{snd}\ v_0)\ v\ c)\ h \quad \text{unless } e_0 \text{ is atomic} \tag{83}$$
$$\triangleleft [\![-e_0]\!]\ v\ c\ h = \triangleleft\ [\![e_0]\!]\ (-v)\ c\ h \tag{84}$$
$$\triangleleft [\![e_0^{-\mathbf{1}}]\!]\ v\ c\ h = [\![\textbf{do}\ \{\textbf{factor}\ \$((v \cdot v)^{-1}); \$(\triangleleft\ [\![e_0]\!]\ v^{-1}\ c\ h)\}]\!] \tag{85}$$
$$\triangleleft [\![e_1 + e_2]\!]\ v\ c\ h = \triangleright\ [\![e_1]\!]\ (\lambda v_1.\triangleleft\ [\![e_2]\!]\ (v - v_1)\ c)\ h \tag{86}$$
$$\sqcup\ \triangleright\ [\![e_2]\!]\ (\lambda v_2.\triangleleft\ [\![e_1]\!]\ (v - v_2)\ c)\ h$$
$$\triangleleft [\![e_1 \bullet e_2]\!]\ v\ c\ h = \triangleright\ [\![e_1]\!]\ (\lambda v_1.\mathit{abs}\ v_1\ (\lambda v_1'.\lambda h'.[\![\textbf{do}\ \{\textbf{factor}\ \$(v_1'^{-1}); \$(\triangleleft\ [\![e_2]\!]\ (v \cdot v_1^{-1})\ c\ h')\}]\!]))\ h \tag{87}$$
$$\sqcup\ \triangleright\ [\![e_2]\!]\ (\lambda v_2.\mathit{abs}\ v_2\ (\lambda v_2'.\lambda h'.[\![\textbf{do}\ \{\textbf{factor}\ \$(v_2'^{-1}); \$(\triangleleft\ [\![e_1]\!]\ (v \cdot v_2^{-1})\ c\ h')\}]\!]))\ h$$
$$\triangleleft [\![x]\!]\ v\ c\ [h_1; [\![x \leftsquigarrow e]\!]; h_2] = \ll [\![e]\!]\ v\ (\lambda h_1'.c\ [h_1'; [\![\textbf{let}\ x = v]\!]; h_2])\quad h_1 \tag{88}$$
$$\triangleleft [\![x]\!]\ v\ c\ [h_1; [\![\textbf{let inl}\ x = e_0]\!]; h_2] = \triangleright\ [\![e_0]\!]\ (\lambda v_0.\mathit{outl}\ v_0\ (\lambda e.\triangleleft\ e\ v\ (\lambda h_1'.c\ [h_1'; [\![\textbf{let}\ x = v]\!]; h_2])))\ h_1 \tag{89}$$
$$\triangleleft [\![x]\!]\ v\ c\ [h_1; [\![\textbf{let inr}\ x = e_0]\!]; h_2] = \triangleright\ [\![e_0]\!]\ (\lambda v_0.\mathit{outr}\ v_0\ (\lambda e.\triangleleft\ e\ v\ (\lambda h_1'.c\ [h_1'; [\![\textbf{let}\ x = v]\!]; h_2])))\ h_1 \tag{90}$$

Figure 6. The implementation of our lazy partial evaluator and disintegrator over $\mathbb{R}$

two real literals, returns a real literal representing their sum; when given any other object-language terms e_1 and e_2, it returns $e_1 + e_2$. Smart constructors are part of functional-programming folklore; they implement constant folding and more besides.

One pair of smart constructors requires more explanation. Functions *outl* and *outr* are the inverses of **inl** and **inr**, but unlike the strict functions, they can fail, and failure is an effect. This effect is handled by adding a binding to the object program, which we do by passing a continuation to *outl*. Function *outl* is given a term of type $\alpha + \beta$ and a continuation that expects a term of type α. The answer type is, as always, a function from a heap to a term of measure type:

$$\mathit{outl}\colon \lfloor \alpha + \beta \rfloor \to (\lceil \alpha \rceil \to \mathit{heap} \to \lfloor \mathbb{M}\ \gamma \rfloor) \to \mathit{heap} \to \lfloor \mathbb{M}\ \gamma \rfloor$$
$$\mathit{outl}\ [\![\mathbf{inl}\ e]\!]\ c\ h = c\ e\ h$$
$$\mathit{outl}\ [\![\mathbf{inr}\ e]\!]\ c\ h = [\![\mathbf{fail}]\!]$$
$$\mathit{outl}\ [\![u]\!] \qquad c\ h = [\![\mathbf{do}\ \{\mathbf{let\ inl}\ x = u; \$(c\ x\ h)\}]\!], \text{ where } x \text{ is fresh.}$$

