Symbolic disintegration with a variety of base measures

PRAVEEN NARAYANAN, Indiana University, USA
CHUNG-CHIEH SHAN, Indiana University, USA

Disintegration is a relation on measures and a transformation on programs that generalizes density and conditioning, two operations widely used for exact and approximate probabilistic inference. Existing program transformations that find a disintegration or density automatically are limited to a fixed base measure that is an independent product of Lebesgue and counting measures, so they are of no help in practical cases that require tricky reasoning about other base measures. We present the first disintegrator that handles variable base measures, including *discrete-continuous mixtures*, *dependent products*, and *disjoint sums*. By analogy to type inference, our disintegrator can check a given base measure as well as infer an unknown one that is principal. We derive the disintegrator and prove it sound by equational reasoning from semantic specifications. It succeeds in a variety of applications where disintegration and density had not been previously mechanized.

Additional Key Words and Phrases: probabilistic programs, density functions, conditional measures

1 INTRODUCTION

Probability distributions are used nowadays to formulate and solve all sorts of problems in artificial intelligence and beyond. Typically, in order to turn a distribution that models a problem into a program that implements a solution, domain experts subject the distribution to measure-theoretic operations such as *density*, *conditioning*, *marginalization*, and *expectation*. Probabilistic programming represents distribution as programs and mechanizes these operations, whether exact or approximate, through symbolic and numerical computations.

Density and conditioning are two important operations on distributions, used to define both inference problems and approximate solutions. For example, density is used to define *maximum likelihood estimates* and *Metropolis-Hastings samplers*, whereas conditioning is used to define *posterior belief updates* and *Gibbs samplers*. In fact, density and conditioning are special cases of *disintegration*. Recent years have seen the exact, symbolic mechanization of both density [Bhat et al. 2012, 2013; Mohammed Ismail and Shan 2016] and disintegration [Shan and Ramsey 2017; Narayanan and Shan 2017]. Unlike other implementations of inference and sampling [Goodman et al. 2008; Wingate et al. 2011; Wood et al. 2014], these mechanizations allow the inference user to specify the observation as an expression separate from random choices in the model, and they allow the sampling user to specify the proposal distribution as a probabilistic program.

Just as monadic bind is a sort of measure 'multiplication', the operations of density, conditioning, and disintegration are a sort of measure 'division'. When performing or reasoning about these operations, it is common to assume that the *base measure* ('denominator') is equal to—or at least absolutely continuous with respect to ('divisible by')—the *stock measure*, an independent product of Lebesgue and counting measures. But as illustrated in Section 2, this assumption is violated in many inference problems and approximate solutions. Those cases are trickier and so call for mechanization, yet existing mechanizations make the same assumption and so produce no output.

This paper presents the first disintegration program transformation to allow the base measure to vary from the stock measure. More precisely, our disintegrator is the first to let the base measure be

- mixtures of the Lebesgue measure and point masses,
- dependent products, and
- disjoint sums.

2019. 2475-1421/2019/1-ART1 $15.00
https://doi.org/

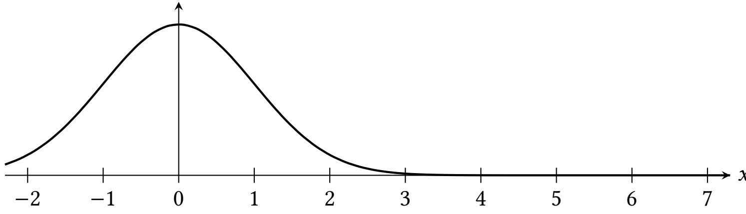

Fig. 1. A density of the normal distribution **normal** 0 1 with respect to the Lebesgue measure

Using this variety, we automate established applications of disintegration, namely

- calculations on models that mix continuous and discrete distributions, and
- Metropolis-Hastings sampling using single-site and reversible-jump proposals.

We prove our disintegrator sound by equational reasoning. Our proof constitutes the core of verifying those applications where non-stock base measures are involved. In particular, our proof encapsulates reasoning about densities among, and reparameterizations of, mixture distributions.

In Section 2, we specify disintegration as a measure-preserving program transformation and motivate variable-base disintegration using instances of density and conditioning. In Sections 3 to 7, we then present our variable-base disintegrator by progressively defining four measure-preserving program transformations. Our development is inspired by an analogy to constraint-based type inference: just as a type checker can be made to infer unknown types by collecting and solving constraints on type variables to find a principal type, it turns out in Section 6 that a base-measure checker can be made to infer unknown base measures by collecting and solving constraints on base-measure variables to find a *principal base measure*. Finally, in Section 8 we confirm empirically that our new disintegrator handles our motivating applications.

2 BACKGROUND

We introduce disintegration as a measure-theoretic relation and probabilistic-program transformation, by generalizing from density and conditioning. Along the way, we tweak each example to motivate varying the base measure from the stock measure such as the Lebesgue measure. Our examples are expressed in *core Hakaru*, a small probabilistic language [Shan and Ramsey 2017], whose formal review we postpone to Section 3.

2.1 Defining density

Density is what relates measures in the popular imagination to the actual definition of measures. In the popular imagination, a measure over a type α is a function that maps an α-value to its weight (a number). For example, the normal distribution in the popular imagination is the bell curve (Figure 1), a function that maps each real to its weight. But actually, a measure over α, or a *generalized* function, is defined as a function that either maps an α-set to its size (a number), or equivalently, maps an α-to-number function to its integral (a number). Thus, density is what relates $\alpha \rightarrow \mathbb{R}_+$ to $\mathbb{M}\,\alpha$, where the type $\mathbb{M}\,\alpha$ of measures over α is either $(\alpha \rightarrow \{0, 1\}) \rightarrow \mathbb{R}_+$ or equivalently $(\alpha \rightarrow \mathbb{R}_+) \rightarrow \mathbb{R}_+$. The latter definition of measures—as integrators—is more convenient for our purposes, so we stick to it in this paper. (More precisely, α and $\mathbb{M}\,\alpha$ are *measurable spaces*, an element of $\mathbb{R}_+$ is either a non-negative real number or ∞, and a measure over α is a certain kind of function that maps each *measurable* α-set or *measurable* α-to-$\mathbb{R}_+$ function to an element of $\mathbb{R}_+$.)

We define density and illustrate its applications using small examples and varying base measures.

Example 2.1. The normal distribution is written **normal** 0 1 in core Hakaru, because its mean is 0 and its standard deviation is 1. Whereas the bell curve in Figure 1 is the function

$$\textbf{dnorm}\ 0\ 1 \quad = \quad \lambda x.\, \exp(-x^2/2)/\sqrt{2\pi} \quad : \quad \mathbb{R} \to \mathbb{R}_+, \tag{1}$$

the normal distribution is the measure

$$\textbf{normal}\ 0\ 1 \quad = \quad \lambda f.\, \int_{\mathbb{R}} (\textbf{dnorm}\ 0\ 1)(x) \cdot f(x)\, dx \quad : \quad \mathbb{M}\,\mathbb{R}. \tag{2}$$

This integrator maps any (measurable) function $f : \mathbb{R} \to \mathbb{R}_+$ to the expected value of $f(x)$ when x is drawn randomly from the normal distribution. For example, if f maps positive reals to 1 and other reals to 0, then (**normal** 0 1)(f) = 1/2, because the probability is 1/2 that a real drawn randomly from the normal distribution is positive.

Another measure is the Lebesgue measure, written **lebesgue** in core Hakaru:

$$\textbf{lebesgue} \quad = \quad \lambda f.\, \int_{\mathbb{R}} f(x)\, dx \quad : \quad \mathbb{M}\,\mathbb{R}. \tag{3}$$

This integrator maps any (measurable) function $f : \mathbb{R} \to \mathbb{R}_+$ to its integral in the introductory-calculus sense. For example, if f maps positive reals x to (**dnorm** 0 1)(x) and other reals to 0, then **lebesgue**(f) = 1/2, because the integral of **dnorm** 0 1 over the positive reals is 1/2.

Definition 2.2. The *total* $|\mu|$ of a measure μ is its integral of the constant-1 function. That is, $|\mu| = \mu(\lambda_.\, 1)$. For example, $|\textbf{normal}\ 0\ 1| = \int_{\mathbb{R}} (\textbf{dnorm}\ 0\ 1)(x)\, dx = 1$. Such a measure, whose total is 1, is called a *probability distribution* and *normalized*. In contrast, $|\textbf{lebesgue}| = \int_{\mathbb{R}} 1\, dx = \infty$, so **lebesgue** is *unnormalized*.

The equation below relates the bell curve **dnorm** 0 1 to the measures **normal** 0 1 and **lebesgue**:

$$\textbf{normal}\ 0\ 1 \quad = \quad \lambda f.\, \textbf{lebesgue}\big(\lambda x.\, (\textbf{dnorm}\ 0\ 1)(x) \cdot f(x)\big). \tag{4}$$

Checking it is a matter of β-reduction. In core Hakaru, the right-hand-side is expressed as follows:

$$\textbf{normal}\ 0\ 1 \quad = \quad \textbf{do}\ \{x \leftsquigarrow \textbf{lebesgue};\ (\textbf{dnorm}\ 0\ 1)(x) \odot \textbf{return}\ x\}. \tag{5}$$

The core Hakaru forms **do** $\{x \leftsquigarrow \cdots; \cdots\}$ and **return** denote bind and unit in the measure monad [Giry 1982; Ramsey and Pfeffer 2002]. Another name for **return** x is the *Dirac distribution* at x; it denotes the integrator $\lambda f.\, f(x)$. The $\odot$ form scales the measure **return**x by the weight (**dnorm**01)(x).

What we have just seen is that **dnorm** 0 1 is a density of **normal** 0 1 with respect to the base measure **lebesgue**.

Definition 2.3. Let $\xi, \mu : \mathbb{M}\,\alpha$ be two measures and $\kappa : \alpha \to \mathbb{R}_+$ be a (measurable) function. We say that κ is a *density* (or *Radon-Nikodym derivative*) of ξ with respect to the *base measure* μ if

$$\xi \quad = \quad \lambda f.\, \mu\big(\lambda x.\, \kappa(x) \cdot f(x)\big), \tag{6}$$

or in core Hakaru,

$$\xi \quad = \quad \textbf{do}\ \{x \leftsquigarrow \mu;\ \kappa(x) \odot \textbf{return}\ x\}. \tag{7}$$

We write $\xi = \kappa \mathrel{=\!\odot} \mu$ for short.

A measure over $\mathbb{R}$ is called *continuous* if it has a density with respect to the Lebesgue measure. Bhat et al.'s density calculation procedure [2012, 2013] turns probabilistic programs that denote continuous measures (such as **normal** 0 1) into their exact densities (such as **dnorm** 0 1) symbolically.

Many measures over $\mathbb{R}$ are continuous, but many are not.

Example 2.4. The measure **return** 42 : $\mathbb{M}\,\mathbb{R}$ is not continuous: If it were, then equation (6) would yield $\lambda f.\, f(42) = \lambda f.\, \int_{\mathbb{R}} \kappa(x) \cdot f(x)\, dx$. When we take the integrators on both sides and apply them to the function f that maps 42 to 37 and all other reals to 0, we get $37 = 0$, a contradiction.

Example 2.5. Non-continuous measures arise from *clamping* continuous measures. Clamping means replacing out-of-bounds outcomes by the bound. For example, a sensor that can measure only reals between 0 and 1 (like a camera pixel) might sense reals less than 0 as 0 and reals greater than 1 as 1. Clamping **normal** 0 1 in this way yields a measure whose outcome is exactly 0 half of the time and exactly 1 almost 16% of the time. More precisely, as an integrator it is

$$\begin{aligned}\xi &= \lambda f.\, \int_{\mathbb{R}} (\mathbf{dnorm}\ 0\ 1)(x) \cdot f(\max\{0, \min\{1, x\}\})\, dx \\ &= \lambda f.\, \int_{-\infty}^{0} (\mathbf{dnorm}\ 0\ 1)(x)\, dx \cdot f(0) + \int_{0}^{1} (\mathbf{dnorm}\ 0\ 1)(x) \cdot f(x)\, dx + \int_{1}^{+\infty} (\mathbf{dnorm}\ 0\ 1)(x)\, dx \cdot f(1),\end{aligned} \tag{8}$$

and core Hakaru can express it as

$$\begin{aligned}\xi \;&=\; \mathbf{do}\ \{x \leftsquigarrow \mathbf{normal}\ 0\ 1;\ \mathbf{return}\ \max\{0, \min\{1, x\}\}\} \\ &=\; \int_{-\infty}^{0} (\mathbf{dnorm}\ 0\ 1)(x)\, dx \odot \mathbf{return}\ 0 \\ &\quad \oplus\ \mathbf{do}\ \{x \leftsquigarrow \mathbf{normal}\ 0\ 1;\ \mathbf{if}\ 0 \le x \le 1\ \mathbf{then\ return}\ x\ \mathbf{else\ fail}\} \\ &\quad \oplus \int_{1}^{+\infty} (\mathbf{dnorm}\ 0\ 1)(x)\, dx \odot \mathbf{return}\ 1.\end{aligned} \tag{9}$$

In equation (9), the binary operator $\oplus$ denotes summing (*mixing*) measures of the same type, and takes precedence lower than $\odot$. Its identity **fail** denotes the zero measure. The first integral is $1/2$ and the second integral is almost 16%. The same measure is denoted whether $\le$ or $<$ is used.

Arguments similar to that in Example 2.4 show that this measure ξ has no density with respect to **lebesgue**, or to **return** 0 or **return** 1. However, with respect to the sum measure

$$\mu \;=\; \mathbf{lebesgue} \oplus \mathbf{return}\ 0 \oplus \mathbf{return}\ 1 \;=\; \lambda f.\, f(0) + \int_{\mathbb{R}} f(x)\, dx + f(1), \tag{10}$$

it does have the density $\kappa : \mathbb{R} \to \mathbb{R}_+$ defined by

$$\kappa \;=\; \lambda x. \begin{cases} \int_{-\infty}^{0} (\mathbf{dnorm}\ 0\ 1)(x)\, dx & \text{if } x = 0 \\ (\mathbf{dnorm}\ 0\ 1)(x) & \text{if } 0 < x < 1 \\ \int_{1}^{+\infty} (\mathbf{dnorm}\ 0\ 1)(x)\, dx & \text{if } x = 1 \\ 0 & \text{otherwise.} \end{cases} \tag{11}$$

The state of the art in mechanizing density on probabilistic programs is limited to the **lebesgue** base measure. Thus, our disintegrator is the first transformation that can turn ξ in (9) into κ in (11) with respect to μ in (10).

2.2 Using density for inference

A basic application of density is to adjudicate between two hypotheses competing to explain an observation.

