Exact Bayesian Inference by Symbolic Disintegration

Chung-chieh Shan
Indiana University, USA
ccshan@indiana.edu

Norman Ramsey
Tufts University, USA
nr@cs.tufts.edu

Abstract

Bayesian inference, of posterior knowledge from prior knowledge and observed evidence, is typically defined by Bayes's rule, which says the posterior multiplied by the probability of an observation equals a joint probability. But the observation of a continuous quantity usually has probability zero, in which case Bayes's rule says only that the unknown times zero is zero. To infer a posterior distribution from a zero-probability observation, the statistical notion of *disintegration* tells us to specify the observation as an expression rather than a predicate, but does not tell us how to compute the posterior. We present the first method of computing a disintegration from a probabilistic program and an expression of a quantity to be observed, even when the observation has probability zero. Because the method produces an exact posterior term and preserves a semantics in which monadic terms denote measures, it composes with other inference methods in a modular way—without sacrificing accuracy or performance.

Categories and Subject Descriptors G.3 [*Probability and Statistics*]: distribution functions, statistical computing; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—denotational semantics, partial evaluation

Keywords probabilistic programs, conditional measures, continuous distributions

1. Introduction

In the Bayesian approach to reasoning about uncertainty, one begins with a probability distribution representing *prior* knowledge or belief about the world. Then, given an *observation* of the world, one uses *probabilistic 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 provides an interface to one or more Bayesian inference algorithms. A 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. But in practice, the generative model is not run; it is used only as input to an inference algorithm which, like an interpreter or compiler, is reused across applications.

POPL'17, January 15–21, 2017, Paris, France
ACM. 978-1-4503-4660-3/17/01...$15.00
http://dx.doi.org/10.1145/3009837.3009852

This paper contributes a new inference algorithm, which computes a posterior distribution by symbolic manipulation of the prior distribution and an observable expression. Derived from the statistical notion of *disintegration* (Chang and Pollard 1997), our *automatic disintegrator* solves a longstanding problem in probabilistic programming: it can draw exact inferences from the observation of a continuous quantity whose probability is zero. Such observations are common in practice: they include any observation of a value drawn from a uniform or normal distribution, as well as combinations such as sums of such values. The latter use case is not supported by existing inference algorithms for probabilistic languages.

Our disintegrator represents the prior, the observation, and even the posterior as terms in the probabilistic language *core Hakaru* (pronounced Hah-KAH-roo). The disintegrator is a 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. Because this transformation preserves semantics, it can be applied repeatedly in a pipeline that turns a generative model into an efficient inference procedure. The full pipeline, which includes such other inference methods as symbolic simplification, Metropolis-Hastings sampling, and Gibbs sampling, is outside the scope of this paper, but experiments with it on a variety of distributions show performance that is competitive with hand-written code.

To use our disintegrator, a programmer represents a prior as a sequence of bindings, each of which binds a variable to the result of a computation in the monad of measures. The programmer then writes an observation, which refers to the bound variables. But the observation is not expressed as a traditional predicate or likelihood function. Instead—a key idea—the programmer writes an *expression* describing the quantity to be observed. The disintegrator extends and transforms the prior so that the observable quantity is bound *first*. The remaining bindings then denote a function from observation to posterior.

Like a partial evaluator, the disintegrator produces a residual program. It preserves semantics by rewriting the program using just enough computer algebra to change variables in simple integrals. Like a lazy evaluator, the disintegrator puts bindings on a heap and orders their evaluation by demand rather than by source code. Whenever it reorders evaluation, it can produce a verification condition which identifies two integrals that have been exchanged. As long as the exchanges are correct, the disintegrator's output is correct.

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 with the problem: we introduce observation and inference, we show the difficulty 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 pose queries about distributions. 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 section we study distributions over pairs (x,y) of real numbers; we study 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 intuitive, but too implicit for our work—for example, it doesn't say with respect to what distribution we are taking the expectation. We write commonly used informal notations in gray.

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

$$\begin{aligned} m_{1a} \triangleq \mathbf{do}\ \{&x \leftsquigarrow \mathbf{uniform}\ 0\ 1; \\ &y \leftsquigarrow \mathbf{uniform}\ 0\ 1; \\ &\mathbf{return}\ (x,y)\} \end{aligned} \tag{1}$$

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 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 explicit $E_{m_{1a}}(\lambda(x,y).x)$. The expectation of a function f is the ratio of two integrals: the integral of f and the integral of the constant 1 function.[1]

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

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

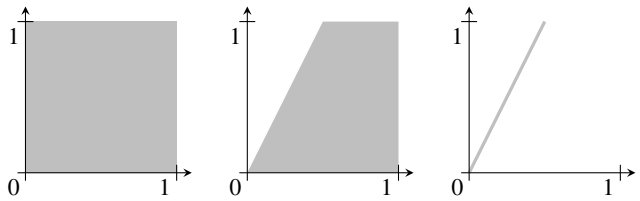
$$\begin{aligned} E_{m_{1b}}(\lambda(x,y).x) &= \frac{\int_{\{(x,y)\in[0,1]\times[0,1]\,|\,y\le 2\cdot x\}} x\,d(x,y)}{\int_{\{(x,y)\in[0,1]\times[0,1]\,|\,y\le 2\cdot x\}} 1\,d(x,y)} \\ &= \frac{1/12+3/8}{1/4+1/2} = \frac{11/24}{3/4} = 11/18. \end{aligned} \tag{3}$$

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

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

[1] Expectation is defined whenever the latter integral is finite and nonzero. The usual definition from probability theory assumes this integral to be 1.

[2] 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 $2\cdot x$.



(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

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.

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:

$$\Pr(A \cap B) = \Pr(A) \cdot \Pr(B \,|\, A). \tag{5}$$

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 $\{(x,y) \mid y \le 2\cdot x\}$.

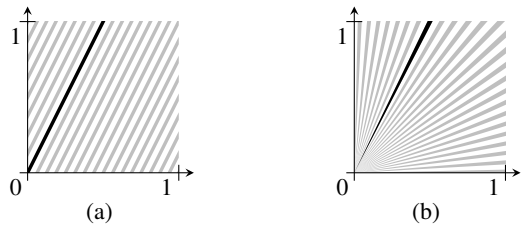
  Set B represents a *query*; for example, we might ask for the probability that $x > 2/3$ or for the probability that the point (x,y) falls within distance 1 of the origin. To keep the analogy simple, we ask only *probability* queries; the probability of a set B is the expectation of the set's characteristic function χ_B, which is 1 on points in B and 0 on points outside 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.
- $\Pr(B \,|\, A)$ represents a query against the posterior; it's the unknown we're trying to compute. To compute it, we solve (5); the solution gives $\Pr(B \,|\, A)$ 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 $0 = 0 \cdot \Pr(B \,|\, A)$, so we can't solve for $\Pr(B \,|\, A)$. 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 this example from a paradox discussed by Bertrand (1889), Borel (1909), Kolmogorov (1933), and Gupta, Jagadeesan, and Panangaden (1999). There is no single posterior distribution and 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? Because it doesn't tell us enough about the observation. To observe a set of measure zero within the unit square, we fill the unit square with an (uncountable) *family* of sets of measure zero, then identify the observed set as a member of the family.



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

(b) 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. Borel's paradox: Two ways to observe $y = 2 \cdot x$

The decomposition of the square into the family determines how probability mass is distributed over each member of the family. Figure 2 depicts two decompositions that yield different answers.

In Figure 2(a), the square is decomposed into a family of parallel line segments, each with slope 2, and each characterized by a real y-intercept between -2 and 1. This family is *indexed* by the intercept $y - 2 \cdot x$; our observation is indexed by intercept 0. And in each member of the family, probability mass is distributed uniformly, as can be calculated by examining the line segments whose intercepts lie within ε of 0. These line segments form a subset of the unit square with nonzero measure—a "strip" around the observation, whose midpoint tends to $(1/4, 1/2)$ as ε tends to 0.

In Figure 2(b), the square is decomposed into a family of line segments that radiate from the origin, each with y-intercept 0, and each characterized by a real slope between 0 and ∞. This family is indexed by the slope y/x; our observation is indexed by slope 2. And in each member of the family, probability mass is distributed more heavily on points that are further from the origin, as can be calculated by examining the line segments whose slopes lie within ε of 2. These line segments form a subset of the unit square with nonzero measure—a "wedge" around the observation, whose midpoint tends to $(1/3, 2/3)$ as ε tends to 0.

Neither $1/4$ nor $1/3$ is the "right" $E(x)$; a key idea of this paper is that we can't just 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: by intercept, or by slope? To say how, we use *disintegration*, an established notion from statistics (Chang and Pollard 1997).

2.3 Resolving the Paradox via Disintegration

We first present disintegration in general, then apply it to the example from Figure 2.

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, μ can be the uniform distribution over the unit interval and κ can 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 $\xi = \mu \otimes \kappa$. The decomposition looks like the conditioning equation (5): μ represents a distribution $\Pr(A)$, and κ represents the conditional probability $\Pr(B \mid A)$. Their product is the joint distribution $\Pr(A, B)$. Accordingly, the kernel κ is also called a *regular conditional distribution given A*.

We think of κ as a family of measures $\{\kappa(a) \mid a \in \alpha\}$ indexed by the observable outcome a. To infer a posterior distribution from the observation a, it's enough to apply κ to a. To reconstruct ξ, we also need μ, which tells us how to weight the members of the family κ.