Function *outr* is analogous. And *abs*, which is used in case 87, is defined by the same method.

Figure 6 uses one more notational convention. When we said the disintegrator returns a term, we glossed over an important truth: the disintegrator actually *searches* for a term, and the search may find multiple terms, or it may fail. For an example of how the disintegrator might find multiple terms, suppose we call $\lhd\ [\![e_1 + e_2]\!]\ v$ to constrain $e_1 + e_2$ to be v. There are two ways to do it: evaluate e_1 to v_1 and constrain e_2 to be $v - v_1$, or evaluate e_2 to v_2 and constrain e_1 to be $v - v_2$. In case 86 of Figure 6, the disintegrator tries both alternatives, and the possibilities are separated by $\sqcup$, which represents nondeterministic choice. For an example of how disintegration might fail, suppose we want to constrain a real literal r to equal v (case 81). This case must fail because the point mass at r is not absolutely continuous with respect to the Lebesgue measure; the failure is written $\bot$. In our implementation, the search is written using Haskell's lazy lists: $\bot$ is represented by the empty list; $\sqcup$ stands for list append; and each single expression is represented by a singleton list.

Automatic disintegration by example Figure 6 is overwhelming in its detail, which we do not attempt to explain in full. The top two functions, $\gg$ and $\triangleright$, define an online partial evaluator that is more or less standard (Fischer et al. 2008; Fischer, Kiselyov, and Shan 2011). The bottom two functions, $\ll$ and $\lhd$, are where the innovation is. To illustrate how the ensemble work together, Figure 7 shows $\gg$ and $\lhd$ applied to our running example (37).

Like many models, the term at the top of Figure 7 begins with bindings, and the disintegrator applies case 57 twice, putting the bindings on the heap. When it gets to **return**, in case 56, it returns a pair to the continuation in (52). The continuation calls $\lhd$ to constrain the value of $y - 2 \cdot x$ to equal t.

The expression $y - 2 \cdot x$, like typical expressions passed to **return**, is formed by arithmetic on random variables. Function $\lhd$ handles arithmetic using cases 82 to 87. These cases search for a random variable that $\lhd$ can make deterministic, which happens in case 88. The idea is that observing t takes away a degree of freedom (a random choice), and in case 88, the probabilistic computation e is rewritten by $\ll$ in such a way that it is forced to produce v. In the example in Figure 7, the search goes to the left summand y, by choosing the right operand of $\sqcup$ in case 86.

In Figure 7, the right summand $-(2 \cdot x)$ is evaluated by $\triangleright$ in cases 64 and 67. Evaluation eventually reaches x, which is bound on the heap to **uniform** 0 1. Term **uniform** 0 1 represents a final action, and in case 55, function $\gg$ emits the action, binding the result to the fresh variable z. In the continuation $(\lambda v.\, \lambda h. \ldots)$ applied to z, because z is not bound on the heap, the disintegrator treats z as atomic.

$\gg$ [[**do** {$x \leftsquigarrow$ **uniform** 0 1;
$y \leftsquigarrow$ **uniform** 0 1;
return $(y - 2 \cdot x, (x, y))$}]]
$(\lambda v. \lhd (\mathit{fst}\ v)\ [\![t]\!]\ (\lambda h'. [\![\mathbf{do}\ \{h'; \mathbf{return}\ \$(\mathit{snd}\ v)\}]\!]))$
[]