Example 2.6. Suppose we observe a real x drawn randomly from a black box, and we wonder whether the black box is **normal** 0 1 or **normal** 3 2. We would like to compare the likelihood of drawing the observed real from each of the two normal distributions. Unfortunately, the probability of drawing any real from any normal distribution is zero, and comparing zero against zero does not help us adjudicate between the two hypotheses.

Instead, we can compare the densities of the two normal distributions with respect to the Lebesgue measure, at the observed real. Figure 2 plots those densities. As the plot shows, if we observe the real 1, then we should favor the hypothesis **normal** 0 1 because $(\mathbf{dnorm}\ 0\ 1)(1) > (\mathbf{dnorm}\ 3\ 2)(1)$, whereas if we observe the real 2, then we should favor the hypothesis **normal** 3 2, because $(\mathbf{dnorm}\ 0\ 1)(2) < (\mathbf{dnorm}\ 3\ 2)(2)$. These densities can be found symbolically using Bhat et al.'s density calculation procedure [2012, 2013].

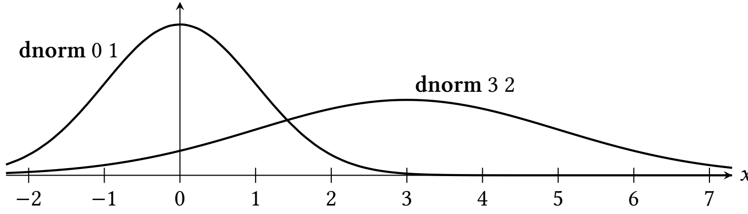

Fig. 2. Densities of two normal distributions, **normal** 0 1 and **normal** 3 2, with respect to the Lebesgue measure

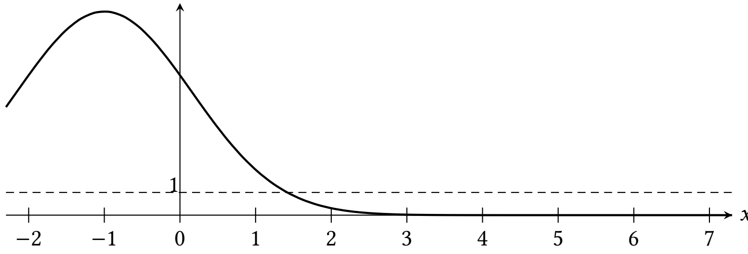

Fig. 3. A density of **normal** 0 1 with respect to **normal** 3 2, the ratio of the two densities in Figure 2

The choice of the Lebesgue measure as the common base measure in this comparison is common because continuous measures are common, but arbitrary. The comparison only depends on the ratio of the two densities, which is the same regardless of the common base measure, as long as both densities exist. For example, if we double the Lebesgue measure ($2 \odot$ **lebesgue**) as the base measure, then the two densities would each be halved, but their ratio would remain the same. In fact, we can just pick **normal** 3 2 as the base measure, and take advantage of the fact that the constant-1 function is a density of every measure with respect to itself. The ratio between **dnorm** 0 1 and **dnorm** 3 2 is itself a density—of **normal** 0 1 with respect to **normal** 3 2. It is plotted in Figure 3 and compared against the constant-1 function: it is greater than 1 at $x = 1$ and less than 1 at $x = 2$.

The ratio being itself a density follows from two general facts.

Proposition 2.7. (1) *If κ is a density of ξ with respect to μ, and λ is a density of μ with respect to ν, then the pointwise multiplication $\lambda x. (\kappa(x) \cdot \lambda(x))$ is a density of ξ with respect to ν. (Thus, the existence of a density is a preorder among measures.)*

(2) *If κ is a density of ξ with respect to μ, and κ is μ-almost everywhere finite and nonzero, then the pointwise reciprocal $\lambda x. 1/\kappa(x)$ is a density of μ with respect to ξ.*

Proof. By monad laws and the equivalence between $l \odot (k \odot m)$ and $(l \cdot k) \odot m$. □

Example 2.8. The reasoning in Example 2.6 can just as well be used to compare non-continuous distributions. For example, to adjudicate whether a certain black box is the clamping of **normal** 0 1 or of **normal** 3 2 to the interval [0, 1], we can find a density of one clamped distribution with respect to the other, namely the ratio of the densities of the two clamped distributions with respect to the common base measure μ in (10). Our disintegrator is the first transformation to automate this reasoning, because these clamped distributions are not continuous.

Example 2.9. Comparing hypotheses is not the only calculation on models that density is used to express. Another application is *mutual information*, a widely used quantity defined as the expected log of a certain density [Cover and Thomas 2006]. The need to estimate mutual information for

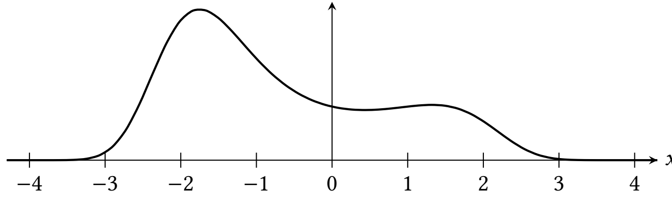

Fig. 4. The target density in Examples 2.10 and 2.14 with respect to the Lebesgue measure

mixtures of continuous and discrete distributions motivated Gao et al. [2017] to estimate mutual information by approximating the density. Our disintegrator can find an exact expression for the same density and plug into Gao et al.'s estimator.

2.3 Using density for sampling

Drawing samples from a distribution is a common way to examine it, such as to estimate its mean and variance or to plot its histogram. However, even when the distribution we care about is easy to define, it is often not obvious how to sample from it. To take a concrete example, it is easy to precisely define a distribution $\xi : \mathbb{M}\,\mathbb{R}$ by [MacKay 1998]

$$\xi \quad = \quad \big(\lambda x.\, \exp(0.4(x-0.4)^2 - 0.08x^4)\big) \mathrel{=\!\!\odot} \textbf{lebesgue}, \tag{12}$$

but it is not obvious how to sample from it.

In these situations, it can help to find a density κ of the *target* distribution ξ with respect to some similar *proposal* distribution μ whose sampling method *is* known. Instead of sampling from ξ, we can draw samples from μ and *weight* each sample x by $\kappa(x)$. This technique is called *importance sampling* or *likelihood weighting*.

In order for the samples x drawn from μ to approximate ξ correctly, we must use their weights $\kappa(x)$ to compensate for the difference between μ and ξ. For example, instead of estimating the mean of ξ by averaging samples from ξ (which may be difficult to draw), we can average samples from μ *weighted* by κ. And instead of plotting a histogram of samples from ξ, we can plot a histogram of samples from μ, but instead of counting the samples in each bin, we should total their weights.

Example 2.10. Suppose we have a function $f : \mathbb{R} \to \mathbb{R}_+$ and we want to estimate its expectation with respect to ξ in (12). For this target ξ, we can use **normal** 3 2 as the proposal, because it is well known how to sample from a normal distribution. To find the density of ξ with respect to **normal** 3 2, we can divide their densities with respect to **lebesgue**, using Proposition 2.7. The form of (12) manifests a density of ξ with respect to **lebesgue**, plotted in Figure 4. And a density of **normal** 3 2 is already plotted in Figure 2. So to estimate the expectation of f with respect to ξ, we can sample x from **normal** 3 2 and average $f(x)$ weighted by $\exp(0.4(x-0.4)^2 - 0.08x^4)/(\textbf{dnorm}\,3\,2)(x)$.

Example 2.11. For densities with many factors revealed gradually (as when monitoring a time series), importance sampling generalizes to *particle filtering* [Gordon et al. 1993]. These techniques are just as useful for distributions that are non-continuous (for example, clamped), but the common base measure used to find a density of the target with respect to the proposal can no longer be the Lebesgue measure. As above, our disintegrator is the first transformation to find such densities. Wu et al. [2018] developed two algorithms, *lexicographic* likelihood weighting and *lexicographic* particle filtering, that handle these distributions, but they carry out particular inference techniques rather than finding densities in general, and they do not allow specifying a custom proposal distribution.

More substantial applications of density—indeed, of probabilistic reasoning in general—require measures over products.

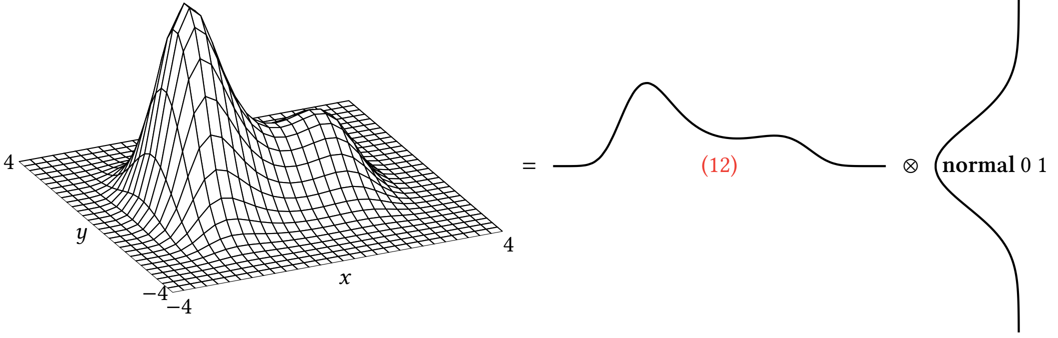

Fig. 5. The product measure of Figures 4 and 1

Definition 2.12. A measure over a product $\alpha \times \beta$ is called a *joint measure*. Given two measures $\mu : \mathbb{M}\,\alpha$ and $\nu : \mathbb{M}\,\beta$, we can construct their *product measure* $\mu \otimes \nu : \mathbb{M}\,(\alpha \times \beta)$ as follows:

$$\mu \otimes \nu \;=\; \lambda f.\, \mu\big(\lambda x.\, \nu\big(\lambda y.\, f(x,y)\big)\big) \;=\; \mathbf{do}\ \{x \leftsquigarrow \mu;\, y \leftsquigarrow \nu;\, \mathbf{return}\ (x,y)\} \tag{13}$$

$$\;=\; \lambda f.\, \nu\big(\lambda y.\, \mu\big(\lambda x.\, f(x,y)\big)\big) \;=\; \mathbf{do}\ \{y \leftsquigarrow \nu;\, x \leftsquigarrow \mu;\, \mathbf{return}\ (x,y)\}. \tag{14}$$

Intuitively, to draw from $\mu \otimes \nu$ is to draw from μ and from ν *independently* then return the results as a pair. For example, Figure 5 depicts the product measure of two distributions over $\mathbb{R}$. In order to ensure *commutativity*, which means that the measures in equations (13) and (14) are defined and equal, Staton [2017] established the invariant that all measure expressions denote *s-finite* measures.

More generally, given a measure $\mu : \mathbb{M}\,\alpha$ and a function $\nu : \alpha \to \mathbb{M}\,\beta$, we can construct their *dependent product measure* $\mu \otimes\!\!= \nu : \mathbb{M}\,(\alpha \times \beta)$ as follows:

$$\mu \otimes\!\!= \nu \;=\; \lambda f.\, \mu\big(\lambda x.\, \nu(x)\big(\lambda y.\, f(x,y)\big)\big) \;=\; \mathbf{do}\ \{x \leftsquigarrow \mu;\, y \leftsquigarrow \nu(x);\, \mathbf{return}\ (x,y)\}. \tag{15}$$

Intuitively, $\mu \otimes\!\!= \nu$ is like $\mu \otimes \nu$, except ν depends on the outcome of μ. Symmetrically, given a function $\mu : \beta \to \mathbb{M}\,\alpha$ and a measure $\nu : \mathbb{M}\,\beta$, we can construct the dependent product $\mu =\!\!\otimes \nu : \mathbb{M}\,(\alpha \times \beta)$ where μ depends on ν instead:

$$\mu =\!\!\otimes \nu \;=\; \lambda f.\, \nu\big(\lambda y.\, \mu(y)\big(\lambda x.\, f(x,y)\big)\big) \;=\; \mathbf{do}\ \{y \leftsquigarrow \nu;\, x \leftsquigarrow \mu(y);\, \mathbf{return}\ (x,y)\}. \tag{16}$$

These dependent products are useful for Metropolis-Hastings sampling, as we illustrate in Example 2.14 below. To work out that example, it is useful to know how to determine the density of a product measure from densities of its factors.

Proposition 2.13. *If $\kappa : \alpha \to \mathbb{R}_+$ is a density of $\xi : \mathbb{M}\,\alpha$ with respect to $\mu : \mathbb{M}\,\alpha$, and $\lambda : \beta \to \mathbb{R}_+$ is a density of $\zeta : \mathbb{M}\,\beta$ with respect to $\nu : \mathbb{M}\,\beta$, then the product measure $\xi \otimes \zeta : \mathbb{M}\,(\alpha \times \beta)$ has the density $\lambda(x,y).\,(\kappa(x) \cdot \lambda(y)) : (\alpha \times \beta) \to \mathbb{R}_+$ with respect to $\mu \otimes \nu : \mathbb{M}\,(\alpha \times \beta)$.*

More generally, the dependent product measure $\xi \otimes\!\!= \zeta$ has the density $\lambda(x,y).\,(\kappa(x) \cdot \lambda(x)(y))$ with respect to $\mu \otimes\!\!= \nu$, provided that κ is a density of ξ with respect to μ and $\lambda(x)$ is a density of $\zeta(x)$ with respect to $\nu(x)$ for ξ-almost all x. And symmetrically for $\xi =\!\!\otimes \zeta$ with respect to $\mu =\!\!\otimes \nu$.

Proof. By monad laws, commutativity, and the equivalence between $l \odot (k \odot m)$ and $(l \cdot k) \odot m$. □

Metropolis-Hastings sampling [Metropolis et al. 1953; Hastings 1970] is a popular inference technique that depends on a *target distribution* $\xi : \mathbb{M}\,\alpha$ and a *proposal kernel* $\zeta : \alpha \to \mathbb{M}\,\alpha$. The proposal kernel ζ specifies a search strategy by which to explore the target distribution ξ. The user of the technique specifies ξ and ζ then calculates the *acceptance ratio*, a density of $\zeta =\!\!\otimes \xi : \mathbb{M}\,\alpha^2$ with respect to $\xi \otimes\!\!= \zeta : \mathbb{M}\,\alpha^2$ [Tierney 1998]. This density is then used in the probabilistic body of

a loop. The density is called a ratio because it is usually calculated by dividing densities of $\zeta \mathrel{=\!\!\otimes} \xi$ and $\xi \mathrel{\otimes\!\!=} \zeta$ with respect to some common base measure, using Proposition 2.7.