Disintegrating our paradoxical example To use disintegration to compute a posterior distribution corresponding to Figure 1(c), we change the coordinate system to express the observation $y = 2 \cdot x$ as an observation of a single variable. (A change of variables involves 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:

$$t = y - 2 \cdot x \qquad\qquad u = 2 \cdot y + x \tag{6}$$

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 (x, y) coordinates to polar (θ, r) coordinates:

$$\theta = \arctan \frac{y}{x} \qquad\qquad r = \sqrt{x^2 + y^2} \tag{7}$$

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

Changing the coordinates of a measure is tedious, but our new program transformation eliminates the tedium. As programmers, all we do is write the *one* expression we plan to observe, using the variables we already have. This observable expression is what we want to index the family of measures. We extend our model with the observable expression on the left. For example, we don't *change* (x, y) to (t, u); instead we 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 $\alpha \times \beta$. Measurable space α has the type of the index expression that we intend to observe; in this paper we consider $\alpha = \mathbb{R}$. Space β has the type of our original model. Here's an extended model for Figure 2(a):

$$\begin{aligned} \overleftarrow{m}_{2\mathrm{a}} \triangleq \mathbf{do}\ \{&x \leftsquigarrow \mathbf{uniform}\ 0\ 1; \\ &y \leftsquigarrow \mathbf{uniform}\ 0\ 1; \\ &\mathbf{let}\ t = y - 2 \cdot x; \\ &\mathbf{return}\ (t, (x, y))\} \end{aligned} \tag{8}$$

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:

$$\overrightarrow{m}_{2\mathrm{a}} = \mathbf{do}\ \{t \leftsquigarrow m;\ p \leftsquigarrow M;\ \mathbf{return}\ (t, p)\} \tag{9}$$

The subterms m and M are found by the disintegrator, which is explained in Section 5 using this example. Crucially, m is independent of the variables of the model; it is closed and denotes a measure on t. By contrast, M typically depends on t, and it denotes a function from t to a measure—that is, a kernel. Hence for a model in the new 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.

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

$$\vec{m}_{2a} \triangleq \textbf{do}\ \{t \leftsquigarrow \textbf{lebesgue};\ p \leftsquigarrow \boxed{\begin{array}{l}\textbf{do}\ \{x \leftsquigarrow \textbf{uniform}\ 0\ 1;\\ \quad \textbf{observe}\ 0 \le t + 2 \cdot x \le 1;\\ \quad \textbf{return}\ (x, t + 2 \cdot x)\}\end{array}};\ \textbf{return}\ (t, p)\} \tag{10}$$

In this model, $m = \textbf{lebesgue}$ is the Lebesgue measure on the real line, and M is the boxed term. The disintegrator guarantees that $\vec{m}_{2a}$ is equivalent to $\overleftarrow{m}_{2a}$, which is to say that the measure denoted by m and the kernel denoted by M together disintegrate the measure denoted by $\overleftarrow{m}_{2a}$. To ensure this equivalence, the disintegrator has rewritten y as $t + 2 \cdot x$, in a sense changing (x, y) coordinates to (t, x).

To infer a posterior from the observation that t is 0, we substitute 0 for t in M:

$$m_{2a} \triangleq M[t \mapsto 0] = \textbf{do}\ \{x \leftsquigarrow \textbf{uniform}\ 0\ 1;\ \textbf{observe}\ 0 \le 0 + 2 \cdot x \le 1;\ \textbf{return}\ (x, 0 + 2 \cdot x)\} \tag{11}$$

Unlike the predicate $y = 2 \cdot x$, the predicate $0 \le 0 + 2 \cdot x \le 1$ describes a set of nonzero measure. Therefore, when we use this posterior to integrate $\lambda(x,y).x$ and the constant 1 function, we calculate the expectation $E_{m_{2a}}(\lambda(x,y).x) = 1/4$ as a ratio of nonzero numbers.

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. Using the ratio keeps the terms simple.

$$\overleftarrow{m}_{2b} \triangleq \textbf{do}\ \{x \leftsquigarrow \textbf{uniform}\ 0\ 1;\ y \leftsquigarrow \textbf{uniform}\ 0\ 1;\ \textbf{let}\ s = y/x;\ \textbf{return}\ (s, (x, y))\} \tag{12}$$

The automatic disintegrator finds this equivalent prior:

$$\vec{m}_{2b} \triangleq \textbf{do}\ \{s \leftsquigarrow \textbf{lebesgue};\ p \leftsquigarrow \boxed{\begin{array}{l}\textbf{do}\ \{x \leftsquigarrow \textbf{uniform}\ 0\ 1;\\ \quad \textbf{observe}\ 0 \le s \cdot x \le 1;\\ \quad \textbf{factor}\ x;\\ \quad \textbf{return}\ (x, s \cdot x)\}\end{array}};\ \textbf{return}\ (s, p)\} \tag{13}$$

The **factor** x in the boxed term M 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 s is 2, we substitute 2 for s in M:

$$m_{2b} \triangleq M[s \mapsto 2] = \textbf{do}\ \{x \leftsquigarrow \textbf{uniform}\ 0\ 1;\ \textbf{observe}\ 0 \le 2 \cdot x \le 1;\ \textbf{factor}\ x;\ \textbf{return}\ (x, 2 \cdot x)\} \tag{14}$$

When we use this posterior to integrate $\lambda(x,y).x$ and the constant 1 function, we calculate the expectation $E_{m_{2b}}(\lambda(x,y).x) = 1/3$.

2.5 From Example to Algorithm

From this example we work up to our disintegration algorithm. The algorithm preserves the semantics of terms, so before presenting the algorithm, we first describe the semantic foundations, 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 in Section 3.1 with *measure*, and in particular, a *measure function*. "Measure" generalizes "probability," so it is closest to the intuition of probabilistic programmers.

In Section 3.2 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 integrating f and integrating the constant 1 function.

An integrator may feel more powerful than a measure function, and perhaps less familiar, but they are equivalent. 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.

In Section 3.3, we discuss the properties of disintegrations that justify our use of **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 that a measure and a measure function are not the same thing—please think of "measure" as an abstract data type, whose possible implementations include "measure function."

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 call $[\![\textbf{uniform}\ 0\ 1]\!]_M$, assigns a measure to many sets, including every interval on the real line. Under $[\![\textbf{uniform}\ 0\ 1]\!]_M$, the measure of any interval is the length of the part of the interval that intersects with $[0,1]$. For example, $[\![\textbf{uniform}\ 0\ 1]\!]_M[2/3, 2] = 1/3$, 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 obtained as a countable linear combination of Dirac measures.

Measurable spaces and measurable sets We've been coy about what sets $[\![\textbf{uniform}\ 0\ 1]\!]_M$ 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 won't 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. In particular, $\{\}$ and $\alpha_{\mathcal{S}}$ must be in $\alpha_{\mathcal{M}}$. 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 $\alpha_{\mathcal{M}}$ 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 $\{S_1, S_2, \dots\} \subseteq \alpha_{\mathcal{M}}$,

$$\mu(S_1 \cup S_2 \cup \cdots) = \mu(S_1) + \mu(S_2) + \cdots. \tag{15}$$

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 to base types and type constructors that are 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 $a \in \alpha$, $f(a) \leq g(a)$. (And the cpo has a bottom element: the constant zero function. That is, f must be nonnegative.) 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 $\mu(f)$, just as we often treat μ as a measure function and write the measure of a set S as $\mu(S)$ (de Finetti 1974; Pollard 2001). To distinguish the measure function from the integrator, we write the measure function as $[\![\mu]\!]_M$, so the measure of S is $[\![\mu]\!]_M(S)$, and we write the integrator as $[\![\mu]\!]_I$, so the integral of f is $[\![\mu]\!]_I(f)$.

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

$$\chi_S(a) \triangleq \begin{cases} 1 & \text{if } a \in S, \\ 0 & \text{if } a \notin S. \end{cases} \tag{16}$$

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

$$[\![\mu]\!]_I(\chi_S) \triangleq [\![\mu]\!]_M(S). \tag{17}$$

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

$$[\![\mu]\!]_I(\lambda a.\, r \cdot f(a)) \triangleq r \cdot [\![\mu]\!]_I(f), \tag{18}$$

$$[\![\mu]\!]_I(\lambda a.\, f(a) + g(a)) \triangleq [\![\mu]\!]_I(f) + [\![\mu]\!]_I(g). \tag{19}$$

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:

$$[\![\mu]\!]_I(\lambda a.\, \lim_{n\to\infty} f_n(a)) \triangleq \lim_{n\to\infty} [\![\mu]\!]_I(f_n) \tag{20}$$

Every measurable function can be approximated from below by a sequence of simple functions, and as in Scott's domain theory, equations (17) to (20) determine $[\![\mu]\!]_I$ uniquely; given $[\![\mu]\!]_M$, there is one and only one function $[\![\mu]\!]_I$ satisfying these equations. Conversely, given $[\![\mu]\!]_I$, equation (17) determines $[\![\mu]\!]_M$ uniquely. So integrators, defined by equations (18) to (20), are in one-to-one correspondence with measure functions. In other words, integrators constitute an alternative—and for us more convenient—implementation of the abstract data type of measures.