= { put bindings on the heap; perform **return**: (57), (57), (56) }

$\triangleright$ [[$(y - 2 \cdot x, (x, y))$]]
$(\lambda v. \lhd (\mathit{fst}\ v)\ [\![t]\!]\ (\lambda h'. [\![\mathbf{do}\ \{h'; \mathbf{return}\ \$(\mathit{snd}\ v)\}]\!]))$
$[x \leftsquigarrow \mathbf{uniform}\ 0\ 1; y \leftsquigarrow \mathbf{uniform}\ 0\ 1]$

= { pass head normal form to continuation (61) }

$\lhd$ [[$y - 2 \cdot x$]] [[t]]
$(\lambda h'. [\![\mathbf{do}\ \{h'; \mathbf{return}\ (x, y)\}]\!])$
$[x \leftsquigarrow \mathbf{uniform}\ 0\ 1; y \leftsquigarrow \mathbf{uniform}\ 0\ 1]$

= { evaluate right summand and constrain left (86);
negate right summand (64) }

$\triangleright$ [[$2 \cdot x$]]
$(\lambda v. \lhd [\![y]\!]\ ([\![t]\!] + v)\ (\lambda h'. [\![\mathbf{do}\ \{h'; \mathbf{return}\ (x, y)\}]\!]))$
$[x \leftsquigarrow \mathbf{uniform}\ 0\ 1; y \leftsquigarrow \mathbf{uniform}\ 0\ 1]$

= { evaluate left multiplicand (67); continue with right (61) }

$\triangleright$ [[x]]
$(\lambda v. \lhd [\![y]\!]\ ([\![t]\!] + 2 \cdot v)\ (\lambda h'. [\![\mathbf{do}\ \{h'; \mathbf{return}\ (x, y)\}]\!]))$
$[x \leftsquigarrow \mathbf{uniform}\ 0\ 1; y \leftsquigarrow \mathbf{uniform}\ 0\ 1]$

= { perform the action bound to x in the heap (69) }

$\gg$ [[**uniform** 0 1]]
$(\lambda v.\, \lambda h. \lhd [\![y]\!]\ ([\![t]\!] + 2 \cdot v)$
$(\lambda h'. [\![\mathbf{do}\ \{h'; \mathbf{return}\ (x, y)\}]\!])$
$[h; x \leftsquigarrow \mathbf{return}\ v; y \leftsquigarrow \mathbf{uniform}\ 0\ 1])$
[]

= { emit code for the **uniform** action and continue (55) }

[[**do** {$z \leftsquigarrow$ **uniform** 0 1;
$\$(\lhd [\![y]\!]\ [\![t + 2 \cdot z]\!]$
$(\lambda h'. [\![\mathbf{do}\ \{h'; \mathbf{return}\ (x, y)\}]\!])$
$[x \leftsquigarrow \mathbf{return}\ z; y \leftsquigarrow \mathbf{uniform}\ 0\ 1])$}]]

= { constrain y: update its heap binding
and constrain outcome of its action (88)}

[[**do** {$z \leftsquigarrow$ **uniform** 0 1;
$\$(\ll [\![\mathbf{uniform}\ 0\ 1]\!]\ [\![t + 2 \cdot z]\!]$
$(\lambda h'. [\![\mathbf{do}\ \{[h'; y \leftsquigarrow \mathbf{return}\ (t + 2 \cdot z)]; \mathbf{return}\ (x, y)\}]\!])$
$[x \leftsquigarrow \mathbf{return}\ z])$}]]

= { constrain **uniform** by emitting observation (74);
beta-reduce continuation, leaving residual program }

[[**do** {$z \leftsquigarrow$ **uniform** 0 1;
observe $0 \le t + 2 \cdot z \le 1$; **factor** 1;
$x \leftsquigarrow$ **return** z; $y \leftsquigarrow$ **return** $(t + 2 \cdot z)$; **return** (x, y)}]]

Figure 7. Automatic disintegration in action

Once z is bound, it is time to constrain y by calling $\lhd\ [\![y]\!]\ [\![t + 2 \cdot z]\!]$. Case 88 finds the binding of y on the heap and replaces the random choice from **uniform** 0 1 with a deterministic binding to $t + 2 \cdot z$. The example continues by constraining the outcome of **uniform** 0 1 to equal $t + 2 \cdot z$. This constraint is enforced by case 74, which emits **observe** and **factor**. The updated heap is then passed to the continuation, which builds the final term in Figure 7. This term simplifies to the desired term $k'\ t$ in equation (44), by monad laws and removing **factor** 1.