Example 2.14. The target distribution $\xi : \mathbb{M}\,\mathbb{R}$ in equation (12) above gives a small instance of Metropolis-Hastings sampling whose acceptance ratio can be calculated using Bhat et al.'s procedure [2012, 2013]. Let us choose the proposal kernel $\zeta = \lambda x.\, \textbf{normal}\ x\ 1 : \mathbb{R} \to \mathbb{M}\,\mathbb{R}$, so $\zeta(x)$ has a density with respect to **lebesgue** for all x. By Proposition 2.13, with respect to the base measure **lebesgue** $\otimes$ **lebesgue**, the measure $\xi \mathrel{\otimes\!\!=} \zeta$ has the density $\kappa : \mathbb{R}^2 \to \mathbb{R}_+$ defined by

$$\kappa \quad = \quad \lambda(x,y).\, \exp(0.4(x-0.4)^2 - 0.08x^4) \cdot \exp(-(y-x)^2/2)/\sqrt{2\pi}, \tag{17}$$

and the measure $\zeta \mathrel{=\!\!\otimes} \xi$ has the density $\kappa \circ \mathit{swap}$ where $\mathit{swap}(y,x) = (x,y)$. Thus by Proposition 2.7, the acceptance ratio is

$$\lambda(x,y).\, \frac{\exp(0.4(y-0.4)^2 - 0.08y^4) \cdot \exp(-(x-y)^2/2)/\sqrt{2\pi}}{\exp(0.4(x-0.4)^2 - 0.08x^4) \cdot \exp(-(y-x)^2/2)/\sqrt{2\pi}}. \tag{18}$$

Often in Metropolis-Hastings sampling, the target distribution ranges over a type α that is a product or sum type, and the proposal kernel is built from sub-kernels on the constituent types. On one hand, when α is a product type $\alpha_1 \times \alpha_2$, a *single-site* kernel $\zeta : \alpha \to \mathbb{M}\,\alpha$ can be built out of sub-kernels $\zeta_1 : \alpha \to \mathbb{M}\,\alpha_1$ and $\zeta_2 : \alpha \to \mathbb{M}\,\alpha_2$ as follows [Goodman et al. 2008; Wingate et al. 2011]: given $(x_1, x_2) : \alpha_1 \times \alpha_2$, flip a coin and either use ζ_1 to update x_1 while keeping x_2 or use ζ_2 to update x_2 while keeping x_1. This composite kernel can be expressed in core Hakaru as

$$\begin{aligned} \zeta \quad = \quad & \lambda(x_1,x_2).\, \tfrac{1}{2} \odot \textbf{do}\ \{x_1' \leftsquigarrow \zeta_1(x_1,x_2);\ \textbf{return}\ (x_1',x_2)\} \\ & \oplus\ \tfrac{1}{2} \odot \textbf{do}\ \{x_2' \leftsquigarrow \zeta_2(x_1,x_2);\ \textbf{return}\ (x_1,x_2')\}. \end{aligned} \tag{19}$$

On the other hand, when α is a sum type $\alpha_1 + \alpha_2$, a *reversible-jump* kernel $\zeta : \alpha \to \mathbb{M}\,\alpha$ can be built out of sub-kernels $\zeta_1 : \alpha_1 \to \mathbb{M}\,\alpha$ and $\zeta_2 : \alpha_2 \to \mathbb{M}\,\alpha$ by case discrimination [Green 1995].

Despite the prevalence of single-site and reversible-jump proposal kernels, existing density calculation procedures cannot find their acceptance ratios, because the necessary base measure is not an independent product but rather a dependent product or disjoint sum respectively. For example, suppose that $\alpha = \mathbb{R}^2$, the target ξ has a density with respect to **lebesgue** $\otimes$ **lebesgue**, and the single-site kernel ζ in (19) is built out of sub-kernels $\zeta_1, \zeta_2 : \mathbb{R}^2 \to \mathbb{M}\,\mathbb{R}$ that always return continuous measures. Still, the measures $\xi \mathrel{\otimes\!\!=} \zeta$ and $\zeta \mathrel{=\!\!\otimes} \xi$ do not have densities with respect to

$$(\textbf{lebesgue} \otimes \textbf{lebesgue}) \otimes (\textbf{lebesgue} \otimes \textbf{lebesgue}) : \mathbb{M}\,(\mathbb{R}^2)^2. \tag{20}$$

Rather, they have densities with respect to the dependent product

$$(\textbf{lebesgue} \otimes \textbf{lebesgue}) \mathrel{\otimes\!\!=} \lambda(x_1,x_2).\, (\textbf{lebesgue} \oplus \textbf{return}\, x_1) \otimes (\textbf{lebesgue} \oplus \textbf{return}\, x_2) : \mathbb{M}\,(\mathbb{R}^2)^2. \tag{21}$$

As for measures over sum types, they call for base measures of the form

$$\mu_1 \oplus \mu_2 \quad = \quad \textbf{do}\ \{x_1 \leftsquigarrow \mu_1;\ \textbf{return}\ (\textbf{inl}\ x_1)\} \oplus \textbf{do}\ \{x_2 \leftsquigarrow \mu_2;\ \textbf{return}\ (\textbf{inr}\ x_2)\}, \tag{22}$$

where the measure μ_1 is over α_1, the measure μ_2 is over α_2, and the measure $\mu_1 \oplus \mu_2$ is over $\alpha_1 + \alpha_2$. Our disintegrator is the first to allow (indeed, infer) dependent products and disjoint sums as base measures, and thus to find these acceptance ratios automatically from programs such as (19).

2.4 Conditioning and its applications

A conditional distribution is a (measurable) function to distributions that is related by monadic bind to a joint distribution and a marginal distribution. Conditional distributions are useful for specifying models, making inferences, and drawing samples. These applications motivate non-continuous marginal distributions.

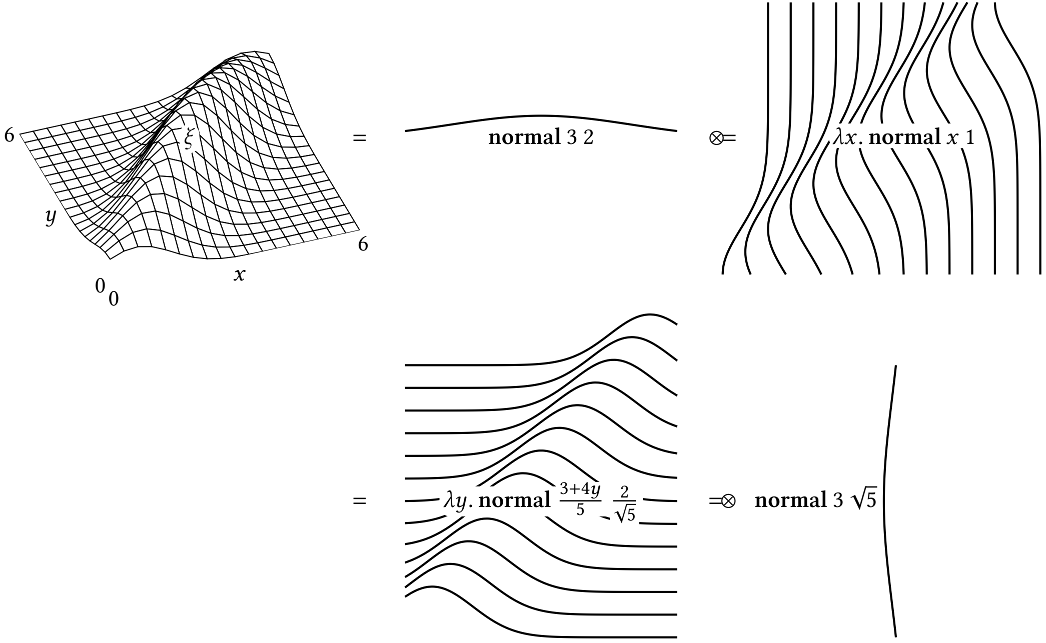

Fig. 6. Conditioning the joint distribution (25) on x (top) and on y (bottom). The left column shows the joint distribution; the middle column shows distributions over x horizontally; the right column shows distributions over y vertically. Each curve depicts a distribution, so each family of curves depicts a family of distributions.

Definition 2.15. Given a joint distribution $\xi : \mathbb{M}\,(\alpha \times \beta)$, the *marginal* distribution $\mathit{fmap}\,\mathit{fst}\,\xi : \mathbb{M}\,\alpha$ over α is defined by

$$\mathit{fmap}\ \mathit{fst}\ \xi \quad = \quad \mathbf{do}\ \{(x,y) \leftsquigarrow \xi;\ \mathbf{return}\ x\} \quad = \quad \lambda f.\, \xi\big(\lambda(x,y).\, f(x)\big). \tag{23}$$

We say that $\kappa : \alpha \to \mathbb{M}\,\beta$ is a *conditional* distribution over β given α if we can decompose ξ as

$$\xi \quad = \quad \mathbf{do}\ \{x \leftsquigarrow \mathit{fmap}\ \mathit{fst}\ \xi;\ y \leftsquigarrow \kappa(x);\ \mathbf{return}\ (x,y)\}, \tag{24}$$

or in short, $\xi = \mathit{fmap}\ \mathit{fst}\ \xi \otimes\!\!= \kappa$. Typically, ξ is normalized. In that case, so are $\mathit{fmap}\ \mathit{fst}\ \xi$ and $\kappa(x)$.

Example 2.16. Suppose we observe a real y drawn randomly from a black box. We believe the black box is **normal** x 1, where x has been drawn from **normal** 3 2, and we wonder what x is. In other words, we want to infer x from a measurement of x that is noisy with standard deviation 1. We write a core Hakaru program to define a joint distribution that models the situation:

$$\xi \quad = \quad \mathbf{do}\ \{x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ y \leftsquigarrow \mathbf{normal}\ x\ 1;\ \mathbf{return}\ (x,y)\} \tag{25}$$

Let us consider in turn the conditional distribution over y given x and that over x given y.

The marginal distribution ($\mathit{fmap}\ \mathit{fst}\ \xi$) over x is

$$\begin{aligned} & \mathbf{do}\ \{x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ y \leftsquigarrow \mathbf{normal}\ x\ 1;\ \mathbf{return}\ x\} \\ = \quad & \mathbf{do}\ \{x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ |\mathbf{normal}\ x\ 1| \odot \mathbf{return}\ x\} \\ = \quad & \mathbf{do}\ \{x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ 1 \odot \mathbf{return}\ x\} \quad = \quad \mathbf{normal}\ 3\ 2. \end{aligned} \tag{26}$$

Thus, we can read off from (25) that the conditional distribution over y given x is just $\lambda x.\,\mathbf{normal}\,x\,1$. This decomposition is shown at the top of Figure 6. Hence, we have already used conditional distributions when specifying the model.

To condition on y, we use equation (5) to rewrite (25), commuting the binding of y to the front:

$$\begin{aligned}\xi &= \textbf{do}\ \{x \leftsquigarrow \textbf{normal}\ 3\ 2;\ y \leftsquigarrow \textbf{lebesgue};\ (\textbf{dnorm}\ x\ 1)(y) \odot \textbf{return}\ (x,y)\} \\ &= \textbf{do}\ \{y \leftsquigarrow \textbf{lebesgue};\ x \leftsquigarrow \textbf{normal}\ 3\ 2;\ (\textbf{dnorm}\ x\ 1)(y) \odot \textbf{return}\ (x,y)\} \\ &= \kappa' \mathrel{=\!\!\otimes} \textbf{lebesgue} \qquad \text{where } \kappa' = \lambda y.\ \textbf{do}\ \{x \leftsquigarrow \textbf{normal}\ 3\ 2;\ (\textbf{dnorm}\ x\ 1)(y) \odot \textbf{return}\ x\}\end{aligned} \tag{27}$$

Thus, the marginal distribution (*fmap snd* ξ) over y is $(\lambda y.\,|\kappa'(y)|) \mathrel{=\!\!\odot}$ **lebesgue**, and the conditional distribution over x given y is $\lambda y.\,|\kappa'(y)|^{-1} \odot \kappa'(y)$, the normalization of κ'. For this model, the marginal turns out equal to **normal** $3\ \sqrt{5}$ and the conditional turns out equal to $\lambda y.$ **normal** $\frac{3+4y}{5}\ \frac{2}{\sqrt{5}}$. This decomposition is shown at the bottom of Figure 6. Hence, we have used our observation y to update our *prior* belief about x, namely **normal** 3 2, and form our *posterior* belief about x, namely **normal** $\frac{3+4y}{5}\ \frac{2}{\sqrt{5}}$. This update illustrates the use of conditional distributions for making inferences.

The choice of the base measure **lebesgue** in the calculation (27) is conventional because marginal distributions are commonly continuous, but arbitrary. The conditioning only depends on the normalization of κ', which is the same regardless of the base measure, as long as the marginal has a density with respect to the base. For example, if we double **lebesgue** as the base measure, then κ' would be halved, but its normalization would remain the same.

Example 2.17. The reasoning in Example 2.16 can just as well be used to make inferences from non-continuous observations. For example, to update our belief about x using an observation of the clamping of **normal** x 1 to the interval $[0,1]$, we can condition the model

$$\textbf{do}\ \{x \leftsquigarrow \textbf{normal}\ 3\ 2;\ y \leftsquigarrow \textbf{normal}\ x\ 1;\ \textbf{return}\ (x, \max\{0, \min\{1, y\}\})\} \tag{28}$$

on y like in (27), but using the base measure in (10) rather than **lebesgue**. This clamped model is a simple instance of a *Tobit model* [Tobin 1958]. Our disintegrator is the first transformation to automate this reasoning, because the clamped marginal over y is not continuous.

Example 2.18. Like density, conditioning is also useful for drawing samples from a given target distribution. In particular, *Gibbs sampling* [Geman and Geman 1984; Gelfand et al. 1992] is a popular inference technique on joint distributions that requires drawing repeatedly from its conditional distributions. Again, our disintegrator allows the base measure to vary from the Lebesgue measure, in order to condition a distribution whose marginals are non-continuous, such as a Tobit model.

2.5 Disintegration

We have defined density and conditioning and illustrated their applications using small examples and varying base measures. Disintegration generalizes density and conditioning.

Definition 2.19. A *disintegration* of a joint measure $\xi : \mathbb{M}\,(\alpha \times \beta)$ is a *base measure* $\mu : \mathbb{M}\,\alpha$ and a *kernel* $\kappa : \alpha \to \mathbb{M}\,\beta$ such that $\xi = \mu \mathrel{\otimes\!\!=} \kappa$.

It is easy to see that disintegration generalizes conditioning (Definition 2.15): let $\mu =$ *fmap fst* ξ. To see that disintegration also generalizes density (Definition 2.3), let β be the unit type $\mathbb{1}$ and note that not only are the types α and $\alpha \times \mathbb{1}$ isomorphic, but $\mathbb{R}_+$ and $\mathbb{M}\,\mathbb{1}$ are also isomorphic: map $r : \mathbb{R}_+$ to $r \odot$ **return** $() : \mathbb{M}\,\mathbb{1}$ and map $\nu : \mathbb{M}\,\mathbb{1}$ to $|\nu| : \mathbb{R}_+$.