The convenience begins with the fact that many useful measures are most easily defined by specifying their integrators. For example, the uniform measure is defined by integration over an interval, and the Dirac measure is defined by function application:

$$[\![\textbf{uniform}\ 0\ 1]\!]_I f = \int_{[0,1]} f(x)\, dx \tag{21}$$

$$[\![\textbf{return}\ a]\!]_I f = f(a) \tag{22}$$

Integrator notation clarifies the relationship between the monad of measures and the monad of continuations. The Dirac measure just defined is the monadic unit, and here is the monadic bind:

$$[\![\mu \gg\!= \kappa]\!]_I f = [\![\mu]\!]_I(\lambda a.\, [\![\kappa\ a]\!]_I f) \tag{23}$$

It is also convenient to express disintegration using integrators. If $\mu \in \mathbb{M}\,\alpha$ and $\kappa \in (\alpha \to \mathbb{M}\,\beta)$, then $\mu \otimes \kappa \in \mathbb{M}\,(\alpha \times \beta)$ is defined by an iterated integral:

$$[\![\mu \otimes \kappa]\!]_I f \triangleq [\![\mu]\!]_I(\lambda a.\, [\![\kappa\ a]\!]_I(\lambda b.\, f(a,b))). \tag{24}$$

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:

$$\int f\, d(\mu \otimes \kappa) = \iint f(a,b)\, d(\kappa(a))(b)\, d\mu(a). \tag{25}$$

One more example: if ξ is the uniform probability distribution over the unit circle, then we can achieve $\xi = \mu \otimes \kappa$ by defining

$$[\![\mu]\!]_I f = \frac{1}{\pi} \cdot \int_{[-1,1]} f(x)\, dx, \quad [\![\kappa\ x]\!]_I f = \int_{[-\sqrt{1-x^2}, \sqrt{1-x^2}]} f(y)\, dy. \tag{26}$$

Real numbers	$r \in \mathbb{R}$
Variables	x, y, z
Atomic terms	$u ::= z$ (not bound in heap) $\mid -u \mid u^{-1} \mid u+u \mid u+r \mid r+u \mid u \cdot u \mid u \cdot r \mid r \cdot u \mid u \le u \mid u \le r \mid r \le u \mid \mathbf{fst}\ u \mid \mathbf{snd}\ u$
Bindings (guards)	$g ::= x \leftsquigarrow m \mid \mathbf{let\ inl}\ x = e \mid \mathbf{let\ inr}\ x = e \mid \mathbf{factor}\ e$
Head normal forms	$v ::= u \mid \mathbf{do}\ \{g; M\} \mid \mathbf{return}\ e \mid \mathbf{fail} \mid \mathbf{mplus}\ m_1\ m_2 \mid r \mid () \mid (e_1, e_2) \mid \mathbf{inl}\ e \mid \mathbf{inr}\ e \mid \mathbf{lebesgue} \mid \mathbf{uniform}\ r_1\ r_2$
Terms	$e, m, M ::= v \mid x \mid -e \mid e^{-1} \mid e+e \mid e \cdot e \mid e \le e \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e$

Figure 3. Syntactic forms of core Hakaru

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

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

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

$$\frac{}{\Gamma \vdash \mathbf{fail} : \mathbb{M}\,\alpha} \qquad \frac{\Gamma \vdash m_1 : \mathbb{M}\,\alpha \quad \Gamma \vdash m_2 : \mathbb{M}\,\alpha}{\Gamma \vdash \mathbf{mplus}\ m_1\ m_2 : \mathbb{M}\,\alpha}$$

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

Figure 4. Typing rules for terms of measure type

$$\begin{aligned}
[\![\mathbf{do}\ \{x \leftsquigarrow m; M\}]\!]_I \rho f &= [\![m]\!]_I \rho (\lambda a.\, [\![M]\!]_I (\rho[x \mapsto a]) f) \\
[\![\mathbf{do}\ \{\mathbf{let\ inl}\ x = e; M\}]\!]_I \rho f &= [\![M]\!]_I (\rho[x \mapsto a]) f && \text{if } [\![e]\!]\rho = \mathbf{inl}\ a \\
[\![\mathbf{do}\ \{\mathbf{let\ inl}\ x = e; M\}]\!]_I \rho f &= 0 && \text{if } [\![e]\!]\rho = \mathbf{inr}\ b \\
[\![\mathbf{do}\ \{\mathbf{factor}\ e; M\}]\!]_I \rho f &= [\![e]\!]\rho \cdot [\![M]\!]_I \rho f && \text{if } [\![e]\!]\rho \ge 0 \\
[\![\mathbf{return}\ e]\!]_I \rho f &= f([\![e]\!]\rho) \\
[\![\mathbf{fail}]\!]_I \rho f &= 0 \\
[\![\mathbf{mplus}\ m_1\ m_2]\!]_I \rho f &= [\![m_1]\!]_I \rho f + [\![m_2]\!]_I \rho f \\
[\![\mathbf{lebesgue}]\!]_I \rho f &= \textstyle\int_{\mathbb{R}} f(x)\,dx \\
[\![\mathbf{uniform}\ r_1\ r_2]\!]_I \rho f &= \tfrac{1}{r_2 - r_1} \cdot \textstyle\int_{[r_1, r_2]} f(x)\,dx
\end{aligned}$$

Figure 5. Denotations of terms of measure type

3.3 Existence and Uniqueness of Disintegrations

Disintegrations do not always exist (Dieudonné 1947–1948), and when they do exist, they are never unique. Disintegrability of a measure ξ over a product space $\alpha \times \beta$ can be guaranteed by any of a variety of side conditions offered by the mathematics literature (Chang and Pollard 1997), such as being a probability measure on a Polish space. Uniqueness can never be guaranteed; if $\xi = \mu \otimes \kappa$, we can always obtain another disintegration by, for example, doubling μ and halving κ. But the properties we are interested in, such as expectation or most likely outcome, are invariant under scaling, so this 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 zero measure to every set of Lebesgue measure zero; it is equivalent to saying that μ is described by a probability-density function. Because most observation distributions μ over $\mathbb{R}$ that arise naturally in probabilistic programs 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 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.

Types as measurable spaces Core Hakaru has standard unit ($\mathbb{1}$), pair ($\times$), and sum ($+$) types; it also has a real-number type ($\mathbb{R}$) and measure types ($\mathbb{M}$). We write types as $\alpha, \beta, \gamma, \ldots$, where

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

Each type 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, which muddies the presentation. 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 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 embeddings of 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 each integrand f:

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

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), function types aren't first-class. Full Hakaru does have λ-abstraction and function types, but we don't put function types under an $\mathbb{M}$ constructor.

Syntax The syntactic forms associated with $\mathbb{1}$, pairs, sums, and reals are standard—except the elimination form for sums. There is no primitive **case** form; as shown below, **case** desugars to effectful elimination forms that interpret pattern-matching failure as the zero measure (**fail**). Core Hakaru needs **fail** anyway, and by encoding conditionals using failure, we reduce the number of syntactic forms the disintegrator must 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* used in core Hakaru's **do** notation. The binding form $x \leftsquigarrow m$ is the classic binding in the monad of measures (or the probability monad); as shown in Figure 4, m 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, as detailed in Figure 5. Finally, the **factor** form weights the measure by a real scaling factor, which can be thought of as a probability density. These four forms all have effects in the monad of measures: probabilistic choice, failure, or weighting the measure.

Atomic terms are a subset of *head normal forms*, which are in turn a subset of all terms. As is standard in lazy languages, a head normal form v is an application of a known constructor. Besides atomic terms, head normal forms include 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 $[r_1, r_2]$. Besides head normal forms and variables, terms e include applications of the same strict functions used in the definition of atomic terms.

Typing The typing rules of core Hakaru are unsurprising; Figure 4 simply formalizes what we say informally about the terms. Figure 4 shows rules only for measure type; rules for other types and for **let inr** 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 x onto values $\rho(x)$ 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 how it applies to an integrand f.

- A term **do** $\{x \leftsquigarrow m; M\}$ denotes (roughly) the integral of the function $\lambda x.M$ with respect to the measure denoted by m.
- Each of our unusual sum-elimination forms acts either as a **let**-binding or as the zero measure, depending on the value of the right-hand side. The equations for **let inr** are omitted.
- The next 4 forms are not so interesting: **factor** scales the integral by a 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, so function $\lambda a.\cdots$ in the first line of Figure 5 is also measurable.

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

$$\textbf{true} \triangleq \textbf{inl}\ () \qquad \textbf{false} \triangleq \textbf{inr}\ () \qquad \textbf{observe}\ e \triangleq \textbf{let inl}\ \_ = e$$
$$e_1 - e_2 \triangleq e_1 + (-e_2) \qquad \textbf{let}\ x = e \triangleq x \leftsquigarrow \textbf{return}\ e \tag{30}$$

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

$$\textbf{case}\ e\ \textbf{of inl}\ x_1 \Rightarrow m_1 \mid \textbf{inr}\ x_2 \Rightarrow m_2 \tag{31}$$
$$\triangleq \textbf{mplus}\ (\textbf{do}\ \{\textbf{let inl}\ x_1 = e; m_1\})\ (\textbf{do}\ \{\textbf{let inr}\ x_2 = e; m_2\})$$

Full Hakaru desugars **case** in any context, by bubbling **case** up 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

$$\textbf{do}\ \{[]; M\} \triangleq M, \tag{33}$$
$$\textbf{do}\ \{[g_1; g_2; \ldots; g_n]; M\} \triangleq \textbf{do}\ \{g_1; \textbf{do}\ \{[g_2; \ldots; g_n]; M\}\}. \tag{34}$$

As detailed below, 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:

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

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 term of the form

$$\vec{m} = \textbf{do}\ \{t \leftsquigarrow m; p \leftsquigarrow M; \textbf{return}\ (t, p)\}, \tag{36}$$

where $m : \mathbb{M}\ \alpha$ and $M : \mathbb{M}\ \beta$. In this paper we focus on the continuous case where $\alpha = \mathbb{R}$ and $m = \textbf{lebesgue}$. Other types α such as $\alpha = \mathbb{R} \times \mathbb{R}$ can be handled by successive disintegration. And as discussed in Section 3.3, $m = \textbf{lebesgue}$ 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 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 $\vec{m}_{2a}$ in equation (8):

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

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

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

We calculate a disintegration by rewriting these integrals, but first we cast them into integrator notation. The Lebesgue integral over a set $S \subseteq \mathbb{R}$ is defined by

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

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

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

This final right-hand-side matches that of (24). The index a there is t here and the integrand $\lambda b.\, f(a,b)$ there is f_p here:

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

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

When we convert μ and κ back to terms of core Hakaru, they match the desired form (36). The result is exactly (10):

$$m = \textbf{lebesgue} \tag{44}$$

$$\begin{aligned} M = \textbf{do}\ \{x \leftsquigarrow \textbf{uniform}\ 0\ 1; \\ \textbf{observe}\ 0 \le t + 2\cdot x \le 1; \\ \textbf{return}\ (x, t+2\cdot x)\}. \end{aligned} \tag{45}$$

Derivations of this kind use two transformations again and again:

- *Change the integration variable* to the observable expression, as in (40). Many of us haven't studied calculus for a long time, but if you want to revisit it, changing variables from y to $t = g(y)$ requires that g be both invertible and differentiable. An integral over y is transformed into an integral over t using the inverse g^{-1} and its derivative $(g^{-1})'$:

$$[\![\Lambda]\!]_I\big(\lambda y.\, f(y)\big) = [\![\Lambda]\!]_I\big(\lambda t.\, |(g^{-1})'(t)| \cdot f(g^{-1}(t))\big). \tag{46}$$

- *Reorder integrals* to move the observable variable to the outermost position, as in (41), justified by Tonelli's theorem.

5.1 Disintegration by Lazy Partial Evaluation

To reorder integrals, we use lazy partial evaluation (Fischer et al. 2008). In the terminology of partial evaluation, the observable variable t and expressions that depend on it are *dynamic*; the disintegrator treats them as opaque values and does not inspect them. But variables that are $\leftsquigarrow$-bound or **let**-bound, and expressions that depend only on them, are *static*; the disintegrator may 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).

The heap can be thought of as embodying "random choices" $\mathcal{R}$ made by a probabilistic abstract machine that evaluates core Hakaru terms. In *generative* mode, the machine is given random choices $\mathcal{R}$ and a term m of measure type, and it produces a value v:

$$m \overset{\mathcal{R}}{\rightsquigarrow} \textbf{return}\ v. \tag{47}$$

But this evaluation relation can used in two other modes. In *backward* mode, a machine is given m and v, and it rewrites the heap, constraining random choices, to ensure the outcome is v. In *forward* mode, a machine is given m, and it extends the heap to produce an outcome v.

The abstract-machine metaphor can be generalized to explain disintegration. The disintegrator does not evaluate terms to produce values—it *partially* evaluates terms, in backward and forward modes, to produce *code*. Moreover, randomized computation is lazy but not pure, and evaluating terms *not* of measure type can *also* extend the heap or constrain its random choices. So the disintegrator processes general terms using the same two modes. In sum, the disintegrator partially evaluates terms m of measure type and e of any type, in two modes each, by means of four functions:

$\lll m\ v$	Put m's bindings on the heap and constrain its final action to produce v.
$\lhd\ e\ v$	Constrain e to evaluate to the head normal form v by fixing the outcome of a choice on the heap.
$\rrbracket\ m$	Put m's bindings on the heap and emit code to perform its final action. Put its outcome in head normal form.
$\rhd\ e$	Emit code to evaluate e. Put it in head normal form.

The functions are pronounced "constrain outcome" ($\lll$), "constrain value" ($\lhd$), "perform" ($\rrbracket$), and "evaluate" ($\rhd$). 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.

5.2 Specification and Definition of the Disintegrator

To interleave heap updates with code emission, we use continuation-passing style (Bondorf 1992; Lawall and Danvy 1994; Danvy, Malmkjær, and Palsberg 1996). A continuation takes a heap and returns a residual program, or informally

$$C = \mathit{heap} \to \lfloor \mathbb{M}\ \gamma \rfloor, \tag{86}$$

in which $\lfloor \mathbb{M}\ \gamma \rfloor$ means head normal forms of type $\mathbb{M}\ \gamma$. Using this continuation 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, Figure 6 defines the functions

$$\lll \text{("constrain outcome")} : \lceil \mathbb{M}\ \mathbb{R} \rceil \to \lfloor \mathbb{R} \rfloor \to C \to C \tag{87}$$

$$\lhd \text{("constrain value")} : \lceil \mathbb{R} \rceil \to \lfloor \mathbb{R} \rfloor \to C \to C \tag{88}$$

$$\rrbracket \text{("perform")} : \lceil \mathbb{M}\ \alpha \rceil \to (\lfloor \alpha \rfloor \to C) \to C \tag{89}$$

$$\rhd \text{("evaluate")} : \lceil \alpha \rceil \to (\lfloor \alpha \rfloor \to C) \to C \tag{90}$$

The backward functions Functions $\lll$ and $\lhd$ embody our technical contribution. A disintegration problem begins with a monadic term that binds the variable t somewhere inside, and seeks to "pull out" the binding of t to the top of the term, so the result has the form $\textbf{do}\ \{t \leftsquigarrow \textbf{lebesgue}; \cdots\}$ (36). Depending on whether t starts out bound probabilistically or deterministically, we pull t out using either $\lll$ or $\lhd$, which obeys these laws:

$$\textbf{do}\ \{h;\quad t \leftsquigarrow m; M\} \equiv \textbf{do}\ \{t \leftsquigarrow \textbf{lebesgue}; \lll m\ t\ \overline{M}\ h\} \tag{91}$$

$$\textbf{do}\ \{h; \textbf{let}\ t = e; M\} \equiv \textbf{do}\ \{t \leftsquigarrow \textbf{lebesgue};\ \lhd e\ \ t\ \overline{M}\ h\} \tag{92}$$

Equivalence $\equiv$ on terms means they denote the same integrator. The final action M, which expresses the posterior distribution to infer, becomes the continuation $\overline{M}$ passed to $\lll$ or $\lhd$. The notation $\overline{M}$ expresses a frequently used form of continuation: a measure term M lifts to the continuation $\overline{M} \triangleq \lambda h.\, \textbf{do}\ \{h; M\}$, which materializes a given heap h by wrapping it around M. (Recall that core Hakaru has no first-class functions, so the λ here, like all the λs in Figure 6, are in the metalanguage of our disintegrator.)

Functions $\lll$ and $\lhd$, which obey a few additional laws mentioned below, deconstruct m and e and rewrite the heap h. Each function ends by passing to $\overline{M}$ a final heap h' that binds at least all the variables bound by the initial heap h, in the same order, and possibly new variables as well. Most likely some variable that is bound *probabilistically* in the initial heap h is bound *deterministically* in the final heap h', to a quantity expressed in terms of the observable t.

Implementations of $\lll$ and $\lhd$ (Figure 6) are derived from specifications (91) and (92). For example, when $m = \textbf{lebesgue}$, (91) requires

$$\textbf{do}\ \{h; t \leftsquigarrow \textbf{lebesgue}; M\} \equiv \textbf{do}\ \{t \leftsquigarrow \textbf{lebesgue}; \lll m\ t\ \overline{M}\ h\}. \tag{93}$$

Because $\lll m$ may be applied only to a real t that is in head normal form (87), and a heap-bound variable is not in head normal form, we know h does not bind t. By Tonelli's theorem, we can change the order of integration, lifting the binding of t:

$$\textbf{do}\ \{h; t \leftsquigarrow \textbf{lebesgue}; M\} \equiv \textbf{do}\ \{t \leftsquigarrow \textbf{lebesgue}; h; M\}. \tag{94}$$

Matching (93) yields $\lll \textbf{lebesgue}\ t\ \overline{M}\ h = \textbf{do}\ \{h; M\} = \overline{M}\ h$, which is case 49 in Figure 6. Case 50 is derived similarly.

Figure 6 is notated using quasiquotation brackets $[\![\cdots]\!]$ (Stoy 1977, Chapter 3). Inside the brackets, juxtaposition constructs syntax; outside, juxtaposition applies a function in the metalanguage.