Narrative of cases In the monadic part of our disintegrator, cases 53 to 55, 58, and 59 all emit code for "final" monadic actions. The "constrain outcome" ($\ll$) cases for the monad are more

interesting; cases 73 and 74 are the two cases where the disintegrator eliminates a random choice and instead forces it to produce v. As for the "constrain value" ($\triangleleft$) cases, they either search for a variable to constrain (cases 80 to 87) or propagate a constraint by converting a binding on the heap to a **let** binding (cases 88 to 90). To preserve semantics, the search uses computer algebra to alter the residual program (cases 84 to 87), much as Bhat et al.'s (2012, 2013) probability-density compiler does.

Pure code generation ($\triangleright$) is mostly standard; cases 61 to 68 use smart constructors and continuation-passing style to emit code. Case 69, for a variable x bound in the heap, is the interesting one: it calls $\gg$ to perform effectful code, then replaces x's binding. This case is the means by which the disintegrator reorders effects.

6. Evaluation

Our disintegration algorithm is part of the implementation of full Hakaru, a language that extends core Hakaru with additional types (such as integers and arrays), primitive operations (such as exp, log, and sin), and primitive distributions (such as normal, gamma, and Poisson). The full disintegrator handles not only real observations with respect to the Lebesgue measure but also integer observations with respect to the counting measure, as well as observations on types built inductively by products and sums.

The full disintegrator works on dozens of generative models and observations, including these:

- Uniform distributions over polytopes, like the unit square
- Normal distributions over correlated real variables (as in Bayesian linear regression)
- Discrete mixtures of continuous distributions, such as flipping a coin to decide whether to sample the observation from a normal or uniform distribution
- Continuous mixtures of discrete distributions, such as minting a biased coin then flipping it several times
- Two time steps of a linear dynamic model in one dimension
- A simplified model of how a seismic event is detected at a seismic station, involving deterministic computations of spherical geometry (Arora, Russell, and Sudderth 2013)
- The example observations in Section 2
- "Microbenchmark" observations such as $\sqrt{x} + e^{-y}$ and $\max(x, y)$ (an example used by Chang and Pollard (1997) to motivate disintegration)

Our disintegrator has always produced a result within a few seconds. And thanks to the smart constructors, it produces code that we find tolerably readable.

Our disintegrator works effectively as a component of larger systems. It produces posterior terms that can be simplified as mathematical expressions, executed as importance samplers, or further transformed into Hakaru programs that carry out other inference methods. For example, our disintegrator can compute the conditional distributions and densities (also called Radon-Nikodym derivatives) required by Monte Carlo sampling methods like importance sampling, Metropolis-Hastings, and Gibbs (MacKay 1998). By invoking the disintegrator, our implementations of these methods manage to stay independent of any model and work with all the distributions listed above.

6.1 Semantic grounding

We believe that our disintegrator is sound: if every kernel denoted by a subterm in the input is σ-finite, then the output denotes a disintegration of the input with respect to the Lebesgue measure. (We require σ-finite kernels (Pollard 2001) to apply Tonelli's theorem, which is what justifies exchanging the order of integration.) Formally, the invocation

$$\begin{array}{l} k' = \gg \overleftarrow{m} \\ \quad (\lambda v. \triangleleft (\mathit{fst}\ v)\ [\![t]\!]\ (\lambda h'. [\![\mathbf{do}\ \{h'; \mathbf{return}\ \$(\mathit{snd}\ v)\}]\!])) \\ \quad [] \end{array} \tag{91}$$

entails the equivalence

$$[\![\overleftarrow{m}]\!]_I = [\![\mathbf{do}\ \{t \leftsquigarrow \mathbf{lebesgue}; p \leftsquigarrow k'; \mathbf{return}\ (t,p)\}]\!]_I. \tag{92}$$

We have yet to prove this conjecture, but as a specification, it is indispensable. For example, the conjecture shows how to use the disintegrator to find a density. This result tells us that we can use our disintegrator in model-independent implementations of inference methods that require the density of a distribution, namely the implementations of importance sampling and Metropolis-Hastings mentioned above. As another example, because the specification describes the result of disintegration as a measure kernel, we can simplify the result and execute it as a sampler, as in our implementation of Gibbs sampling.