Disintegration is well known as a useful measure-theoretic relation. Shan and Ramsey [2017] showed that disintegration is also a useful probabilistic-program transformation, but only automated it for base measures that are independent products of Lebesgue and counting measures. Narayanan and Shan [2017] generalized those independent products to handle arrays without unrolling. Any disintegration program transformation can also be used to find densities, because the totaling map $\lambda v.\,|v|$ is easy to implement as a program transformation, though it can produce integrals and sums that witness the fundamental intractability of probabilistic inference.

Theorems about the existence and uniqueness of densities and disintegrations are a mainstay of measure theory [Dieudonné 1948; Chang and Pollard 1997; Ackerman et al. 2011, 2016]. Recently, Ong and Vákár [2018] proved theorems about the existence of densities and disintegrations for *s-finite kernels*. Our disintegrator falls under a simple special case of those theorems, because it handles s-finite measures with respect to σ-finite base measures. Unfortunately, like previous disintegrators, our program transformation is proven sound but far from complete: there are easy ways to stymie it intentionally, and we demonstrate its utility only empirically (Section 8).

3 OVERVIEW OF OUR APPROACH

This paper presents a new disintegrator that allows the base measure $\mu : \mathbb{M}\,\alpha$ to vary rather than being determined by the type α. We handle base measures that are sums of **lebesgue** and **return** as in (10), dependent products as in (21), and disjoint sums as in (22). In this section, we give an overview of our approach, then review the syntax of our object language.

Strictly speaking, variable-base disintegration can be two program transformations:

- The *base-checking disintegrator* takes the input programs $\xi : \mathbb{M}\,(\alpha \times \beta)$ and $\mu : \mathbb{M}\,\alpha$ and returns a set of output programs $\kappa : \alpha \to \mathbb{M}\,\beta$ such that $\xi = \mu \otimes\!\!= \kappa$.
- The *base-inferring disintegrator* takes the input program $\xi : \mathbb{M}\,(\alpha \times \beta)$ and returns a set of pairs of output programs $\mu : \mathbb{M}\,\alpha$ and $\kappa : \alpha \to \mathbb{M}\,\beta$ such that $\xi = \mu \otimes\!\!= \kappa$.

Both disintegrators are useful. We build them by progressively defining four transformations.

First, in Section 4, we refactor Shan and Ramsey's original disintegrator [2017] as

- a language of base measures $\mathbb{B}\,\alpha$ that is a restricted subset of $\mathbb{M}\,\alpha$, and
- a restricted checking disintegrator, which takes the input program $\xi : \mathbb{M}\,(\alpha \times \beta)$ and (instead of $\mu : \mathbb{M}\,\alpha$) the base measure $b : \mathbb{B}\,\alpha$.

This initial base-measure language is so restricted that for any given type α there exists a unique base measure $genBase(\alpha) : \mathbb{B}\,\alpha$. This base measure is Bhat et al.'s *stock measure* [2012].

Second, in Section 5, we extend the base-measure language by

- replacing **lebesgue** by sums of **lebesgue** and **return**, and
- replacing independent products $\otimes$ by dependent products $\otimes\!\!=$.

After this extension, for any given type α the base measures $\mathbb{B}\,\alpha$ may not be unique but form a preorder. We extend the restricted checking disintegrator to handle the extended base-measure language while respecting this preorder.

In Section 6, to build a base-inferring disintegrator, we add $\mathbb{B}\,\mathbb{R}$ variables to the base-measure language and update the *genBase* function to produce them. The restricted checking disintegrator turns a base measure with variables into constraints on the variables. Our base-inferring disintegrator solves these constraints to get a *principal base measure* $b : \mathbb{B}\,\alpha$ and returns it as $\mu : \mathbb{M}\,\alpha$.

Finally, in Section 7 we build an unrestricted base-checking disintegrator. It invokes the base-inferring disintegrator on $\mu : \mathbb{M}\,\alpha$ to infer $b : \mathbb{B}\,\alpha$. It then disintegrates both ξ and μ with respect to b, and divides the results to cancel b out and produce κ.

3.1 Program syntax

Figure 7 shows the syntax of core Hakaru, the input and output language [Shan and Ramsey 2017].

The definition of terms at the top of Figure 7 does not include many operations on reals, because we use **sqrt** (square root) to illustrate how in general to handle invertible functions (such as negation, reciprocal, exp, log). Similarly, we use squaring ^{2} and binary multiplication $\cdot$ to illustrate how to handle piecewise-invertible and coordinatewise-invertible functions (such as absolute value and +).

Terms	$e, m, M ::= v \mid x \mid \textbf{sqrt}\ e \mid e^2 \mid e \cdot e \mid e < e \mid \textbf{fst}\ e \mid \textbf{snd}\ e$
Head normal forms	$v ::= u \mid \textbf{lebesgue} \mid \textbf{return}\ e \mid \textbf{fail} \mid m \oplus m \mid \textbf{do}\ \{g; M\}$
	$\mid r \mid () \mid (e, e) \mid \textbf{inl}\ e \mid \textbf{inr}\ e$
Atomic terms	$u ::= z$ (not heap-bound) $\mid \textbf{sqrt}\ u \mid u^2 \mid u \cdot u \mid u \cdot r \mid r \cdot u$
	$\mid u < u \mid u < r \mid r < u \mid \textbf{fst}\ u \mid \textbf{snd}\ u$
Real numbers	$r \in \mathbb{R}$
Variables	x, y, z
Bindings (guards)	$g ::= x \leftsquigarrow m \mid \textbf{factor}\ e \mid \textbf{let inl}\ x = e \mid \textbf{let inr}\ x = e$
Heaps	$h ::= [g; \ldots; g]$
Types	$\alpha, \beta, \gamma ::= \mathbb{R} \mid \mathbb{1} \mid \alpha \times \beta \mid \alpha + \beta \mid \mathbb{M}\,\alpha$
Bases	$b ::= \textbf{lebesgue} \mid \textbf{return}\ () \mid b \otimes b \mid b \oplus b$

Fig. 7. The syntax of core Hakaru

$$\frac{}{\textbf{lebesgue} : \mathbb{M}\,\mathbb{R}} \qquad \frac{e : \gamma}{\textbf{return}\ e : \mathbb{M}\,\gamma} \qquad \frac{}{\textbf{fail} : \mathbb{M}\,\gamma} \qquad \frac{m_1 : \mathbb{M}\,\gamma \quad m_2 : \mathbb{M}\,\gamma}{m_1 \oplus m_2 : \mathbb{M}\,\gamma}$$

$$\frac{m : \mathbb{M}\,\alpha \quad \begin{matrix}[x:\alpha]\\ \vdots \\ M : \mathbb{M}\,\gamma\end{matrix}}{\textbf{do}\ \{x \leftsquigarrow m; M\} : \mathbb{M}\,\gamma} \qquad \frac{e : \mathbb{R} \quad M : \mathbb{M}\,\gamma}{e \odot M : \mathbb{M}\,\gamma} \qquad \frac{e : \alpha + \beta \quad \begin{matrix}[x:\alpha]\\ \vdots \\ M : \mathbb{M}\,\gamma\end{matrix}}{\textbf{do}\ \{\textbf{let inl}\ x = e; M\} : \mathbb{M}\,\gamma} \qquad \frac{e : \alpha + \beta \quad \begin{matrix}[x:\beta]\\ \vdots \\ M : \mathbb{M}\,\gamma\end{matrix}}{\textbf{do}\ \{\textbf{let inr}\ x = e; M\} : \mathbb{M}\,\gamma}$$

Fig. 8. Typing rules for measure terms. The measure term $e \odot M$ abbreviates $\textbf{do}\ \{\textbf{factor}\ e; M\}$.

The disintegrator distinguishes certain terms as *head normal forms*, and certain head normal forms as *atomic terms*. An atomic term u is either a variable z for which no information can be found in the *heap* maintained by the disintegrator, or built up from such a variable by applying strict functions. A head normal form v is either an atomic term u or a constructor application. In particular, head normal forms include measure terms and real literals.

By way of explaining the measure terms, Figure 8 shows their typing rules; the rest of the type system is standard and elided. Our types are simple: as defined in Figure 7, they are the type of reals $\mathbb{R}$, the unit type $\mathbb{1}$, product types $\times$, sum types $+$, and measure types $\mathbb{M}$. Each type denotes a measurable space; in particular, the type $\mathbb{M}\,\alpha$ denotes the measurable space of (s-finite [Staton 2017]) measures over α.

The measure terms **lebesgue**, **return** e, **fail**, and $m_1 \oplus m_2$ denote respectively the Lebesgue measure, the Dirac measure at e (in other words, monadic unit), the zero measure, and the sum of the measures m_1 and m_2. The remaining measure terms have the form $\textbf{do}\ \{g; M\}$, where g is one of several kinds of *bindings*, also called *guards*. The typical such measure term is monadic bind, $\textbf{do}\ \{x \leftsquigarrow m; M\}$, where m and M are measure terms and x takes scope over M. If x is not used in M, then instead of x we can write _, or write () if x has type $\mathbb{1}$. Another kind of guard builds the measure term $\textbf{do}\ \{\textbf{factor}\ e; M\}$, which means to scale the measure M by the weight e. We write this term as $e \odot M$ for short. We also write $\textbf{normal}\ e_1\ e_2$ as syntactic sugar for $\textbf{dnorm}\ e_1\ e_2 \mathrel{=\!\!\odot} \textbf{lebesgue}$.

Following Shan and Ramsey [2017], to make the disintegrator easier to explain, sum types in core Hakaru are deconstructed by bindings like $\textbf{let inl}\ x = e$, which may fail. If e is $\textbf{inl}\ e_1$, then

the term **do** {**let inl** $x = e$; M} just means $M\{x \mapsto e_1\}$, but if e is **inr** e_2, then the same term **do** {**let inl** $x = e$; M} means the zero measure **fail**. Ordinary pattern matching on sum types can be recovered by duplicating a measure context $M[\ ]$:

$$M\begin{bmatrix}\textbf{case}\ e\ \textbf{of}\ \textbf{inl}\ x_1 \to e_1 \\ \qquad\qquad \textbf{inr}\ x_2 \to e_2\end{bmatrix} \;=\; \textbf{do}\ \{\textbf{let inl}\ x_1 = e;\ M[e_1]\} \oplus \textbf{do}\ \{\textbf{let inr}\ x_2 = e;\ M[e_2]\}. \tag{29}$$

Booleans **true**, **false** can be encoded using the sum type $bool = \mathbb{1}+\mathbb{1}$ as usual, and Boolean operations can be encoded in terms of **case**, so numeric comparisons such as equality can be encoded in terms of $<$. If e is a Boolean expression, we write **observe** e to mean the guard **let true** $= e$.

A *heap* is a sequence of zero or more bindings, each taking scope to its right. The disintegrator uses heaps to maintain information about random variables. We also use heaps to define the usual syntactic sugar for nested bindings, which wraps a heap around a measure term:

$$\textbf{do}\ \{g_1; \ldots; g_n; M\} \;=\; \textbf{do}\ \{g_1; \ldots \textbf{do}\ \{g_n; M\} \ldots\}. \tag{30}$$

3.2 Base-measure language

The bottom of Figure 7 defines a language of base measures. For each type α, the language $\mathbb{B}\,\alpha$ of base measures b over α is a restricted subset of the language $\mathbb{M}\,\alpha$ of measures over α. In fact, this base language is so restricted that there is only one base measure per type. To wit, the function *genBase* defined below maps each non-measure type α to its unique base measure $genBase(\alpha) : \mathbb{B}\,\alpha$:

$$\begin{aligned} genBase(\mathbb{R}) &= \textbf{lebesgue} \qquad & genBase(\alpha \times \beta) &= genBase(\alpha) \otimes genBase(\beta) \\ genBase(\mathbb{1}) &= \textbf{return}\ () \qquad & genBase(\alpha + \beta) &= genBase(\alpha) \oplus genBase(\beta) \end{aligned} \tag{31}$$

This base language is new. The goal of this paper is to relax the restriction it represents.

4 A FIXED-BASE DISINTEGRATOR

In this section, we refactor Shan and Ramsey's original disintegrator [2017] to make explicit the base measures it fixes and to isolate the few places where it operates on a base measure over $\mathbb{R}$. The refactoring hence eases the remainder of our development. We also extend the disintegrator to handle base measures that are disjoint sums ($b ::= b \oplus b$ in Figure 7).

Figure 9 shows the top four functions that make up our restricted base-checking disintegrator. These are meta-functions that operate on object syntax; after all, functions are not values in core Hakaru itself. Each function comes with an informal type, a semantic specification (boxed), and an implementation.

At the top of Figure 9 is *check*, the external interface. Unpacking the new notation used in *check* reveals structure that recurs throughout Shan and Ramsey's and our disintegrators.

Meta types. On the second line of Figure 9, the (meta) type $\lceil \mathbb{M}\,(\alpha \times \beta) \rceil$ means a core Hakaru term of type $\mathbb{M}\,(\alpha \times \beta)$, and the (meta) type $\lfloor \alpha \rfloor$ means a head normal form of type α.

Nondeterminism. The return type of *check* is $\{\lfloor \mathbb{M}\,\beta \rfloor\}$, which means a set (of head normal forms of type $\mathbb{M}\,\beta$). This set type reveals that the disintegrator is nondeterministic: it tries multiple ways to disintegrate a program and may end up with zero, one, or more results. Because this nondeterminism pervades the disintegrator, our notation shows sets explicitly in types but builds sets implicitly in terms. For example, in the specification for *check* boxed in the upper-right corner of Figure 9, the symbol $\sqsupseteq$ appears to relate two terms but actually relates two sets of terms. The right-hand side is the set of all terms **do** $\{t \leftsquigarrow b; p \leftsquigarrow e;$ **return** $(t, p)\}$ where the subterm e belongs to the set *check* $m\ b\ t$. The left-hand side is the singleton set containing the term m.