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

$\ll [\![u]\!]\ v\ c\ h = \bot$ where u is atomic (48)

$\ll [\![\mathbf{lebesgue}]\!]\ v\ c\ h = c\ h$ (49)

$\ll [\![\mathbf{uniform}\ r_1\ r_2]\!]\ v\ c\ h = [\![\mathbf{do}\ \{\mathbf{observe}\ \$(r_1 \le v \le r_2); \mathbf{factor}\ \$((r_2 - r_1)^{-1}); \$(c\ h)\}]\!]$ (50)

$\ll [\![\mathbf{return}\ e]\!]\ v\ c\ h = \lhd\ [\![e]\!]\ v\ c\ h$ (51)

$\ll [\![\mathbf{do}\ \{g; m\}]\!]\ v\ c\ h = \ll [\![m]\!]\ v\ c\ [h; [\![g]\!]]$ unless g binds a variable in h (52)

$\ll [\![\mathbf{fail}]\!]\ v\ c\ h = [\![\mathbf{fail}]\!]$ (53)

$\ll [\![\mathbf{mplus}\ m_1\ m_2]\!]\ v\ c\ h = [\![\mathbf{mplus}\ \$(\ll [\![m_1]\!]\ v\ c\ h)\ \$(\ll [\![m_2]\!]\ v\ c\ h)]\!]$ (54)

$\ll [\![e]\!]\ v\ c\ h = \rhd\ [\![e]\!]\ (\lambda m.\ \ll m\ v\ c)\ h$ where e is not in head normal form (55)

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

$\lhd [\![u]\!]\ v\ c\ h = \bot$ where u is atomic (56)

$\lhd [\![r]\!]\ v\ c\ h = \bot$ where r is a literal real number (57)

$\lhd [\![\mathbf{fst}\ e_0]\!]\ v\ c\ h = \rhd\ [\![e_0]\!]\ (\lambda v_0.\ \lhd\ (\mathit{fst}\ v_0)\ v\ c)\ h$ unless e_0 is atomic (58)

$\lhd [\![\mathbf{snd}\ e_0]\!]\ v\ c\ h = \rhd\ [\![e_0]\!]\ (\lambda v_0.\ \lhd\ (\mathit{snd}\ v_0)\ v\ c)\ h$ unless e_0 is atomic (59)

$\lhd [\![-e_0]\!]\ v\ c\ h = \lhd\ [\![e_0]\!]\ (-v)\ c\ h$ (60)

$\lhd [\![e_0^{-1}]\!]\ v\ c\ h = [\![\mathbf{do}\ \{\mathbf{factor}\ \$((v \cdot v)^{-1}); \$(\lhd\ [\![e_0]\!]\ v^{-1}\ c\ h)\}]\!]$ (61)

$\lhd [\![e_1 + e_2]\!]\ v\ c\ h = \rhd\ [\![e_1]\!]\ (\lambda v_1.\ \lhd\ [\![e_2]\!]\ (v - v_1)\ c)\ h$ (62)
$\sqcup\ \rhd\ [\![e_2]\!]\ (\lambda v_2.\ \lhd\ [\![e_1]\!]\ (v - v_2)\ c)\ h$

$\lhd [\![e_1 \cdot e_2]\!]\ v\ c\ h = \rhd\ [\![e_1]\!]\ (\lambda v_1.\ \mathit{abs}\ v_1\ (\lambda v_1'.\ \lambda h'.\ [\![\mathbf{do}\ \{\mathbf{factor}\ \$(v_1'^{-1}); \$(\lhd\ [\![e_2]\!]\ (v \cdot v_1^{-1})\ c\ h')\}]\!]))\ h$ (63)
$\sqcup\ \rhd\ [\![e_2]\!]\ (\lambda v_2.\ \mathit{abs}\ v_2\ (\lambda v_2'.\ \lambda h'.\ [\![\mathbf{do}\ \{\mathbf{factor}\ \$(v_2'^{-1}); \$(\lhd\ [\![e_1]\!]\ (v \cdot v_2^{-1})\ c\ h')\}]\!]))\ h$

$\lhd [\![x]\!]\ v\ c\ [h_1; [\![x \leftsquigarrow m]\!]; h_2] = \ll [\![m]\!]\ v\ (\overline{[[\![\mathbf{let}\ x = v]\!]; h_2]}\ \fatsemi\ c)\quad h_1$ (64)

$\lhd [\![x]\!]\ v\ c\ [h_1; [\![\mathbf{let\ inl}\ x = e_0]\!]; h_2] = \rhd\ [\![e_0]\!]\ (\lambda v_0.\ \mathit{outl}\ v_0\ (\lambda e.\ \lhd\ e\ v\ (\overline{[[\![\mathbf{let}\ x = v]\!]; h_2]}\ \fatsemi\ c)))\ h_1$ (65)

$\lhd [\![x]\!]\ v\ c\ [h_1; [\![\mathbf{let\ inr}\ x = e_0]\!]; h_2] = \rhd\ [\![e_0]\!]\ (\lambda v_0.\ \mathit{outr}\ v_0\ (\lambda e.\ \lhd\ e\ v\ (\overline{[[\![\mathbf{let}\ x = v]\!]; h_2]}\ \fatsemi\ c)))\ h_1$ (66)

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

$\gg [\![u]\!]\ k\ h = [\![\mathbf{do}\ \{z \leftsquigarrow u;\ \$(k\ z\ h)\}]\!]$ where u is atomic, z is fresh (67)

$\gg [\![\mathbf{lebesgue}]\!]\ k\ h = [\![\mathbf{do}\ \{z \leftsquigarrow \mathbf{lebesgue};\ \$(k\ z\ h)\}]\!]$ where z is fresh (68)

$\gg [\![\mathbf{uniform}\ r_1\ r_2]\!]\ k\ h = [\![\mathbf{do}\ \{z \leftsquigarrow \mathbf{uniform}\ r_1\ r_2; \$(k\ z\ h)\}]\!]$ where z is fresh (69)

$\gg [\![\mathbf{return}\ e]\!]\ k\ h = \rhd\ [\![e]\!]\ k\ h$ (70)

$\gg [\![\mathbf{do}\ \{g; m\}]\!]\ k\ h = \gg [\![m]\!]\ k\ [h; [\![g]\!]]$ unless g binds a variable in h (71)

$\gg [\![\mathbf{fail}]\!]\ k\ h = [\![\mathbf{fail}]\!]$ (72)

$\gg [\![\mathbf{mplus}\ m_1\ m_2]\!]\ k\ h = [\![\mathbf{mplus}\ \$(\gg [\![m_1]\!]\ k\ h)\ \$(\gg [\![m_2]\!]\ k\ h)]\!]$ (73)

$\gg [\![e]\!]\ k\ h = \rhd\ [\![e]\!]\ (\lambda m.\ \gg m\ k)\ h$ where e is not in head normal form (74)

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

$\rhd [\![v]\!]\ k\ h = k\ [\![v]\!]\ h$ where v is in head normal form (75)

$\rhd [\![\mathbf{fst}\ e_0]\!]\ k\ h = \rhd\ [\![e_0]\!]\ (\lambda v_0.\ \rhd\ (\mathit{fst}\ v_0)\ k)\ h$ unless e_0 is atomic (76)

$\rhd [\![\mathbf{snd}\ e_0]\!]\ k\ h = \rhd\ [\![e_0]\!]\ (\lambda v_0.\ \rhd\ (\mathit{snd}\ v_0)\ k)\ h$ unless e_0 is atomic (77)

$\rhd [\![-e_0]\!]\ k\ h = \rhd\ [\![e_0]\!]\ (\lambda v_0.\ k\ (-v_0))\ h$ (78)

$\rhd [\![e_0^{-1}]\!]\ k\ h = \rhd\ [\![e_0]\!]\ (\lambda v_0.\ k\ (v_0^{-1}))\ h$ (79)

$\rhd [\![e_1 + e_2]\!]\ k\ h = \rhd\ [\![e_1]\!]\ (\lambda v_1.\ \rhd\ [\![e_2]\!]\ (\lambda v_2.\ k\ (v_1 + v_2)))\ h$ (80)

$\rhd [\![e_1 \cdot e_2]\!]\ k\ h = \rhd\ [\![e_1]\!]\ (\lambda v_1.\ \rhd\ [\![e_2]\!]\ (\lambda v_2.\ k\ (v_1 \cdot v_2)))\ h$ (81)

$\rhd [\![e_1 \le e_2]\!]\ k\ h = \rhd\ [\![e_1]\!]\ (\lambda v_1.\ \rhd\ [\![e_2]\!]\ (\lambda v_2.\ k\ (v_1 \le v_2)))\ h$ (82)

$\rhd [\![x]\!]\ k\ [h_1; [\![x \leftsquigarrow m]\!]; h_2] = \gg [\![m]\!]\ (\lambda v.\ \overline{[[\![\mathbf{let}\ x = v]\!]; h_2]}\ \fatsemi\ k\ v)\quad h_1$ (83)

$\rhd [\![x]\!]\ k\ [h_1; [\![\mathbf{let\ inl}\ x = e_0]\!]; h_2] = \rhd\ [\![e_0]\!]\ (\lambda v_0.\ \mathit{outl}\ v_0\ (\lambda e.\ \rhd\ e\ (\lambda v.\ \overline{[[\![\mathbf{let}\ x = v]\!]; h_2]}\ \fatsemi\ k\ v)))\ h_1$ (84)

$\rhd [\![x]\!]\ k\ [h_1; [\![\mathbf{let\ inr}\ x = e_0]\!]; h_2] = \rhd\ [\![e_0]\!]\ (\lambda v_0.\ \mathit{outr}\ v_0\ (\lambda e.\ \rhd\ e\ (\lambda v.\ \overline{[[\![\mathbf{let}\ x = v]\!]; h_2]}\ \fatsemi\ k\ v)))\ h_1$ (85)

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

But quasiquotation is most useful when augmented with antiquotation (Mainland 2007). Lacking a standard notation, we borrow the antiquotation notation $\$(\cdots)$ from the POSIX shell language. Within $\$(\cdots)$, juxtaposition once again means application in the metalanguage. And, again following Stoy, we quasiquote metalanguage variables in pattern matching and on right-hand sides.