We have confidence in the conjecture not just because of our experimental results, but because the conjecture has guided us in defining and implementing the disintegrator. We originally discovered the disintegrator by generalizing a probability-density calculator that we derived equationally from a semantic specification. Detailed induction hypotheses, which specify what it means for each of the disintegrator's functions to preserve semantics, have guided us in adding support for tricky, non-invertible operators such as sine. We expect the same induction hypotheses to help us extend full Hakaru with more advanced computer algebra.

6.2 Limitations

Our disintegrator has limitations. Some are of little consequence, and some can probably be addressed by known techniques, but some may present obstacles to larger-scale use.

One limitation is that, like many proof-of-concept partial evaluators, the disintegrator can duplicate code (such as $t + 2 \cdot z$ in Figure 7) and produce large residual programs. The limitation does present an obstacle to large-scale use, but it should yield to standard techniques for eliminating common subexpressions.

Another limitation is that the disintegrator reasons locally and syntactically, and while it can disintegrate an observation like $2 \cdot x$, it can't disintegrate the semantically equivalent observation $x + x$. Perhaps surprisingly, this limitation has not caused trouble in practice. It may be that in today's probabilistic programs, the deterministic computations leading from random choice to final observation are relatively simple. If the limitation proves to be an obstacle in the future, it could be addressed by more advanced computer algebra.

Another current limitation is that only single elements of arrays may be observed—full Hakaru cannot express an observation of an entire array, as in a hidden Markov model. Extending Hakaru to provide container data types, loops over them, and container-typed observations would constitute *lifted inference* (Poole 2003). This extension is important future work that will be facilitated by our disintegrator's ability to cope with open input terms.

Another limitation is that when the disintegrator fails, it provides no diagnostics. Failure is typically caused by a distribution on observations that is not absolutely continuous with respect to the Lebesgue measure. For example, the observation distribution might assign nonzero probability to a single point on the real line. It would help if the disintegrator could produce some kind of trace of its

work, if the user could provide hints as to what they expect to be constrainable, and if we knew how to do "separate disintegration." That is all future work.

A final limitation is that when the output of the disintegrator is used as a sampler, it is not necessarily efficient. For example, the sampler derived from equation (13) rejects half the samples taken from **uniform** 0 1; it could be made more efficient by changing **uniform** 0 1 to **uniform** 0 $(1/2)$. We mitigate the limitation by tweaking the search to prefer choices that produce more efficient output programs. For example, **uniform** e_1 e_2 (which in full Hakaru does not require e_1 and e_2 to be real literals) can always be implemented by emitting **lebesgue**, but because **uniform** is more efficient, we try it first. These kinds of tricks will carry us only so far; in general, efficient sampling is a hard, longstanding problem that our denotational foundation for equational reasoning merely helps to address. In separate, ongoing, unpublished work, we are improving probabilistic programs using computer algebra.

7. Related work

Inference from zero-probability observations is a longstanding concern of measure-theoretic probability theory. In Kolmogorov's classic general approach (1933, Chapter 5), there is no such thing as a conditional distribution, only conditional expectations. Chang and Pollard (1997) advocate disintegration as an approach that is equally rigorous, more intuitive, and only slightly less general. As they describe, many authors have used topological notions such as continuity to give *mathematical* constructions of conditional distributions (Tjur 1975; Ackerman, Freer, and Roy 2014). For example, to specify the semantics of the probabilistic language Fun, Borgström et al. (2013) present a new topological construction for conditional distributions. By contrast, we address the problem *linguistically*: we find a term that represents a conditional distribution. If you like, we recast disintegration as a program-synthesis problem (Srivastava et al. 2011).

To specify the semantics of a probabilistic programming language with uncountable spaces, it is popular to let terms denote *samplers*, or computations that produce a random outcome when given a source of entropy (Park, Pfenning, and Thrun 2008). In contrast, we equate terms (not just whole programs) that denote the same *measure*, even if they denote different samplers. This semantics enables us to improve performance by reasoning equationally from inefficient samplers to efficient ones.