Check base $\quad\boxed{m \sqsupseteq \mathbf{do}\ \{t \leftsquigarrow b;\ p \leftsquigarrow check\ m\ b\ t;\ \mathbf{return}\ (t,p)\}}$

$$check : \lceil \mathbb{M}\,(\alpha \times \beta) \rceil \to \mathbb{B}\,\alpha \to \lfloor \alpha \rfloor \to \{\lfloor \mathbb{M}\,\beta \rfloor\}$$

$$check\ m\ b\ t = \ggg\ m\ \big(\lambda v.\ \lhd\ (\mathbf{fst}\ v)\ b\ t\ \overline{(\mathbf{return}\ (\mathbf{snd}\ v))}\big)\ []$$

Constrain value $\quad\boxed{\mathbf{do}\ \{h;\ t \leftsquigarrow \mathbf{return}\ e;\ M\} \sqsupseteq \mathbf{do}\ \{t \leftsquigarrow b;\ \lhd\ e\ b\ t\ \overline{M}\ h\}}$

$$\lhd : \lceil \alpha \rceil \to \mathbb{B}\,\alpha \to \lfloor \alpha \rfloor \to (heap \to \{\lfloor \mathbb{M}\,\gamma \rfloor\}) \to heap \to \{\lfloor \mathbb{M}\,\gamma \rfloor\}$$

$$\begin{array}{llll}
\lhd\ e & (\mathbf{return}\ ())\ v\ c\ h & = c\ h & \\
\lhd\ e & (b_1 \otimes b_2)\quad v\ c\ h & = \lhd\ (fst\ e)\ b_1\ (fst\ v)\ \big(\lhd\ (snd\ e)\ b_2\ (snd\ v)\ c\big)\ h & \\
\lhd\ e & (b_1 \oplus b_2)\quad v\ c\ h & = outl\ v\ \big(\lambda e_0.\ \rhd\ e_0\ (\lambda v_0.\ \lhd\ x\ b_1\ v_0\ c)\big)\ [h;\ \mathbf{let\ inl}\ x = e] & \\
 & & \quad \oplus\ outr\ v\ \big(\lambda e_0.\ \rhd\ e_0\ (\lambda v_0.\ \lhd\ x\ b_2\ v_0\ c)\big)\ [h;\ \mathbf{let\ inr}\ x = e] & \\
\lhd\ u & b\quad v\ c\ h & = \bot & \text{where } u \text{ is atomic} \\
\lhd\ r & b\quad v\ c\ h & = \bot & \text{where } r \text{ is a literal real number} \\
\lhd\ (\mathbf{sqrt}\ e)\ b & v\ c\ h & = \lessdot\ \mathbf{sqrt}^+\ e\ b\ v\ c\ h & \\
\lhd\ e^2 & b\quad v\ c\ h & = \lessdot\ \mathbf{sqr}^+\ e\ b\ v\ c\ h \oplus \lessdot\ \mathbf{sqr}^-\ e\ b\ v\ c\ h & \\
\lhd\ (e_1 \cdot e_2)\ b & v\ c\ h & = \rhd\ e_1\ \big(\lambda v_1.\ \lessdot\ (\mathbf{mul}\ v_1)\ e_2\ b\ v\ c\big)\ h \sqcup \rhd\ e_2\ \big(\lambda v_2.\ \lessdot\ (\mathbf{mul}\ v_2)\ e_1\ b\ v\ c\big)\ h & \\
\lhd\ (\mathbf{fst}\ e) & b\quad v\ c\ h & = \rhd\ e\ \big(\lambda v_0.\ \lhd\ (fst\ v_0)\ b\ v\ c\big)\ h & \text{unless } e \text{ is atomic} \\
\lhd\ (\mathbf{snd}\ e) & b\quad v\ c\ h & = \rhd\ e\ \big(\lambda v_0.\ \lhd\ (snd\ v_0)\ b\ v\ c\big)\ h & \text{unless } e \text{ is atomic}
\end{array}$$

$$\begin{array}{ll}
\lhd\ x\ b\ v\ c\ [h_1;\ x \leftsquigarrow m;\ h_2] & = \lll\ m\ b\ v\ (\overline{[x \leftsquigarrow \mathbf{return}\ v;\ h_2]} \mathbin{⨟} c)\quad h_1 \\
\lhd\ x\ b\ v\ c\ [h_1;\ \mathbf{let\ inl}\ x = e;\ h_2] & = \rhd\ e\ \big(\lambda v_0.\ outl\ v_0\ (\lambda e_0.\ \lhd\ e_0\ b\ v\ (\overline{[x \leftsquigarrow \mathbf{return}\ v;\ h_2]} \mathbin{⨟} c))\big)\ h_1 \\
\lhd\ x\ b\ v\ c\ [h_1;\ \mathbf{let\ inr}\ x = e;\ h_2] & = \rhd\ e\ \big(\lambda v_0.\ outr\ v_0\ (\lambda e_0.\ \lhd\ e_0\ b\ v\ (\overline{[x \leftsquigarrow \mathbf{return}\ v;\ h_2]} \mathbin{⨟} c))\big)\ h_1
\end{array}$$

Constrain outcome $\quad\boxed{\mathbf{do}\ \{h;\ t \leftsquigarrow m;\ M\} \sqsupseteq \mathbf{do}\ \{t \leftsquigarrow b;\ \lll\ m\ b\ t\ \overline{M}\ h\}}$

$$\lll : \lceil \mathbb{M}\,\alpha \rceil \to \mathbb{B}\,\alpha \to \lfloor \alpha \rfloor \to (heap \to \{\lfloor \mathbb{M}\,\gamma \rfloor\}) \to heap \to \{\lfloor \mathbb{M}\,\gamma \rfloor\}$$

$$\begin{array}{llll}
\lll\ u & b\ v\ c\ h & = \bot & \text{where } u \text{ is atomic} \\
\lll\ \mathbf{lebesgue} & b\ v\ c\ h & = \mathbf{do}\ \{() \leftsquigarrow (\mathbf{lebesgue} \div b)\ v;\ c\ h\} & \\
\lll\ (\mathbf{return}\ e) & b\ v\ c\ h & = \lhd\ e\ b\ v\ c\ h & \\
\lll\ \mathbf{fail} & b\ v\ c\ h & = \mathbf{fail} & \\
\lll\ (m_1 \oplus m_2) & b\ v\ c\ h & = \lll\ m_1\ b\ v\ c\ h \oplus \lll\ m_2\ b\ v\ c\ h & \\
\lll\ (\mathbf{do}\ \{g;\ m\})\ b\ v\ c\ h & & = \lll\ m\ b\ v\ c\ [h;\ g] & \\
\lll\ e & b\ v\ c\ h & = \rhd\ e\ (\lambda m.\ \lll\ m\ b\ v\ c)\ h & \text{where } e \text{ is not in head normal form}
\end{array}$$

Constrain op $\quad\boxed{\mathbf{do}\ \{h;\ \mathbf{observe}\ (e \in dom\ f);\ t \leftsquigarrow \mathbf{return}\ (f\ @\ e);\ M\} \sqsupseteq \mathbf{do}\ \{t \leftsquigarrow b;\ \lessdot\ f\ e\ b\ t\ \overline{M}\ h\}}$

$$\lessdot : invertible \to \lceil \mathbb{R} \rceil \to \mathbb{B}\,\mathbb{R} \to \lfloor \mathbb{R} \rfloor \to (heap \to \{\lfloor \mathbb{M}\,\gamma \rfloor\}) \to heap \to \{\lfloor \mathbb{M}\,\gamma \rfloor\}$$

$$\lessdot\ f\ e\ b\ v\ c\ h = \mathbf{do}\ \{\mathbf{observe}\ (v \in rng\ f);\ \mathbf{factor}\ (jacobian\ f\ b\ v);\ \lhd\ e\ (reparam\ f\ b)\ (inv\ f\ @\ v)\ c\ h\}$$

Fig. 9. The restricted base-checking disintegrator

Definition 4.1. Let S and T be two sets of core Hakaru terms. We write $S \sqsupseteq T$ to mean that the set of denotations of elements of S is a superset of the set of denotations of elements of T. In other words, every term in T denotes the same as some term in S.

Thus, the specification for *check* can be paraphrased using the fact that the left-hand side m is a singleton set: for every term e returned by $\mathit{check}\ m\ b\ t$ (if any), the denotation of $\mathbf{do}\ \{t \leftsquigarrow b;\ p \leftsquigarrow e;\ \mathbf{return}\ (t, p)\}$ equals the denotation of m.

In other definitions (such as the next function $\lhd$ in Figure 9), we write $\bot$ for the empty set of terms and $\sqcup$ for the union of two sets of terms. These are the only two ways to incur nondeterminism. Note that $\bot$ is different from **fail**, which is (a singleton set of) a term that denotes the zero measure, and $\sqcup$ is different from $\oplus$, which constructs a term that sums two measures.

Continuations. The disintegrator is written in continuation-passing style: many functions take and return continuations of type $\mathit{heap} \to \{\lfloor \mathbb{M}\,\gamma \rfloor\}$ and maintain the *heap* as a piece of mutable state. We use the metavariable c for these continuations. They typically have the form $\overline{M}$, defined as the function

$$\overline{M} \quad = \quad \lambda h.\, \mathbf{do}\ \{h;\ M\}, \tag{32}$$

which implicitly builds a singleton set by wrapping the given heap h around some measure term M. Thus, the third line of Figure 9 defines *check* as the following steps in continuation-passing style: given a joint distribution m, a base measure b, and an index t (usually a variable representing an observation), the function *check* initializes the heap to the empty $[]$, invokes $\rhd$ on m to get v, invokes $\lhd$ on $\mathbf{fst}\ v$, and wraps the final heap around $\overline{\mathbf{return}\ (\mathbf{snd}\ v)}$ to form the result.

In other definitions (such as the function $\lhd$ in Figure 9), we write $\overline{h_0}$ for the heap-to-heap function

$$\overline{h_0} \quad = \quad \lambda h.\, [h;\ h_0], \tag{33}$$

which wraps the given heap h around some heap h_0. We also define the reverse function-composition operator $\fatsemi$ by $(g \fatsemi c)(h) = c(g(h))$, so that $\overline{h_0} \fatsemi \overline{M} = \overline{\mathbf{do}\ \{h_0;\ M\}}$. To draw an example from the definition of $\lhd$, if $c = \overline{M}$ then $\overline{[x \leftsquigarrow \mathbf{return}\ v;\ h_2]} \fatsemi c = \overline{\mathbf{do}\ \{x \leftsquigarrow \mathbf{return}\ v;\ h_2;\ M\}}$.

From specifications to implementations. All the specifications boxed in Figure 9 have the form

$$\boxed{\cdots \quad \sqsupseteq \quad \mathbf{do}\ \{t \leftsquigarrow b;\ \ldots\}}, \tag{34}$$

where the right-hand side calls the function being specified and the left-hand side is made of inputs to the function. This pattern reflects the fact that the main job of the disintegrator is to rewrite a given measure term (the left-hand side) to a semantically equivalent one (the right-hand side) that begins with the binding $t \leftsquigarrow b$. Whereas the external interface *check* rewrites an arbitrary measure term m, the other functions $\lhd$ ("constrain value"), $\lll$ ("constrain outcome"), and $\lessdot$ ("constrain op") rewrite measure terms of more specific forms, focusing on an expression e, action m, or operation f in the context of a heap h and final action M.

All the implementations in Figure 9 are derived from the semantic specifications by equational reasoning. (Some technicalities of the proofs are discussed in Section 4.3.) For example, the case $\lll$ **fail** is derived and proven from the specification

$$\boxed{\mathbf{do}\ \{h;\ t \leftsquigarrow m;\ M\} \quad \sqsupseteq \quad \mathbf{do}\ \{t \leftsquigarrow b;\ \lll m\ b\ t\ \overline{M}\ h\}} \tag{35}$$

by substituting **fail** for m and equating both sides to **fail**. Similarly, the case $\lll (m_1 \oplus m_2)$ is derived and proven from (35) by substituting $m_1 \oplus m_2$ for m and using the induction hypotheses derived also from (35) by substituting m_1 and m_2 for m. An example where one function calls another is the case $\lll (\mathbf{return}\ e)$, which is derived and proven from (35) by substituting $\mathbf{return}\ e$ for m and using

$$\text{Invertibles} \qquad f ::= \mathbf{mul}\ v \mid \mathbf{div}\ v \mid \mathbf{sqr}^{+} \mid \mathbf{sqr}^{-} \mid \mathbf{sqrt}^{+} \mid \mathbf{sqrt}^{-}$$

Domain

$$\begin{aligned}
(\in\!dom) &: \lceil\mathbb{R}\rceil \to invertible \to \lceil bool \rceil\\
e \in\!dom\ (\mathbf{mul}\ v) &= \mathbf{true}\\
e \in\!dom\ (\mathbf{div}\ v) &= \mathbf{true}\\
e \in\!dom\ \mathbf{sqr}^{+} &= 0 \le e\\
e \in\!dom\ \mathbf{sqr}^{-} &= e < 0\\
e \in\!dom\ \mathbf{sqrt}^{+} &= 0 \le e\\
e \in\!dom\ \mathbf{sqrt}^{-} &= 0 < e
\end{aligned}$$

Apply invertible

$$\begin{aligned}
(@) &: invertible \to \lceil\mathbb{R}\rceil \to \lceil\mathbb{R}\rceil\\
\mathbf{mul}\ v\ @\ e &= v \cdot e\\
\mathbf{div}\ v\ @\ e &= v^{-1} \cdot e\\
\mathbf{sqr}^{+}\ @\ e &= e^{2}\\
\mathbf{sqr}^{-}\ @\ e &= e^{2}\\
\mathbf{sqrt}^{+}\ @\ e &= \mathbf{sqrt}\ e\\
\mathbf{sqrt}^{-}\ @\ e &= -\mathbf{sqrt}\ e
\end{aligned}$$

Invert $\boxed{(inv\ f@) = (f@)^{-1}}$

$$\begin{aligned}
inv &: invertible \to invertible\\
inv\ (\mathbf{mul}\ v) &= \mathbf{div}\ v\\
inv\ (\mathbf{div}\ v) &= \mathbf{mul}\ v\\
inv\ \mathbf{sqr}^{+} &= \mathbf{sqrt}^{+}\\
inv\ \mathbf{sqr}^{-} &= \mathbf{sqrt}^{-}\\
inv\ \mathbf{sqrt}^{+} &= \mathbf{sqr}^{+}\\
inv\ \mathbf{sqrt}^{-} &= \mathbf{sqr}^{-}
\end{aligned}$$

Range

$$\begin{aligned}
(\in\!rng) &: \lceil\mathbb{R}\rceil \to invertible \to \lceil bool \rceil\\
e \in\!rng\ f &= e \in\!dom\ (inv\ f)
\end{aligned}$$

Apply derivative

$$diff : invertible \to \lfloor\mathbb{R}\rfloor \to \lfloor\mathbb{R}\rfloor \qquad \boxed{diff\ f\ v = \left.\frac{d(f\ @\ x)}{dx}\right|_{x=v}}$$

Fig. 10. Invertible functions and operations on them

the induction hypothesis that is the specification of $\lhd$. Similarly, the definition of top-level *check* is derived and proven from the specifications of $\rhd\!\!\rhd$ and $\lhd$.

4.1 Constraining with respect to a base measure

Three crucial aspects of this disintegrator are new compared to Shan and Ramsey's.