After **lebesgue**, easily derived cases of $\lll$ include **do** $\{g; m\}$, **fail**, and **mplus** $m_1\ m_2$ (cases 52 to 54). To justify case 52, which moves a binding g onto the heap, requires the associativity law on the monad of core Hakaru terms, plus the induction hypothesis for $\lll m$. To justify case 54 requires measure additivity and the induction hypotheses for $\lll m_1$ and $\lll m_2$. As a final example, when m is **return** e, binding $t \leftsquigarrow$ **return** e is the definition of **let** $t = e$, and (92) indicates that we should call $\lhd$, to which we now turn.

The most fundamental application of $\lhd$, case 64 in Figure 6, is to a variable x that is probabilistically bound to a measure term m in heap $h = [h_1; [\![x \leftsquigarrow m]\!]; h_2]$. To constrain x to evaluate to v, we constrain m to produce v: we pass m and v to $\lll$, along with a carefully constructed continuation and heap. Only heap h_1, which precedes $[\![x \leftsquigarrow m]\!]$, is available to be rewritten by $\lll$. The new continuation wraps its argument around $[[\![\textbf{let}\ x = v]\!]; h_2]$ and then continues with c. That continuation is written using more new notation: when h_0 is a heap, $\overline{h_0}$ is the function $\lambda h.\,[h; h_0]$. This function is composed with continuation c using the reverse function-composition operator $\fatsemi$, which is defined by $(g \fatsemi f)(x) \triangleq f(g(x))$. Composition commutes with lifting: $\overline{h_0} \fatsemi \overline{M} = \overline{\textbf{do}\ \{h_0; M\}}$.

Case 64 rewrites a random choice $x \leftsquigarrow m$ into a binding **let** $x = v$, where v is a deterministic function of the observable t. In our running example (37), it rewrites the binding $y \leftsquigarrow$ **uniform** 0 1 to **let** $y = t + 2 \cdot z$. The case is shown correct by appealing to the induction hypothesis (91) for $\lll m$; the rest of the proof requires only alpha- and beta-conversions.

Many cases of $\lhd$ invert functions and change variables of integration. One of the simplest is $\lhd\ (-e_0)$. The derivation begins with the left side of (92) where $e = -e_0$. We expand the binding **let** $t = -e_0$ into two bindings **let** $s = e$; **let** $t = -s$, where s is a fresh variable. We apply the induction hypothesis for $\lhd\ e_0\ s$, eventually arriving at

$$\textbf{do}\ \{s \leftsquigarrow \textbf{lebesgue}; \textbf{let}\ t = -s; \lhd\ e_0\ s\ \overline{M}\ h\}, \tag{95}$$

then change the integration variable from s to t to get

$$\textbf{do}\ \{t \leftsquigarrow \textbf{lebesgue}; \textbf{let}\ s = -t; \lhd\ e_0\ s\ \overline{M}\ h\}. \tag{96}$$

At this point we exploit a parametricity property of $\lhd$, given in (104) below, to substitute $-t$ for s. We end up with the definition in case 60, namely $\lhd\ (-e_0)\ v \triangleq \lhd\ e_0\ (-v)$.

On the left of case 60, the $-$ sign is object-language syntax. But on the right, the $-$ sign is a *smart constructor*, which in special cases can simplify its argument. We use one smart constructor for each strict function in core Hakaru. (These are the functions that define atomic terms in Figure 3.) As examples, applying $-$ to a real literal produces a real literal, and applying *fst* to (e_1, e_2) produces e_1. Applying $-$ or *fst* to any other e produces $-e$ or **fst** e.

But 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 for a fresh object variable x:

$$\begin{aligned}
&\mathit{outl}\colon \lfloor \alpha + \beta \rfloor \to (\lceil \alpha \rceil \to \mathit{heap} \to \lfloor \mathbb{M}\ \gamma \rfloor) \to \mathit{heap} \to \lfloor \mathbb{M}\ \gamma \rfloor \\
&\mathit{outl}\ [\![\textbf{inl}\ e]\!]\ k\ h = c\ e\ h \\
&\mathit{outl}\ [\![\textbf{inr}\ e]\!]\ k\ h = [\![\textbf{fail}]\!] \\
&\mathit{outl}\ [\![u]\!] \qquad k\ h = [\![\textbf{do}\ \{\textbf{let inl}\ x = u; \$(k\ x\ h)\}]\!]
\end{aligned}$$

In the same way, we define *outr* and, for use in case 63, *abs*.

The most interesting cases of $\lhd$ are those that constrain the result of a binary operator like $+$. If $e = e_1 + e_2$, we apply the same technique as for negation, with a twist: $\lhd$ can either treat $(e_1\ +)$ as a unary operator and recursively constrain e_2, or it can treat $(+\ e_2)$ as a unary operator and recursively constrain e_1. Either way may fail—for example, a literal real number cannot be constrained (case 57)—so the disintegrator nondeterministically *searches* for a way that works. The search is notated by writing $\sqcup$ between possibilities.

Treating $(e_1\ +)$ as an invertible unary operator gets us only so far: when e_1 includes free variables that are bound in h, the parametricity property of $\lhd$ in (104) does not permit $t - e_1$ to be substituted for s. Problematic free variables are eliminated by replacing them with fresh variables bound *outside* heap h. That is the job of the forward functions $\ggg$ and $\rhd$.

The forward functions Whereas the backward functions $\lll$ and $\lhd$ represent the innovative part of our disintegrator, the forward functions $\ggg$ and $\rhd$ define an online partial evaluator that is more or less standard (Fischer et al. 2008; Fischer, Kiselyov, and Shan 2011). Just as the laws (91) and (92) for the backward functions "pull out" a binding of t, the laws for the forward functions replace a heap-bound variable x with a head normal form v:

$$\textbf{do}\ \{h; \quad x \leftsquigarrow m; M\} \equiv \ggg m\ (\lambda v.\, \overline{M[x \mapsto v]})\ h \tag{97}$$

$$\textbf{do}\ \{h; \textbf{let}\ x = e; M\} \equiv\ \rhd e\ \ (\lambda v.\, \overline{M[x \mapsto v]})\ h \tag{98}$$

Replacing x with v in M enables us to complete the derivation of $\lhd\ (e_1 + e_2)$, case 62, which calls $\rhd\ e_1$ to evaluate e_1 to v_1, then substitutes $t - v_1$ for s. The substitution is safe because v_1, being a head normal form of type $\mathbb{R}$, cannot include any free variable bound in the heap.

The easy cases of $\rhd$ recur structurally through applications of strict functions like $-$ and $+$ (cases 76 to 82). Bottoming out at a real literal or a variable not bound in h is also easy, because these arguments are already in head normal form v (case 75). All these cases are justified by (98).

The interesting case is $\rhd\ x\ k\ [h_1; [\![x \leftsquigarrow m]\!]; h_2]$ (case 83), which reaches a variable x that *is* bound in h. We pass m to $\ggg$, along with the partial heap h_1 and a continuation that composes $\overline{[\textbf{let}\ x = v; h_2]}$ with $k\ v$, where v is the head normal form produced by $\ggg m$. Like case 64, this case replaces the binding $x \leftsquigarrow m$ by **let** $x = v$. It is justified by the induction hypothesis for $\ggg m$ (97) and by alpha- and beta-conversions.

The interesting cases of $\ggg m$ (67 to 69) emit code for a random choice *outside* the heap, binding its outcome to a fresh variable z. The cost of getting this atomic representation of the outcome is that the disintegrator cannot further rewrite m. These cases are justified by using Tonelli's theorem to lift m over h.

Disintegrating a term Disintegration uses both backward and forward functions. It starts with 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. The disintegration of $\tilde{m}$ is the residual program M produced by this combination of $\lhd$ and $\ggg$:

$$M = \ggg \tilde{m}\ (\lambda v.\, \lhd\ (\mathit{fst}\ v)\ t\ (\overline{\textbf{return}\ (\mathit{snd}\ v)})\ [\,]. \tag{99}$$

This code starts with an empty heap $[\,]$ and proceeds in three steps:

1. Perform $\tilde{m}$ and name the resulting pair v.
2. Constrain *fst* v to equal t.
3. Continue with a lifted term that forms a residual program by wrapping the final heap around the final action **return** $(\mathit{snd}\ v)$.

The residual program M represents the posterior distribution, which depends on free variable t.