The two modes in our disintegrator may remind you of bidirectional programming (Foster, Matsuda, and Voigtländer 2012), as well as modes in bidirectional type checking (Dunfield and Krishnaswami 2013) and logic programming. The forward mode ($\triangleright$ or $\gg$) resembles lazy evaluation (Launchbury 1993), in particular lazy partial evaluation (Jørgensen 1992; Fischer et al. 2008; Mitchell 2010; Bolingbroke and Peyton Jones 2010). Our laziness postpones nondeterminism (Fischer, Kiselyov, and Shan 2011) in the measure monad, an extension of the probability monad (Giry 1982; Ramsey and Pfeffer 2002). The backward mode ($\triangleleft$ or $\ll$) resembles weakest-precondition reasoning (Dijkstra 1975; Nori et al. 2014), pre-image computation (Toronto, McCarthy, and Horn 2015), constraint propagation (Saraswat, Rinard, and Panangaden 1991; Gupta, Jagadeesan, and Panangaden 1999), and probability-density calculation (Bhat et al. 2012, 2013).

It is well known that continuations can be used to manage mutable state (Gunter, Rémy, and Riecke 1998; Kiselyov, Shan, and Sabry 2006), to express nondeterminism (Danvy and Filinski 1990) such as probabilistic choice (Kiselyov and Shan 2009), and to generate bindings and guards and improve binding times in partial evaluation (Bondorf 1992; Lawall and Danvy 1994; Danvy, Malmkjær, and Palsberg 1996; Dybjer and Filinski 2002). Our disintegrator uses continuations in all these ways. Meanwhile, thanks to the view of measures as integrators, our semantics uses continuation passing to compose measures easily (Audebaud and Paulin-Mohring 2009).

8. Discussion

This paper uses disintegration to infer a posterior distribution from an observation of a continuous real variable at a point—an event with probability zero. The problem is one of long standing, and it is where the rigor and reason of disintegration shine brightest. But disintegration is good for more than just real spaces and their products and sums; it works equally well on countable spaces, such as the Booleans and the integers. Countable spaces are easy to deal with; every measure is absolutely continuous with respect to the counting measure, so on countable spaces, disintegration gives exactly the same answers as the classic conditioning equation (5).

When disintegrating with respect to the counting measure, functions $\ll$ and $\triangleleft$ in Figure 6 needn't fail in cases 72, 80, and 81; because the Dirac measure is absolutely continuous with respect to the counting measure, these cases can succeed by emitting a pattern-matching guard. Extended thus, our disintegrator works on Figure 1(b): if we disintegrate on the observation $y \leq 2 \cdot x$, we get a kernel that maps **true** to the trapezoid and **false** to the triangle that is its complement. In Figure 1(c), if we disintegrate on the observation $y = 2 \cdot x$, we get a kernel that maps **true** to the zero measure and **false** to the uniform distribution over the unit square. Thus, to the paradoxical question posed in Section 2, what is $E(x)$ when observing the Boolean $y = 2 \cdot x$ is true, Hakaru responds that the posterior measure is zero and no expectation can be computed.

Eventually we want to disintegrate hybrid distributions $\xi = \mu \otimes \kappa$ where the underlying measure μ is not the Lebesgue measure but, say, a sum of the Lebesgue measure with a counting measure—that is, where probability is defined by the sum of a density function and a countable number of point masses. Such a measure may arise, for example, if a continuous distribution like a normal distribution is "clamped" by treating all negative values as if they were 0. We would also like to disintegrate distributions *embedded* in higher-dimensional spaces; for example, Metropolis-Hastings sampling calls for a measure on the union of two hyperplanes embedded in $\mathbb{R}^4$, defined by $\{(x_1,x_2,x_3,x_4) \mid x_1 = x_3 \vee x_2 = x_4\}$ (Tierney 1998). These examples work in a hacked version of our disintegrator, but the hack compromises soundness. We believe that instead, the disintegrator can be generalized beyond the Lebesgue measure in a sound and principled way.