First, we add a base-measure argument $b : \mathbb{B}\ \alpha$ to every function in Figure 9. The case $\lhd e(\mathbf{return}())$ fleshes out Shan and Ramsey's remark that "on countable spaces, life is easy." The case $\lhd\ e\ (b_1 \otimes b_2)$ fleshes out their remark that "other types α such as $\alpha = \mathbb{R} \times \mathbb{R}$ can be handled by successive disintegration." The case $\lhd\ e\ (b_1 \oplus b_2)$ adds handling for disjoint sums, exactly as defined and motivated in equation (22). The fact that only $\lhd$ inspects b, and only if α is not $\mathbb{R}$, helps us add $\mathbb{B}\,\mathbb{R}$ variables to the base-measure language in Section 6.

Second, we introduce the function $\lessdot$ ("constrain op") to encapsulate a repeated pattern in how *invertible* operations are handled. Figure 10 defines *invertibles*, a simple data type used by the disintegrator internally to represent possibly-partial functions from $\mathbb{R}$ to $\mathbb{R}$ that are invertible and differentiable. The functions $\in\!dom$ and $@$ define the meaning of each invertible: its domain, and its value at each point in the domain. The other functions are semantically specified and equationally derived as usual. When $\lhd$ encounters an operation that is invertible, such as $\mathbf{sqrt}\ e = \mathbf{sqrt}^{+}\ @\ e$, it hands it to $\lessdot$; when $\lhd$ encounters an operation that is piecewise-invertible, such as e^2 (which can be decomposed into two invertible pieces, namely squaring a non-negative number and squaring a negative number), it hands each invertible piece to $\lessdot$.

Third, even though the Lebesgue measure is the only base measure over $\mathbb{R}$ handled so far, we localize our reasoning about it to three new auxiliary functions, to help us add more base measures over $\mathbb{R}$ in Section 5. The functions *jacobian* and *reparam* reparameterize a base measure over $\mathbb{R}$ by an invertible and produce a Jacobian factor and a new base measure; these functions are used by $\lessdot$. The function $\div$ ("divide") finds a density of one base measure over $\mathbb{R}$ with respect to another; it is used by $\lll$. We now turn to these three functions. Figure 11 shows their informal types, semantic specifications (boxed), and implementations.

Reparameterize a base measure with a Jacobian

$jacobian : invertible \to \mathbb{B}\,\mathbb{R} \to \lfloor\mathbb{R}\rfloor \to \lfloor\mathbb{R}\rfloor$

$reparam : invertible \to \mathbb{B}\,\mathbb{R} \to \mathbb{B}\,\mathbb{R}$

$jacobian\ f\ \mathbf{lebesgue} = \lambda v.\, |diff\ (inv\ f)\ v|$

$reparam\ f\ \mathbf{lebesgue} = \mathbf{lebesgue}$

$$\begin{aligned} &\mathbf{do}\ \{x \leftsquigarrow reparam\ f\ b;\ \mathbf{observe}\ (x \in dom\ f);\\ &\quad \mathbf{return}\ x\}\}\\ = {}&\mathbf{do}\ \{y \leftsquigarrow b;\ \mathbf{observe}\ (y \in rng\ f);\\ &\quad \mathbf{factor}\ (jacobian\ f\ b\ y);\\ &\quad \mathbf{return}\ (inv\ f\ @\ y)\}\end{aligned}$$

Divide base measures

$(\div) : \mathbb{B}\,\mathbb{R} \to \mathbb{B}\,\mathbb{R} \to \{\lfloor\mathbb{R}\rfloor \to \mathbb{M}\,\mathbb{1}\}$

$\mathbf{lebesgue} \div \mathbf{lebesgue} = \lambda v.\, \mathbf{return}\ ()$

$$b_1 \sqsupseteq \mathbf{do}\ \{t \leftsquigarrow b_2;\ () \leftsquigarrow (b_1 \div b_2)\ t;\ \mathbf{return}\ t\}$$

Fig. 11. Operations on the Lebesgue base measure

4.1.1 Reparameterizing base measures. Many realistic models invoke deterministic mathematical operations over $\mathbb{R}$, such as curving a random student grade by taking its square root, combining random particle momenta by summing them [Afshar et al. 2016], or just converting a random measurement from Celsius to Fahrenheit. To constrain an expression that invokes such operations, $\lhd$ calls $\lessdot$, which in turn calls *jacobian* and *reparam*, defined in Figure 11.

To see how handling deterministic mathematical operations amounts to reparameterizing a base measure and changing an integration variable, suppose we want the density of the distribution that is like **normal** 0 1 but with its outcome multiplied by 5. According to equation (6), we seek $\kappa : \mathbb{R} \to \mathbb{R}_+$ such that

$$(\mathbf{normal}\ 0\ 1)(\lambda x.\, f(5 \cdot x)) = \mathbf{lebesgue}(\lambda y.\, \kappa(y) \cdot f(y)) \tag{36}$$

for all $f : \mathbb{R} \to \mathbb{R}_+$. In other words, we solve for κ in

$$\int_{\mathbb{R}} (\mathbf{dnorm}\ 0\ 1)(x) \cdot f(5 \cdot x)\, dx = \int_{\mathbb{R}} \kappa(y) \cdot f(y)\, dy. \tag{37}$$

To match the two sides, we need to change the integration variable from x to $y = 5 \cdot x$. Equivalently, we need to express—or *reparameterize*—the Lebesgue measure over y in terms of $x = y/5$. Using integral calculus, we calculate

$$\int_{\mathbb{R}} (\mathbf{dnorm}\ 0\ 1)(x) \cdot f(5 \cdot x)\, dx = \int_{\mathbb{R}} (\mathbf{dnorm}\ 0\ 1)(y/5) \cdot f(y) \cdot (dx/dy)\, dy, \tag{38}$$

in which $dx/dy = 1/5$. Hence, we find the density $\kappa = \lambda y.\, (\mathbf{dnorm}\ 0\ 1)(y/5) \cdot (1/5)$.

Using the disintegrator to find the same density, the top-level call

$$check\ (\mathbf{do}\ \{x \leftsquigarrow \mathbf{normal}\ 0\ 1;\ \mathbf{return}\ (5 \bullet x, ())\})\ \mathbf{lebesgue}\ y \tag{39}$$

eventually turns into

$$\lhd\ (5 \bullet x)\ \mathbf{lebesgue}\ y\ \overline{(\mathbf{return}\ ())}\ [x \leftsquigarrow \mathbf{normal}\ 0\ 1] \tag{40}$$

and then into

$$\lessdot\ (\mathbf{mul}\ 5)\ x\ \mathbf{lebesgue}\ y\ \overline{(\mathbf{return}\ ())}\ [x \leftsquigarrow \mathbf{normal}\ 0\ 1], \tag{41}$$

which calls *jacobian* (**mul** 5) **lebesgue**. The *jacobian* function differentiates $inv\ (\mathbf{mul}\ 5) = \mathbf{div}\ 5$ and returns $\lambda y.\, 1/5$, hence computing the Jacobian factor necessary to change the integration variable. After $\lessdot$ emits this **factor** 1/5, it proceeds to constrain the value of x to be $y/5$, by calling

$$\lhd\ x\ \mathbf{lebesgue}\ (y/5)\ \overline{(\mathbf{return}\ ())}\ [x \leftsquigarrow \mathbf{normal}\ 0\ 1]. \tag{42}$$

Evaluate $\boxed{\mathbf{do}\ \{h; x \leftsquigarrow \mathbf{return}\ e; M\} \sqsupseteq \triangleright\, e\, (\lambda v.\, \overline{M\{x \mapsto v\}})\, h}$

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

Perform $\boxed{\mathbf{do}\ \{h; x \leftsquigarrow m; M\} \sqsupseteq \gg\!\!\!\triangleright\, m\, (\lambda v.\, \overline{M\{x \mapsto v\}})\, h}$

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

Smart constructors $\boxed{\mathit{fst}\ e = \mathbf{fst}\ e \quad \ldots \quad \mathit{outr}\ v\ k\ h = \mathbf{do}\ \{\mathbf{let\ inr}\ x = v; k\ x\ h\}}$

$\mathit{fst} : \lceil\alpha \times \beta\rceil \to \lceil\alpha\rceil \quad \ldots \quad \mathit{outr} : \lfloor\alpha + \beta\rfloor \to (\lceil\beta\rceil \to \mathit{heap} \to \{\lfloor\mathbb{M}\,\gamma\rfloor\}) \to (\mathit{heap} \to \{\lfloor\mathbb{M}\,\gamma\rfloor\})$

Fig. 12. Operations for partial evaluation. The implementations are same as Shan and Ramsey's and omitted.

4.1.2 Dividing base measures. Although the input to the disintegrator is a distribution over $\alpha \times \beta$, buried inside that input is a distribution over just α. That is the distribution of **fst** v in *check* in Figure 9, akin to a marginal distribution. The disintegrator succeeds if it can find a density for this distribution. The disintegrator gradually reduces its problem to that of finding a density of a given *primitive* measure ξ with respect to a given base measure ν.

The density of a primitive measure with respect to a given base is found by the call $\lll \xi\ \nu$ using Proposition 2.7(1). In our core language, the only primitive measure is **lebesgue**, but a larger language might feature primitive measures such as uniform, Beta, or Gamma distributions. For each primitive measure ξ, the $\lll$ function should choose an intermediate measure μ that is represented in the base-measure language, and emit a density of ξ with respect to μ (unless $\xi = \mu$, as in the case $\lll$ **lebesgue**). It then remains to find a density of the base measure μ with respect to the base measure ν, and that is the job of the auxiliary function $\div$, defined in Figure 11.

Because the only base measure over $\mathbb{R}$ so far is **lebesgue**, the implementation of $\div$ is extremely simple: the constant-1 function is a density of **lebesgue** with respect to **lebesgue**. The return type of $\div$ is not $\mathbb{R}_+$ but its isomorphic type $\mathbb{M}\,\mathbb{1}$, so $\div$ returns not 1 but its isomorphic value **return** ().

4.2 Partial evaluation

The functions $\gg\!\!\!\triangleright$ and $\triangleright$ perform lazy partial evaluation to bring terms into head normal form. They are assisted by *smart constructors* such as *fst* and *outr*, which are meta-functions that are semantically equivalent to **fst** and **let inr** in core Hakaru syntax but reduce away constructor applications if possible; for example, $\mathit{fst}\,(e_1, e_2) = e_1$ and $\mathit{outr}\,(\mathbf{inr}\ e)\ k\ h = k\ e\ h$. These parts of the disintegrator are standard and unchanged from Shan and Ramsey's—they do not even take a new base-measure argument—so we omit their implementations and only show their informal types and semantic specifications in Figure 12.

4.3 Proof technicalities

The semantic specifications (boxed) above, including the soundness of any result returned by the external interface *check*, follow from three inductively proven properties of the 4 workhorse functions $\lll, \lhd, \gg\!\!\!\triangleright, \triangleright$. These properties are specified in Figure 13.

(1) *Commutativity* says that emitting a binding g and then constraining or evaluating an expression e or m is same as constraining or evaluating e or m and then emitting g. In other words, we can use Tonelli's theorem to exchange g with whatever a workhorse functions emits.
(2) *Associativity* says that [constraining or evaluating e or m with the continuation c or k] followed by the final action M is same as constraining or evaluating e or m with the continuation [c or k followed by the final action M].

Commutativity

$$\begin{array}{|l|}\hline \textbf{do}\ \{g;\ \lhd\ e\ b\ v\quad c\ h\} = \lhd\ e\ b\quad v\ (\lambda h'.\,\textbf{do}\ \{g;\ c\quad h'\})\ h \\ \textbf{do}\ \{g;\ \lll\ m\ b\ v\ c\ h\} = \lll\ m\ b\ v\ (\lambda h'.\,\textbf{do}\ \{g;\ c\quad h'\})\ h \\ \textbf{do}\ \{g;\ \rhd\ e\qquad k\ h\} = \rhd\ e\qquad (\lambda v h'.\,\textbf{do}\ \{g;\ k\ v\ h'\})\ h \\ \textbf{do}\ \{g;\ \ggg\ m\qquad k\ h\} = \ggg\ m\qquad (\lambda v h'.\,\textbf{do}\ \{g;\ k\ v\ h'\})\ h \\ \hline \end{array}$$

Associativity

$$\begin{array}{|l|}\hline \textbf{do}\ \{p \leftsquigarrow \lhd\ e\ b\ v\quad c\ h;\ M\} = \lhd\ e\ b\ v\quad (\lambda h'.\,\textbf{do}\ \{p \leftsquigarrow c\quad h';\ M\})\ h \\ \textbf{do}\ \{p \leftsquigarrow \lll\ m\ b\ v\ c\ h;\ M\} = \lll\ m\ b\ v\ (\lambda h'.\,\textbf{do}\ \{p \leftsquigarrow c\quad h';\ M\})\ h \\ \textbf{do}\ \{p \leftsquigarrow \rhd\ e\qquad k\ h;\ M\} = \rhd\ e\qquad (\lambda v h'.\,\textbf{do}\ \{p \leftsquigarrow k\ v\ h';\ M\})\ h \\ \textbf{do}\ \{p \leftsquigarrow \ggg\ m\qquad k\ h;\ M\} = \ggg\ m\qquad (\lambda v h'.\,\textbf{do}\ \{p \leftsquigarrow k\ v\ h';\ M\})\ h \\ \hline \end{array}$$

Parametricity

$$\begin{array}{|l|}\hline (\lhd\ e\quad b\ v\ \overline{M}\ h)\{s \mapsto e'\} = \lhd\ e\quad b\ (v\{s \mapsto e'\})\ \overline{M\{s \mapsto e'\}}\ h \\ (\lll\ m\ b\ v\ \overline{M}\ h)\{s \mapsto e'\} = \lll\ m\ b\ (v\{s \mapsto e'\})\ \overline{M\{s \mapsto e'\}}\ h \\ \hline \end{array}$$

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

Fig. 13. Inductively proven properties of the 4 workhorse functions $\lll, \lhd, \ggg, \rhd$

(3) *Parametricity* says that the backward functions $\lhd$ and $\lll$ treat their arguments v and c parametrically. That is, these functions never inspect those arguments, so they commute with substitution on those arguments.

The definition of the 4 workhorse functions $\lll, \lhd, \ggg, \rhd$ uses recursion and nondeterminism, so assembling all the equational derivations into an overall soundness proof is not entirely trivial. Technically, Figures 9 and 12 define a function F from 4-tuples of functions $(\lll_n, \lhd_n, \ggg_n, \rhd_n)$ to 4-tuples of functions $(\lll_{n+1}, \lhd_{n+1}, \ggg_{n+1}, \rhd_{n+1})$, but the step-indexing subscripts n in the right-hand-sides and $n+1$ in the left-hand-sides are elided. Moreover, as the types show, each of these step-indexed functions returns a set of terms and takes a continuation that also returns a set of terms. These sets of terms are partially ordered by the subset relation, and these functions (including continuations) returning a set of terms are then partially ordered pointwise.