$\gg$ ⟦**do** {$x \leftsquigarrow$ **uniform** 0 1;
$y \leftsquigarrow$ **uniform** 0 1;
return $(y-2\cdot x,(x,y))$}⟧
$(\lambda v.\lhd\ (fst\ v)\ [\![t]\!]\ \overline{[\![\textbf{return}\ \$(snd\ v)]\!]})$
[]

= { put bindings on heap; perform **return**: (71), (71), (70) }

$\rhd$ ⟦$(y-2\cdot x,(x,y))$⟧ $(\lambda v.\lhd\ (fst\ v)\ [\![t]\!]\ \overline{[\![\textbf{return}\ \$(snd\ v)]\!]})$
$[x \leftsquigarrow$ **uniform** 0 1; $y \leftsquigarrow$ **uniform** 0 1]

= { pass head normal form to continuation (75) }

$\lhd$ ⟦$y-2\cdot x$⟧ ⟦t⟧ $\overline{[\![\textbf{return}\ (x,y)]\!]}$
$[x \leftsquigarrow$ **uniform** 0 1; $y \leftsquigarrow$ **uniform** 0 1]

= { evaluate right summand and constrain left (62);
negate right summand (78) }

$\rhd$ ⟦$2\cdot x$⟧ $(\lambda v.\lhd$ ⟦y⟧ $(\llbracket t\rrbracket + v)$ $\overline{[\![\textbf{return}\ (x,y)]\!]})$
$[x \leftsquigarrow$ **uniform** 0 1; $y \leftsquigarrow$ **uniform** 0 1]

= { evaluate left multiplicand (81);
continue with right multiplicand (75) }

$\rhd$ ⟦x⟧ $(\lambda v.\lhd$ ⟦y⟧ $(\llbracket t\rrbracket + 2\cdot v)$ $\overline{[\![\textbf{return}\ (x,y)]\!]})$
$[x \leftsquigarrow$ **uniform** 0 1; $y \leftsquigarrow$ **uniform** 0 1]

= { perform the action bound to x in the heap (83) }

$\gg$ ⟦**uniform** 0 1⟧
$(\lambda v.\lambda h.\lhd$ ⟦y⟧ $(\llbracket t\rrbracket + 2\cdot v)$ $\overline{[\![\textbf{return}\ (x,y)]\!]}$
$[h;$ **let** $x = v;$ $y \leftsquigarrow$ **uniform** 0 1$])$
[]

= { emit code for the **uniform** action and continue (69) }

⟦**do** {$z \leftsquigarrow$ **uniform** 0 1;
$\$(\lhd$ ⟦y⟧ ⟦$t+2\cdot z$⟧ $\overline{[\![\textbf{return}\ (x,y)]\!]}$
[**let** $x = z;$ $y \leftsquigarrow$ **uniform** 0 1$])$}⟧

= { constrain y: update its heap binding
after constraining outcome of its action (64)}

⟦**do** {$z \leftsquigarrow$ **uniform** 0 1;
$\$(\lll$ ⟦**uniform** 0 1⟧ ⟦$t+2\cdot z$⟧
$\overline{[\![\textbf{do}\ \{\textbf{let}\ y = t+2\cdot z;\ \textbf{return}\ (x,y)\}]\!]}$
[**let** $x = z$]$)$}⟧

= { constrain **uniform** by emitting observation (50);
beta-reduce continuation, leaving residual program }

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

Figure 7. Automatic disintegration in action

An example computation is shown in Figure 7, which disintegrates example (37) starting with the form (99). When $\lhd$ is applied to a sum in case 62, the search chooses the right operand of $\sqcup$. To reach the desired term M in equation (45), the result of Figure 7 is simplified by applying monad laws and by removing **factor** 1.

5.3 Correctness

In (99), M represents a disintegration of $\tilde{m}$ iff

$$\tilde{m} \equiv \textbf{do}\ \{t \leftsquigarrow \textbf{lebesgue};\ p \leftsquigarrow M;\ \textbf{return}\ (t,p)\}. \tag{100}$$

This correctness follows from the specifications of $\lhd$ and $\gg$ in (92) and (97) and from the following associativity laws:

$$\textbf{do}\ \{p \leftsquigarrow \lhd\ e\ v\ c\ h; M\} \equiv \lhd\ e\ v\ (\lambda h'.\ \textbf{do}\ \{p \leftsquigarrow c\ h'; M\})\ h \tag{101}$$

$$\textbf{do}\ \{p \leftsquigarrow \gg\ m\ k\ h; M\} \equiv \gg m\,(\lambda v.\lambda h'.\ \textbf{do}\ \{p \leftsquigarrow k\ v\ h'; M\})\ h \tag{102}$$

To show that the definitions in Figure 6 meet their specifications, we also need associativity laws for $\lll$ and $\rhd$. And we need some *parametricity* properties, which say what happens when the second argument to $\lll$ and $\lhd$ is an expression, not just a variable. Parametricity states that

$$(\lll m\ v\ \overline{M}\ h)[s \mapsto e'] \equiv \lll m\ (v[s \mapsto e'])\ \overline{M[s \mapsto e']}\ h \tag{103}$$

$$(\lhd\ e\ \ v\ \overline{M}\ h)[s \mapsto e'] \equiv\ \lhd\ e\ \ (v[s \mapsto e'])\ \overline{M[s \mapsto e']}\ h, \tag{104}$$

provided variable s is not free in m or e or h, and provided no free variable of e' is bound in h.

Using the parametricity laws, the associativity laws, and the fundamental law of each function (91, 92, 97, and 98), we prove the disintegrator correct by induction on its number of computation steps. The proof also uses monad laws, changes of integration variables according to (46), and exchanges of integrals using the equation

$$\textbf{do}\ \{x \leftsquigarrow m; y \leftsquigarrow n; M\} = \textbf{do}\ \{y \leftsquigarrow n; x \leftsquigarrow m; M\}. \tag{105}$$

Induction hypotheses, monad laws, and changes of variables are correct regardless of inputs. But (105) holds only for some measures m and n (Royden 1988, Chapter 12, problem 25; Pollard 2001, Chapter 4, problem 12). In particular (105) holds (by Tonelli's theorem) whenever m and n denote finite or σ-*finite* measures. Measures that occur naturally in probabilistic programs *are* σ-finite, but because core Hakaru can express measures that are not, in any disintegration the correctness of each exchange of integrals (105) must be verified by some means beyond our disintegrator.

To verify correctness, we first extend the output language with a new syntactic form, which records an exchange of integrals:

$$\text{Terms}\quad e ::= \cdots \mid \textbf{ex}\ m\ h\ e \tag{106}$$

The new form **ex** $m\ h\ e$ means the same as e except it asserts that

$$\textbf{do}\ \{x \leftsquigarrow m; h; M\} = \textbf{do}\ \{h; x \leftsquigarrow m; M\} \tag{107}$$

for all M. We interpret these assertions denotationally by extending the domain of measure $\mathbb{M}\,\alpha$ with an error value. If (107) holds, then **ex** $m\ h\ e$ denotes whatever e denotes; otherwise, **ex** $m\ h\ e$ denotes the error value. In this *instrumented* semantics, the denotations of all other terms are also extended to account for the error value. The only subtle case is the denotation of a monadic-bind term **do** $\{x \leftsquigarrow m; M\}$: it denotes error iff either m denotes error or the set of x for which M denotes error has nonzero measure with respect to m.

We instrument our disintegrator by adding assertions to those cases of $\lll$ and $\gg$ in Figure 6 that handle primitive measures:

$\lll$ ⟦**lebesgue**⟧ $v\ c\ h =$ ⟦**ex lebesgue** $h\ \$(c\ h)$⟧ (49′)

$\lll$ ⟦**uniform** $r_1\ r_2$⟧ $v\ c\ h =$ ⟦**ex** (**uniform** $r_1\ r_2$) h (**do** {…})⟧ (50′)

$\gg$ ⟦u⟧ $k\ h =$ ⟦**ex** u h (**do** {…})⟧ (67′)

$\gg$ ⟦**lebesgue**⟧ $k\ h =$ ⟦**ex lebesgue** h (**do** {…})⟧ (68′)

$\gg$ ⟦**uniform** $r_1\ r_2$⟧ $k\ h =$ ⟦**ex** (**uniform** $r_1\ r_2$) h (**do** {…})⟧ (69′)

Theorem (Correctness). *If, starting with invocation* (99)*, the instrumented disintegrator produces an output M such that the right-hand-side of the specification* (100) *denotes a measure, not the error value, then the two sides of* (100) *denote the same measure.*

The instrumented semantics is defined in such a way that we can verify the correctness of an output simply by proving, for each assertion in the output, that (107) holds, except possibly on a set of measure zero. In our experience, (107) can often be proved by a simple heuristic; for example, if m is **lebesgue** and h contains no **lebesgue** and no unbounded **factor**, then by Tonelli's theorem, (107) follows.

6. Evaluation

Our disintegrator is used with full Hakaru, a suite of program transformations for a language that extends core Hakaru with additional primitive types (such as integers and arrays), operations (such as exp, log, and sin), and distributions (such as normal, gamma, and Poisson). The full disintegrator handles observations not only on real numbers but also on integers, pairs, sums, and arrays.

The full disintegrator works on dozens of generative models and observations, including these (whose code is available at `https://github.com/hakaru-dev/hakaru/blob/master/examples/README-popl2017.md`):

- Uniform distributions over polytopes, such as the unit square, subject to the observations in Section 2 and to the observation $\max(x,y)$, which is used by Chang and Pollard (1997) to motivate disintegration
- Normal distributions over correlated real variables, such as Bayesian linear regression
- Discrete mixtures of continuous distributions, such as Gaussian mixture models for clustering
- Continuous mixtures of discrete distributions, such as Naive Bayes models for document classification
- A simplified model of seismic event detection, involving spherical geometry (Arora, Russell, and Sudderth 2013)
- The collision model used by Afshar, Sanner, and Webers (2016) to motivate observing combinations (such as sums) of random variables
- A linear dynamic model in one dimension, with time steps t_1 and t_2 to be observed, expressed as follows:

$$\begin{array}{l}\textbf{do}\ \{n_p \leftsquigarrow \textbf{uniform}\ 3\ 8; \quad n_t \leftsquigarrow \textbf{uniform}\ 1\ 4; \\ \quad p_1 \leftsquigarrow \textbf{normal}\ 21\ n_p; \quad t_1 \leftsquigarrow \textbf{normal}\ p_1\ n_t; \\ \quad p_2 \leftsquigarrow \textbf{normal}\ p_1\ n_p; \quad t_2 \leftsquigarrow \textbf{normal}\ p_2\ n_t; \\ \quad \textbf{return}\ ((t_1,t_2),(n_p,n_t))\}\end{array} \tag{108}$$

The disintegrator succeeds on these inputs because they share a form typical of probabilistic programs: the observable expression denotes $f(x,y)$, where x and y are random choices, the distribution of x has a density, and the function f is invertible in the argument x.