We began this paper with an old paradox: you can't just claim to observe a zero-probability event and hope for a single correct posterior distribution. If instead you pose the inference problem in terms of an observed *expression*, then whether the probability of any given value is zero or not, disintegration decomposes your model into a meaningful *family* of posterior distributions. Every probabilistic programming language should let its users specify observations this way. And an automatic disintegrator delivers the posterior family in a wonderfully useful form: as a term in the modeling language. Such terms are amenable to equational reasoning and compositional reuse, which makes it easy to automate the implementation of inference algorithms that are usually coded by hand. We can't wait to see how we and others will use automatic disintegration to further advance the state of probabilistic programming.

Acknowledgments

Thanks to Nathanael Ackerman, Jacques Carette, Ryan Culpepper, Cameron Freer, Praveen Narayanan, Daniel Roy, Joe Stoy, Dylan

Thurston, Mitchell Wand, and Robert Zinkov for helpful comments and discussions.

This research was supported by DARPA grant FA8750-14-2-0007, NSF grant CNS-0723054, Lilly Endowment, Inc. (through its support for the Indiana University Pervasive Technology Institute), and the Indiana METACyt Initiative. The Indiana METACyt Initiative at IU is also supported in part by Lilly Endowment, Inc. The second author was supported by DARPA contract FA8750-14-C-0002.

References

Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. 2014. On computability and disintegration (extended abstract). In *11th International Conference on Computability and Complexity in Analysis*.

Nimar S. Arora, Stuart Russell, and Erik Sudderth. 2013. NET-VISA: Network processing vertically integrated seismic analysis. *Bulletin of the Seismological Society of America*, 103(2A): 709–729.

Philippe Audebaud and Christine Paulin-Mohring. 2009. Proofs of randomized algorithms in Coq. *Science of Computer Programming*, 74(8):568–589.

Robert J. Aumann. 1961. Borel structures for function spaces. *Illinois Journal of Mathematics*, 5(4):614–630.

Joseph Bertrand. 1889. *Calcul des Probabilités*. Gauthier-Villars et fils, Paris.

Sooraj Bhat, Ashish Agarwal, Richard Vuduc, and Alexander Gray. 2012. A type theory for probability density functions. In *POPL'12: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 545–556, New York. ACM Press.

Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio V. Russo. 2013. Deriving probability density functions from probabilistic functional programs. In *Proceedings of TACAS 2013: 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 7795 in Lecture Notes in Computer Science, pages 508–522, Berlin. Springer.

Maximilian Bolingbroke and Simon Peyton Jones. 2010. Supercompilation by evaluation. In *Haskell'10: Proceedings of the 2010 ACM SIGPLAN Haskell Symposium*, pages 135–146, New York. ACM Press.

Anders Bondorf. 1992. Improving binding times without explicit CPS-conversion. In *LFP'92: Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 1–10, New York. ACM Press.

Émile Borel. 1909. *Éléments de la Théorie des Probabilités*. Librairie scientifique A. Hermann et fils, Paris.

Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. 2013. Measure transformer semantics for Bayesian machine learning. *Logical Methods in Computer Science*, 9(3:11):1–39.

Joseph T. Chang and David Pollard. 1997. Conditioning as disintegration. *Statistica Neerlandica*, 51(3):287–317.

Olivier Danvy and Andrzej Filinski. 1990. Abstracting control. In *LFP'90: Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, New York. ACM Press.

Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. 1996. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 18(6):730–751.

Bruno de Finetti. 1974. *Theory of Probability: A Critical Introductory Treatment*, volume 1. Wiley, New York. Translated from *Teoria Delle Probabilità*, 1970.

Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457.

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP'13: Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming*, pages 429–442, New York. ACM Press.

Peter Dybjer and Andrzej Filinski. 2002. Normalization and partial evaluation. In *APPSEM 2000: International Summer School on Applied Semantics, Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Berlin. Springer.

Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. 2011. Purely functional lazy nondeterministic programming. *Journal of Functional Programming*, 21(4–5):413–465.

Sebastian Fischer, Josep Silva, Salvador Tamarit, and Germán Vidal. 2008. Preserving sharing in the partial evaluation of lazy functional programs. In *Revised Selected Papers from LOPSTR 2007: 17th International Symposium on Logic-Based Program Synthesis and Transformation*, number 4915 in Lecture Notes in Computer Science, pages 74–89, Berlin. Springer.

Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. 2012. Three complementary approaches to bidirectional programming. In *Generic and Indexed Programming, International Spring School, SSGIP 2010, Revised Lectures*, number 7470 in Lecture Notes in Computer Science, pages 1–46, Berlin. Springer.

Michèle Giry. 1982. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis: Proceedings of an International Conference Held at Carleton University, Ottawa, August 11–15, 1981*, number 915 in Lecture Notes in Mathematics, pages 68–85, Berlin. Springer.

Carl A. Gunter, Didier Rémy, and Jon G. Riecke. 1998. Return types for functional continuations. URL `http://pauillac.inria.fr/~remy/work/cupto/`.

Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. 1999. Stochastic processes as concurrent constraint programs. In *POPL'99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 189–202, New York. ACM Press.

Jesper Jørgensen. 1992. Generating a compiler for a lazy language by partial evaluation. In *POPL'92: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 258–268, New York. ACM Press.

Oleg Kiselyov and Chung-chieh Shan. 2009. Embedded probabilistic programming. In *Proceedings of the Working Conference on Domain-Specific Languages*, number 5658 in Lecture Notes in Computer Science, pages 360–384, Berlin. Springer.

Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. 2006. Delimited dynamic binding. In *ICFP'06: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 26–37, New York. ACM Press.

Andrey Nikolaevich Kolmogorov. 1933. *Grundbegriffe der Wahrscheinlichkeitsrechnung*. Springer, Berlin. URL `http://www.mathematik.com/Kolmogorov/`. English translation *Foundations of the Theory of Probability*, Chelsea, New York, 1950.

John Launchbury. 1993. A natural semantics for lazy evaluation. In *POPL'93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, New York. ACM Press.

Julia L. Lawall and Olivier Danvy. 1994. Continuation-based partial evaluation. In *LFP'94: Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 227–238, New York. ACM Press.

David J. C. MacKay. 1998. Introduction to Monte Carlo methods. In Michael I. Jordan, editor, *Learning and Inference in Graphical Models*. Kluwer, Dordrecht. Paperback: *Learning in Graphical Models*, MIT Press.

Geoffrey Mainland. 2007. Why it's nice to be quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, Haskell '07, pages 73–82, New York, NY, USA. ACM.

Neil Mitchell. 2010. Rethinking supercompilation. In *ICFP'10: Proceedings of the 2010 ACM SIGPLAN International Conference on Functional Programming*, pages 309–320, New York. ACM Press.

Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. 2014. R2: An efficient MCMC sampler for probabilistic programs. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, pages 2476–2482. AAAI Press.

Sungwoo Park, Frank Pfenning, and Sebastian Thrun. 2008. A probabilistic language based on sampling functions. *ACM Transactions on Programming Languages and Systems*, 31(1): 4:1–4:46.

David Pollard. 2001. *A User's Guide to Measure Theoretic Probability*. Cambridge University Press, Cambridge.

David Poole. 2003. First-order probabilistic inference. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 985–991, San Francisco, CA. Morgan Kaufmann.

Norman Ramsey and Avi Pfeffer. 2002. Stochastic lambda calculus and monads of probability distributions. In *POPL'02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 154–165, New York. ACM Press.

H. L. Royden. 1988. *Real Analysis*. Macmillan, third edition.

Vijay A. Saraswat, Martin C. Rinard, and Prakash Panangaden. 1991. Semantic foundations of concurrent constraint programming. In *POPL'91: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–352, New York. ACM Press.

Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. 2011. Path-based inductive synthesis for program inversion. In *PLDI'11: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 492–503, New York. ACM Press.

Joseph E. Stoy. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press.

Luke Tierney. 1998. A note on Metropolis-Hastings kernels for general state spaces. *The Annals of Applied Probability*, 8(1): 1–9.

Tue Tjur. 1975. A constructive definition of conditional distributions. Preprint 13, Institute of Mathematical Statistics, University of Copenhagen.

Neil Toronto, Jay McCarthy, and David Van Horn. 2015. Running probabilistic programs backwards. In *ESOP 2015: Proceedings of the 24th European Symposium on Programming*, number 9032 in Lecture Notes in Computer Science, pages 53–79, Berlin. Springer.