To kick off the recursion, we set $\lll_0, \lhd_0, \ggg_0, \rhd_0$ to constant functions returning the empty set. An easy induction on n then shows that each function $\lll_n, \lhd_n, \ggg_n, \rhd_n$ is monotonic in its continuation argument. Restricted to such monotonic functions, F is itself monotonic, so another easy induction on n shows that the 4-tuple $(\lll_n, \lhd_n, \ggg_n, \rhd_n)$ is monotonic in the step-index n. We can thus define the 4-tuple of functions $(\lll, \lhd, \ggg, \rhd)$ as the least fixed point of F—in other words, as the pointwise union of all the step-indexed 4-tuples. Each property of the 4 functions (commutativity, associativity, parametricity, and then the semantic specifications) is then proven by induction on the step-index n.

5 A RESTRICTED BASE-CHECKING DISINTEGRATOR

Now that we have a disintegrator that takes a base measure as a second input and uses it in just a few semantically specified operations, we are ready to enrich the variety of base measures. Recall from Section 2 that we want to allow base measures such as

$$\begin{array}{rll} \textbf{lebesgue} \oplus \textbf{return}\ 0 \oplus \textbf{return}\ 1 & : \mathbb{M}\,\mathbb{R} & \text{in (10),} \\ \textbf{lebesgue} \otimes\!\!= \lambda x.\,\textbf{return}\ x & : \mathbb{M}\,\mathbb{R}^2 & \text{(a diagonal), or} \\ (\textbf{lebesgue} \otimes \textbf{lebesgue}) \otimes\!\!= \lambda x.\,\big((\textbf{lebesgue} \oplus \textbf{return}\ (\textbf{fst}\ x)) \otimes & & \\ (\textbf{lebesgue} \oplus \textbf{return}\ (\textbf{snd}\ x))\big) & : \mathbb{M}\,(\mathbb{R}^2)^2 & \text{in (21).} \end{array}$$

$$
\begin{array}{lll}
\text{Bases} & b ::= \mathbf{mix}\ l \lbag e, \ldots \rbag \mid \mathbf{return}\ () \mid b \otimes\!= \lambda x.\, b \mid b \oplus b & \dfrac{e : \mathbb{R} \quad \cdots}{\mathbf{mix}\ l \lbag e, \ldots \rbag : \mathbb{B}\,\mathbb{R}} \\
\text{Continuity} & l ::= \mathrm{ff} \mid \mathrm{tt} &
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\lhd : \lceil \alpha \rceil \to \mathbb{B}\,\alpha \to \lfloor \alpha \rfloor \to (\mathit{heap} \to \{\lfloor \mathbb{M}\,\gamma \rfloor\}) \to \mathit{heap} \to \{\lfloor \mathbb{M}\,\gamma \rfloor\}} \\
\lhd\, e & (b_1 \otimes\!= \lambda x.\, b_2)\ v\ c\ h = \lhd\, (\mathit{fst}\ e)\ b_1\ (\mathit{fst}\ v)\ \big(\lhd\, (\mathit{snd}\ e)\ (b_2\{x \mapsto \mathit{fst}\ v\})\ (\mathit{snd}\ v)\ c\big)\ h & \\
\lhd\, u & b \qquad v\ c\ h = \mathbf{do}\ \{() \leftsquigarrow (\mathbf{mix}\ \mathrm{ff}\ \lbag u \rbag \div b)\ v;\ c\ h\} & \text{where } u : \mathbb{R} \text{ is atomic} \\
\lhd\, r & b \qquad v\ c\ h = \mathbf{do}\ \{() \leftsquigarrow (\mathbf{mix}\ \mathrm{ff}\ \lbag r \rbag \div b)\ v;\ c\ h\} & \text{where } r : \mathbb{R} \text{ is a literal} \\
\multicolumn{3}{l}{\lll : \lceil \mathbb{M}\,\alpha \rceil \to \mathbb{B}\,\alpha \to \lfloor \alpha \rfloor \to (\mathit{heap} \to \{\lfloor \mathbb{M}\,\gamma \rfloor\}) \to \mathit{heap} \to \{\lfloor \mathbb{M}\,\gamma \rfloor\}} \\
\lll\, \mathbf{lebesgue}\ b & \qquad v\ c\ h = \mathbf{do}\ \{() \leftsquigarrow (\mathbf{mix}\ \mathrm{tt}\ \lbag \rbag \div b)\ v;\ c\ h\} &
\end{array}
$$

$$
\begin{array}{ll}
\mathit{jacobian} : \mathit{invertible} \to \mathbb{B}\,\mathbb{R} \to \lfloor \mathbb{R} \rfloor \to \lfloor \mathbb{R} \rfloor & \mathit{reparam} : \mathit{invertible} \to \mathbb{B}\,\mathbb{R} \to \mathbb{B}\,\mathbb{R} \\
\mathit{jacobian}\ f\ (\mathbf{mix}\ \mathrm{ff}\ _) = \lambda v.\, 1 & \mathit{reparam}\ f\ (\mathbf{mix}\ l\ \lbag e_1, \ldots \rbag) = \mathbf{mix}\ l\ \lbag \mathit{inv}\ f\ @\ e_1, \ldots \rbag \\
\multicolumn{2}{l}{\mathit{jacobian}\ f\ (\mathbf{mix}\ \mathrm{tt}\ \lbag e_1, \ldots \rbag) = \lambda v.\, \mathbf{if}\ v = e_1 \lor \cdots\ \mathbf{then}\ 1\ \mathbf{else}\ |\mathit{diff}\ (\mathit{inv}\ f)\ v|}
\end{array}
$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{(\div) : \mathbb{B}\,\mathbb{R} \to \mathbb{B}\,\mathbb{R} \to \{\lfloor \mathbb{R} \rfloor \to \mathbb{M}\,\mathbb{1}\}} \\
\mathbf{mix}\ \mathrm{tt}\ _ & \div\ \mathbf{mix}\ \mathrm{ff}\ _ & = \bot & \\
\mathbf{mix}\ \mathrm{tt}\ \lbag \rbag & \div\ \mathbf{mix}\ \mathrm{tt}\ \lbag e_1, \ldots \rbag & = \lambda v.\, \mathbf{do}\ \{\mathbf{observe}\ (v \neq e_1 \land \cdots);\ \mathbf{return}\ ()\} & \\
\mathbf{mix}\ \mathrm{ff}\ \lbag \rbag & \div\ _ & = \mathbf{fail} & \\
\mathbf{mix}\ \mathrm{ff}\ \lbag e \rbag & \div\ \mathbf{mix}\ _\ \lbag e_1, \ldots \rbag & = \lambda v.\, \mathbf{do}\ \{\mathbf{observe}\ (v = e);\ \mathbf{factor}\ 1/\#\lbag i \mid v = e_i \rbag;\ \mathbf{return}\ ()\} & \\
 & & \quad \text{if } e \text{ is known to be equal to at least one of } \lbag e_1, \ldots \rbag & \\
\mathbf{mix}\ \mathrm{ff}\ \lbag e \rbag & \div\ \mathbf{mix}\ _\ \lbag e_1, \ldots \rbag & = \bot \quad \text{otherwise} & \\
\multicolumn{2}{l}{\mathbf{mix}\ l\ \lbag e, e_1, \ldots \rbag \div b} & = \lambda v.\, (\mathbf{mix}\ \mathrm{ff}\ \lbag e \rbag \div b)\ v \oplus (\mathbf{mix}\ l\ \lbag e_1, \ldots \rbag \div b)\ v &
\end{array}
$$

Fig. 14. Changes to handle base measures that are either mixtures of the Lebesgue measure and point masses or dependent products

Accordingly, we generalize the base-measure language in two ways.

(1) Base measures over $\mathbb{R}$ have the form $\mathbf{mix}\ l\ \lbag e_1, \ldots \rbag$. Here $\lbag e_1, \ldots \rbag$ is a bag of real terms whose Dirac measures are mixed together, and l is a meta-level Boolean indicating whether the Lebesgue measure is mixed in as well. In other words, the base measure $\mathbf{mix}\ \mathrm{ff}\ \lbag e_1, \ldots \rbag$ means $\mathbf{return}\ e_1 \oplus \cdots$, and the base measure $\mathbf{mix}\ \mathrm{tt}\ \lbag e_1, \ldots \rbag$ means $\mathbf{lebesgue} \oplus \mathbf{return}\ e_1 \oplus \cdots$. Hence $\mathbf{lebesgue}$ can be expressed as $\mathbf{mix}\ \mathrm{tt}\ \lbag \rbag$, and $\mathbf{return}\ x$ can be expressed as $\mathbf{mix}\ \mathrm{ff}\ \lbag x \rbag$.

(2) Independent products $b_1 \otimes b_2$ become dependent products $b_1 \otimes\!= \lambda x.\, b_2$, in which x can appear in b_2, namely in a bag of real terms.

These changes are summarized at the top of Figure 14. The rest of the figure updates the functions $\lhd$, $\lll$, *jacobian*, *reparam*, and $\div$ to handle the new base measures. Thanks to the groundwork laid in Section 4, the semantic specifications for these functions remain the same, and only cases corresponding to the new base measures need to be added. As in Section 4, these cases are derived and proven from those semantic specifications by equational reasoning.

The new cases of $\lhd$ take advantage of the extended base-measure language. Constraining the second element of a pair ($\mathit{snd}\ e$) now uses a base measure that can depend on what the first element was constrained to be ($\mathit{fst}\ v$). And constraining a Dirac measure (at u or r) now passes the job to $\div$ instead of returning $\bot$ right away. In fact, now the only function that can return $\bot$ is $\div$.

It is instructive to derive the second and fourth cases of $\div$, which compute the density of the Lebesgue measure and of a Dirac measure with respect to their mixture. In particular, exactly the

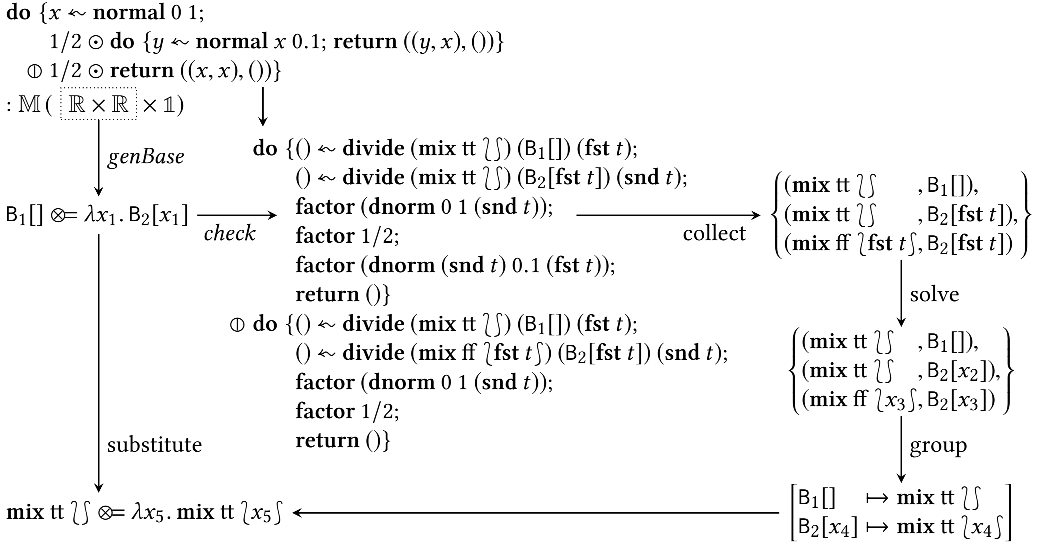

Fig. 15. Inferring a principal base measure (lower left) for an input program (upper left)

same sequence of justifications derive both **mix** tt $\wr\int$ ÷ **mix** tt $\wr 0 \int$ and **mix** ff $\wr 0\int$ ÷ **mix** tt $\wr 0 \int$. We show the former case, which means dividing **lebesgue** by **lebesgue** ⊕ **return** 0:

$$
\begin{aligned}
\textbf{lebesgue} &= \textbf{lebesgue} \oplus \textbf{fail} && \textbf{fail} \text{ is identity of } \oplus \\
&= \textbf{do}\ \{t \leftsquigarrow \textbf{lebesgue};\ \textbf{observe}\ (t \neq 0);\ \textbf{return}\ t\} && \{0\} \text{ is } \textbf{lebesgue}\text{-negligible} \\
&\quad \oplus \textbf{do}\ \{t \leftsquigarrow \textbf{return}\ 0;\ \textbf{observe}\ (t \neq 0);\ \textbf{return}\ t\} && 0 = 0 \\
&= \textbf{do}\ \{t \leftsquigarrow \textbf{lebesgue} \oplus \textbf{return}\ 0;\ \textbf{observe}\ (t \neq 0);\ \textbf{return}\ t\} && \oplus \text{ distributivity}
\end{aligned} \tag{43}
$$

The fourth case of ÷ invokes a *term-equality checker* that is sound but incomplete.

6 A BASE-INFERRING DISINTEGRATOR

A base-inferring disintegrator saves the probabilistic programmer from having to construct complex base measures for every application. For example, in Section 2.3 above we saw that single-site Metropolis-Hastings sampling requires the complex non-stock base measure in (21). Constructing base measures requires careful analysis of the marginal of the input program, and knowledge of the robustness of the disintegrator. As our applications increase in complexity, so do the base measures.

A base-inferring disintegrator also reduces the interface complexity of tools that depend on density and disintegration calculation. For instance, a Metropolis-Hastings transformation [Zinkov and Shan 2017; Ścibior et al. 2018] is better served by a density calculator that infers appropriate base measures rather than one that leaks the base requirement onto the type of any tool that uses it. In the end, this rationale is another way of avoiding constructing base measures by hand.

A final motivation for base inference is that we use it to perform disintegration even when the base is known—if the base is expressed as a core Hakaru measure term rather than in the base language of Figure 14. That *unrestricted* disintegrator is described in Section 7 below.

We implement base inference like constraint-based type inference: by introducing *base variables*, extending base-checking disintegration to gather *constraints* on base variables, and finally *solving* these constraints to produce a base measure. We summarize these steps on an example input program in Figure 15; let us now look at them in more detail.