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 and compact. These results are encouraging for practice because there exist inputs that make our disintegrator consume exponential time and space (cases 54 and 73 double the work for every **mplus**).

The disintegrator produces *exact, symbolic* posteriors. Like all terms of full Hakaru, these posteriors can be simplified using computer algebra, interpreted as importance samplers, or transformed into terms that implement other sampling methods (Narayanan et al. 2016). The disintegrator thus enables Hakaru to achieve unprecedented modularity. 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 sampling, and Gibbs sampling (MacKay 1998). By invoking the disintegrator, our implementations of those methods stay independent of any model, and they work with every model listed above.

The disintegrator enables Hakaru to achieve this modularity without imposing significant run-time costs. Although those Hakaru samplers just mentioned are not yet competitive with state-of-the-art, application-specific samplers, which are hand-written in general-purpose programming languages and then compiled, they are competitive with samplers written in other probabilistic programming languages such as WebPPL (Goodman and Stuhlmüller 2014). Quantitative details are beyond the scope of this paper.

Limitations Like many proof-of-concept partial evaluators, the disintegrator can duplicate code (such as $t + 2 \cdot z$ in Figure 7) and produce large programs. The limitation obstructs large-scale use, but it should yield to standard common-subexpression elimination.

The disintegrator is limited by its local, syntactic reasoning. For example, it can disintegrate an observation like $2 \cdot x$, but it can't disintegrate the semantically equivalent observation $x + x$. Perhaps surprisingly, this limitation has not caused trouble in practice. It may be that the expressions observed in today's probabilistic programs are relatively simple. If the limitation proves problematic in the future, it could be addressed by more advanced computer algebra.

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 hint what they expect to be constrainable, and if we knew how to do "separate disintegration." That is all future work.

When the output of the disintegrator is used as a sampler, it is not necessarily efficient. For example, the sampler derived from equation (11) rejects half the samples taken from **uniform** 0 1; it would be more efficient if we changed **uniform** 0 1 to **uniform** 0 $(1/2)$ by symbolic simplification (Carette and Shan 2016; Gehr, Misailovic, and Vechev 2016). 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. Such tricks will carry us only so far; efficient sampling is a longstanding problem that measure theory merely helps to address.

7. Related Work

Algorithms for Bayesian inference have become widespread. But the algorithms found in a typical probabilistic language (Goodman et al. 2008; Wingate, Stuhlmüller, and Goodman 2011) require the observation to be specified as a random choice in the model, and they produce approximate, numerical answers. Our disintegrator is the first algorithm to let the observation be specified as an expression separate from the generative model, so we can infer a posterior distribution from a zero-probability observation without tripping over Borel's paradox. And our disintegrator produces an exact answer expressed as a term in the modeling language, so it can be combined with other inference algorithms, both exact and approximate, without losing performance.

Inference from zero-probability observations is a longstanding concern. 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 construct conditional distributions *mathematically* (Tjur 1975; Ackerman, Freer, and Roy 2011, 2016; Borgström et al. 2013). 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).

A disintegrator can succeed only when a disintegration exists and can be expressed. Ackerman, Freer, and Roy (2011) show that when

the modeling language expresses all and only *computable* distributions, not all disintegrations can be expressed. When the language is less expressive (like core Hakaru) or differently expressive, we don't know what disintegrations can be expressed.

To specify the semantics of a probabilistic language, it is popular to let terms denote *samplers*, or computations that produce a random outcome 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 (Kozen 1981; Staton et al. 2016). This semantics lets us reason 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 backward mode ($\lll$ and $\lhd$) resembles weakest-precondition reasoning (Dijkstra 1975; Nori et al. 2014), pre-image computation (Toronto, McCarthy, and Horn 2015), and constraint propagation (Saraswat, Rinard, and Panangaden 1991; Gupta, Jagadeesan, and Panangaden 1999). The forward mode ($\ggg$ and $\rhd$) 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).

It is well known that continuations can be used to manage 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. Also, thanks to the view of measures as integrators, our semantics uses continuation passing to compose measures easily (Audebaud and Paulin-Mohring 2009).

8. Discussion

Our new disintegration algorithm infers a posterior distribution from a zero-probability observation of a continuous quantity. This long-standing problem 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. On countable spaces, life is easy: every measure is absolutely continuous with respect to the counting measure, so disintegration gives exactly the same answers as the classic conditioning equation (5).

When disintegrating with respect to the counting measure, the functions $\lll$ and $\lhd$ in Figure 6 needn't fail in cases 48, 56, and 57; 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 Boolean 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 Boolean 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 that the Boolean $y = 2 \cdot x$ is **true**—Hakaru responds that the posterior measure is zero and thus has no expectation.

Disintegrating with respect to the Lebesgue measure or the counting measure has the advantage that density calculation (Bhat et al. 2012, 2013; Mohammed Ismail and Shan 2016) falls out as a special case. But practical applications demand more expressive measures, such as the sum of the Lebesgue measure with a counting measure—that is, where probability is defined by combining a density function with a set of point masses. Examples include

- A continuous distribution, like a normal distribution, that is "clamped" by treating negative values as 0
- A measure on the union of two particular hyperplanes in $\mathbb{R}^4$, namely $\{ (x_1, x_2, x_3, x_4) \mid x_1 = x_3 \vee x_2 = x_4 \}$, which is used to define *single-site* Metropolis-Hastings sampling (Tierney 1998)

These examples work in a hacked version of our disintegrator, but the hack compromises soundness. Future work must generalize the disintegrator beyond the Lebesgue and counting measures in a sound and principled way.

In general, the posterior measure produced by our disintegrator is not a probability distribution—the constant 1 function may integrate to any number. If a probability distribution is desired, the posterior measure can easily be normalized after the fact. (Disintegration and normalization cannot be accomplished in a single pass because normalized disintegration is not compositional. For example, the normalized disintegrations of m_1 and of m_2 are not enough to determine the normalized disintegration of **mplus** m_1 m_2.)

The definition of disintegration allows latitude that our disintegrator does not take: When we disintegrate $\xi = \Lambda \otimes \kappa$, the output κ is unique only *almost everywhere*—$\kappa\, x$ may return an arbitrary measure at, for example, any finite set of x's. But our disintegrator never invents an arbitrary measure at any point. The mathematical definition of disintegration is therefore a bit too loose to describe what our disintegrator actually does. How to describe our disintegrator by a tighter class of "well-behaved disintegrations" is a question for future research. In particular, the notion of *continuous* disintegrations (Ackerman, Freer, and Roy 2016) is too tight, because depending on the input term, our disintegrator does not always return a continuous disintegration, even if one exists.

This paper begins 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 observable *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, Prakash Panangaden, Daniel Roy, Dylan Thurston, Mitchell Wand, and Robert Zinkov for helpful comments and discussions. Joe Stoy helped us understand notation.

This research was supported by DARPA contracts FA8750-14-2-0007 and FA8750-14-C-0002, 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.

References

Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. 2011. Noncomputable conditional distributions. In *LICS 2011: Proceedings of the 26th Symposium on Logic in Computer Science*, pages 107–116, Washington, DC. IEEE Computer Society Press.

Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. 2016. On computability and disintegration. *Mathematical Structures in Computer Science*, pages 1–28.

Hadi Mohasel Afshar, Scott Sanner, and Christfried Webers. 2016. Closed-form Gibbs sampling for graphical models with algebraic constraints. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*. AAAI Press.

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.

Jacques Carette and Chung-chieh Shan. 2016. Simplifying probabilistic programs using computer algebra. In *Practical Aspects of Declarative Languages: 18th International Symposium, PADL 2016*, Lecture Notes in Computer Science, pages 135–152, Berlin. Springer.

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.

Jean Dieudonné. 1947–1948. Sur le théorème de Lebesgue-Nikodym (III). *Annales de l'université de Grenoble*, 23:25–53.

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.

Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact symbolic inference for probabilistic programs. In *Proceedings of the 28th International Conference on Computer Aided Verification, Part I*, number 9779 in Lecture Notes in Computer Science, pages 62–83, 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.

Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A language for generative models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, pages 220–229, Corvallis, Oregon. AUAI Press.

Noah D. Goodman and Andreas Stuhlmüller. 2014. The design and implementation of probabilistic programming languages. `http://dippl.org`. Accessed: 2016-11-04.

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. English translation *Foundations of the Theory of Probability*, Chelsea, New York, 1950.

Dexter Kozen. 1981. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350.

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.

Wazim Mohammed Ismail and Chung-chieh Shan. 2016. Deriving a probability density calculator (functional pearl). In *ICFP'16: Proceedings of the 2016 ACM SIGPLAN International Conference on Functional Programming*, pages 47–59, New York. ACM Press.

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *Functional and Logic Programming: 13th International Symposium, FLOPS 2016*, number 9613 in Lecture Notes in Computer Science, pages 62–79, Berlin. Springer.

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.

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.

Sam Staton, Hongseok Yang, Chris Heunen, Ohad Kammar, and Frank Wood. 2016. Semantics for probabilistic programming: Higher-order functions, continuous distributions, and soft constraints. In *LICS 2016: Proceedings of the 31st Symposium on Logic in Computer Science*, Washington, DC. IEEE Computer Society 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.

David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. 2011. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of AISTATS 2011: 14th International Conference on Artificial Intelligence and Statistics*, number 15 in JMLR Workshop and Conference Proceedings, pages 770–778, Cambridge. MIT Press.