Bases	$b ::= \mathbf{mix}\; l \wr e, \dots \int \mid \mathbf{return}\;() \mid b \otimes\!\!= \lambda x.\, b \mid b \oplus b \mid B$	
Unknown bases	$B ::= \mathsf{B}[\vec{e}] \mid \mathbf{reparam}\; f\; B$	$\overline{B : \mathbb{B}\,\mathbb{R}}$
Base variables	$\mathsf{B}[]$	
Terms	$e ::= \cdots \mid \mathbf{jacobian}\; f\; B\; e \mid \mathbf{divide}\; \underline{b}\; \mathsf{B}[\vec{e}]\; e$	

$$\begin{aligned}
&\mathit{genBase}(\alpha) : \mathit{variables} \to \mathbb{B}\,\alpha \\
&\mathit{genBase}(\mathbb{R}) \quad \vec{x} = \mathsf{B}[\vec{x}] && \text{where } \mathsf{B} \text{ is fresh} \\
&\mathit{genBase}(\mathbb{1}) \quad \vec{x} = \mathbf{return}\;() \\
&\mathit{genBase}(\alpha \times \beta)\; \vec{x} = \mathit{genBase}(\alpha)\; \vec{x} \otimes\!\!= \lambda y.\, \mathit{genBase}(\beta)\; (\vec{x}, y) && \text{where } y : \alpha \text{ is fresh} \\
&\mathit{genBase}(\alpha + \beta)\; \vec{x} = \mathit{genBase}(\alpha)\; \vec{x} \oplus \mathit{genBase}(\beta)\; \vec{x}
\end{aligned}$$

$$\begin{aligned}
&\mathit{jacobian} : \mathit{invertible} \to \mathbb{B}\,\mathbb{R} \to \lfloor\mathbb{R}\rfloor \to \lfloor\mathbb{R}\rfloor && \mathit{reparam} : \mathit{invertible} \to \mathbb{B}\,\mathbb{R} \to \mathbb{B}\,\mathbb{R} \\
&\mathit{jacobian}\; f\; B\; e = \mathbf{jacobian}\; f\; B\; e && \mathit{reparam}\; f\; B = \mathbf{reparam}\; f\; B
\end{aligned}$$

$$\begin{aligned}
&(\div) : \mathbb{B}\,\mathbb{R} \to \mathbb{B}\,\mathbb{R} \to \{\lfloor\mathbb{R}\rfloor \to \mathbb{M}\,\mathbb{1}\} \\
&\underline{b} \div \mathbf{reparam}\; f\; B = \lambda v.\, \big(\mathit{jacobian}\; f\; B\; (f \mathbin{@} v) \cdot \mathit{jacobian}\; (\mathit{inv}\; f)\; \underline{b}\; v\big)^{-1} \\
&\qquad\qquad \odot \big(\mathit{reparam}\; (\mathit{inv}\; f)\; \underline{b} \div B\big)\; (f \mathbin{@} v) \\
&\underline{b} \div \mathsf{B}[\vec{e}] \qquad = \lambda v.\, \mathbf{divide}\; \underline{b}\; \mathsf{B}[\vec{e}]\; v
\end{aligned}$$

Fig. 16. Introducing base-measure variables and updating the base-checking disintegrator

6.1 Introducing base variables

We start by adding variables $\mathsf{B}[]$ to our language of bases in Figure 14. The B part of the notation expresses that a base variable names a stand-in that must be solved to produce a base measure, while the $[]$ part expresses that the solution is a base measure *with holes*. A base with holes is like a function from terms to bases but cannot, say, do case distinction on a term that plugs a hole.

Base variables represent bases with holes because they might occur under a binder λx as the second argument to $\otimes\!\!=$. A hole in a base is where the bound variable x is used. Our first instinct is to track what variables x are in scope by augmenting base variables $\mathsf{B}[]$ with a sequence of core Hakaru variables $\vec{x}$. Upon closer inspection of the pair case $\lhd\; e\; (b_1 \otimes\!\!= \lambda x.\, b_2)\; v$ in Figure 14, we see the need for a sequence of core Hakaru terms $\vec{e}$, and not just variables $\vec{x}$. In this case, $\lhd$ deconstructs the $\otimes\!\!=$ base measure and substitutes $\mathit{fst}\; v$ for the bound variable x in b_2. Hence, a *plugged* base variable must store core Hakaru terms that may be in scope after such a substitution.

We thus add a new construct, $\mathsf{B}[\vec{e}]$, which represents a base measure with holes that are plugged by terms $\vec{e}$ in scope. This construct stores a name B along with a sequence of core Hakaru terms $\vec{e}$. We refer to such expressions as *plugged base variables*. This construct becomes a part of a new syntactic category of *unknown bases*, notated B. The new base language, which combines unknown bases with *ground bases* (defined below), is shown at the top of Figure 16.

Definition 6.1. A ground base $\underline{b}$ is a base measure with no base variables. The type of a ground base is notated $\underline{\mathbb{B}}\,\alpha$.

We only need variables of type $\mathbb{B}\,\mathbb{R}$, because ground base measures of the same type differ only in their subterms of type $\mathbb{B}\,\mathbb{R}$. Base measures have a unique structure for each of the other non-measure type constructors: $\mathbf{return}\;()$ is the only base over $\mathbb{1}$, bases over pairs are always of the form $b_1 \otimes\!\!= \lambda x.\, b_2$, and bases over disjoint sums are always of the form $b_1 \oplus b_2$.

6.2 Producing constraints by base-checking

Having added base variables in our language, we focus on producing a set of *constraints* on them. A constraint is represented as a pair of base measures. Its meaning is that calling $\div$ on the two bases must succeed rather than returning $\bot$. By design, the first base is ground while the second is a plugged base variable; i.e., constraints are of the form $(\underline{b}, \mathsf{B}[\vec{e}])$. Given an input program, we obtain a set of constraints by

(1) automatically constructing a base measure b^* that may contain base variables, and
(2) running base-checking disintegration with respect to b^*.

First, we need to construct a base b^* that contains variables for sub-expressions of type $\mathbb{B}\,\mathbb{R}$. As done above in Section 3.2, we construct the base using *genBase*(α). This time we use a modified definition of *genBase*(α)—shown in Figure 16—to construct a base measure of type $\mathbb{B}\,\alpha$ that includes plugged base variables in any leaf positions of type $\mathbb{B}\,\mathbb{R}$. The function now takes a set of core Hakaru variables as input. This set is initially empty, extended with a fresh core Hakaru variable whenever α is a pair type, and stored alongside a fresh base variable whenever α is $\mathbb{R}$. Thus, we set $b^* = genBase(\alpha)\,()$. For the example in Figure 15, $b^* = \mathsf{B}_1[] \otimes\!\!= \lambda x_1.\, \mathsf{B}_2[x_1]$.

Second, we need to run base-checking disintegration with respect to b^*. In order for this to work we need to update *jacobian*, *reparam*, and $\div$ to handle unknown bases.

Two new syntax extensions help *jacobian* and *reparam* handle unknown bases. We add a **jacobian** core Hakaru construct of type $\mathbb{R}$ that is produced by *jacobian*, and a **reparam** base-measure construct of type $\mathbb{B}\,\mathbb{R}$ that is produced by *reparam*, whenever each function encounters an unknown base and needs to suspend (or residualize) itself. Bases composed using **reparam** belong themselves to the category of unknown bases. These extensions are shown near the top of Figure 16.

A third syntax extension helps $\div$ handle unknown bases. We introduce a new core Hakaru construct called **divide**, representing a suspended (or residualized) call to $\div$. Now, the way the disintegrator calls $\div$ ensures that the *dividend* (the first argument) of $\div$ is always a ground base. Only the *divisor* (the second argument) may contain unknown bases as sub-expressions, and since it is of type $\mathbb{B}\,\mathbb{R}$, there are two possibilities.

- The divisor is an unknown base of the form **reparam** $f\,B$. In this case we use *jacobian* to "undo the reparameterization" of the unknown base B, and reparameterize the (ground) dividend by the inverse of f. We implement this near the bottom of Figure 16. The call to *reparam* here is guaranteed to produce a ground base.
- The divisor is a plugged base variable, i.e., an unknown base of the form $\mathsf{B}[\vec{e}]$. In this case we suspend (or residualize) the call using **divide**.

Thus, running base-checking disintegration with respect to a *non-ground* base produces a program with embedded residual $\div$ calls of the form **divide** $\underline{b}\,\mathsf{B}[\vec{e}]\,e$, where $\underline{b}$ is a ground base, and e is a core Hakaru term of type $\mathbb{R}$. We walk through this program and collect the first two arguments of each **divide** expression into a set of constraints $(\underline{b}, \mathsf{B}[\vec{e}])$. In Figure 15 this set is the output of a *collect* transformation.

6.3 Solving constraints to produce the base measure

In the final step of base inference we aim to produce a ground base $\underline{b}^*$ that shares the structure of the base measure b^* given initially by *genBase*, but replaces all *plugged base variables* with *plugged ground bases*. Our inferred base must permit base-checking disintegration on the original input program. There may be infinitely many ground instantiations of base variables that satisfy this requirement, and we aim to infer a *principal base measure*. Finally, our inferred base must contain no open core Hakaru terms.

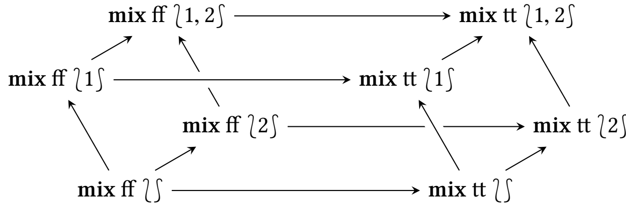

Fig. 17. The preorder <: ("is divisible by"), where an edge from b_1 to b_2 means $b_1 <: b_2$

From the previous step we obtain a set of constraints of the form $(\underline{b}, \mathsf{B}[\vec{e}])$. This set may contain multiple constraints for the same base variable $\mathsf{B}[]$ (plugged with possibly different term sequences $\vec{e}$). From such a set we obtain a principal base in three steps.

(1) *Solve* each individual constraint in the set. This is the only step that can fail.
(2) *Group* (or *unify*) constraints by their base variable.
(3) *Substitute* into b^* the (unified) solutions for each base variable.

6.3.1 Solving a base variable constraint. A particular constraint $(\underline{b}_1, \mathsf{B}[\vec{e}])$ is deemed *solvable* if we can produce a ground base replacement $\underline{b}_2[]$ for the base variable $\mathsf{B}[]$ such that $\underline{b}_1 \div \underline{b}_2[\vec{e}] \neq \bot$. Now, there may be infinitely many $\underline{b}_2[]$ that satisfy this property, which forms a preorder when defined as a syntactic relation.

Definition 6.2. Let $\underline{b}_1$ and $\underline{b}_2$ be ground bases over $\mathbb{R}$. If $\underline{b}_1 \div \underline{b}_2 \neq \bot$, then we say that $\underline{b}_1 <: \underline{b}_2$, pronounced "$\underline{b}_1$ is divisible by $\underline{b}_2$".

Proposition 6.3. *The relation <: is a preorder.*

We illustrate a small slice of this preorder as a graph in Figure 17, where nodes represent base measures, and an edge from b_1 to b_2 means that $b_1 <: b_2$.

The transitivity of <: can bring about infinitely many solutions for a constraint $(\underline{b}_1, \mathsf{B}[\vec{e}])$. We aim for a *principal solution* $\underline{b}_1^*[]$ such that

$$\forall \underline{b}_2[].\ \underline{b}_1 <: \underline{b}_2[\vec{e}] \Leftrightarrow \underline{b}_1^*[\vec{x}] <: \underline{b}_2[\vec{x}] \tag{44}$$

so in particular $\underline{b}_1 <: \underline{b}_1^*[\vec{e}]$. Here $\vec{x}$ are as many fresh variables as there are terms in $\vec{e}$. If $\underline{b}_1$ contained no open core Hakaru terms, then we could just let $\underline{b}_1^*[] = \underline{b}_1$. This is *almost* what we do.

What we *actually* do is solve the problem $\underline{b}_1^*[\vec{e}] = \underline{b}_1$, a special case of second-order matching [Huet and Lang 1978; Huet 1976], by checking that all core Hakaru variables in $\underline{b}_1$ occur in the context of some subterm of $\underline{b}_1$ that is equal to some term in $\vec{e}$. To check this, we replace every subterm of $\underline{b}_1$ that is known to be equal to some term in $\vec{e}$ by a hole. We notate the result of this replacement by $\underline{b}_1\{[] \hookleftarrow e\}$. If the result contains any free variable, we conclude that there is no solution and overall there is no base that permits disintegration of the original input program. Otherwise, the result is the principal solution; we set $\underline{b}_1^*[] = \underline{b}_1\{[] \hookleftarrow e\}$.

6.3.2 Grouping constraints by base variables. We are home free once we have successfully solved each constraint. The next step is to group the solved constraints $(\underline{b}_1^*[\vec{x}], \mathsf{B}[\vec{x}])$ by their base variables $\mathsf{B}[]$. For this we define a binary operation *bplus* that acts as a join in the preorder <: by summing together ground base measures of type $\mathbb{B}\,\mathbb{R}$. Since **mix** is the only way to construct such bases, *bplus* has a one-line definition:

$$\begin{aligned} &bplus : \underline{\mathbb{B}}\,\mathbb{R} \to \underline{\mathbb{B}}\,\mathbb{R} \to \underline{\mathbb{B}}\,\mathbb{R} \\ &bplus\ (\mathbf{mix}\ \ l\ \lbag e_1, \ldots \rbag)\ (\mathbf{mix}\ \ l'\ \lbag e_1', \ldots \rbag) = \mathbf{mix}\ \ (l \vee l')\ \lbag e_1, \ldots, e_1', \ldots \rbag \end{aligned} \tag{45}$$

Any two solved constraints $(\underline{b}_1^*[\vec{x}], \mathsf{B}[\vec{x}])$ and $(\underline{b}_2^*[\vec{y}], \mathsf{B}[\vec{y}])$ that share the same base variable $\mathsf{B}[]$ now get grouped into a solved constraint $(\mathit{bplus}\ \underline{b}_1^*[\vec{z}]\ \underline{b}_2^*[\vec{z}], \mathsf{B}[\vec{z}])$, where the variables $\vec{z}$ are fresh. After grouping, we have *one solved constraint per base variable*. (The odd unconstrained base variable $\mathsf{B}[]$ gets the least solution $(\mathbf{mix}\ \mathsf{ff}\ \wr\int, \mathsf{B}[\vec{z}])$.)

6.3.3 Substituting solutions to form a principal base. At this point we have one ground instantiation $\underline{b}_1^*$ for each base variable B_1. We obtain our principal base measure $\underline{b}^*$ by substituting these ground bases into b^*:

$$\underline{b}^* = b^*\{\mathsf{B}_1[] \mapsto \underline{b}_1^*[], \dots\} \tag{46}$$

In Figure 15 we produce the base $\mathbf{mix}\ \mathsf{tt}\ \wr\int \otimes\!\!= \lambda x_5.\ \mathbf{mix}\ \mathsf{tt}\ \wr x_5 \int$.

7 AN UNRESTRICTED BASE-CHECKING DISINTEGRATOR

When the user of disintegration has a base measure in mind, it is easier to specify it as a core Hakaru measure term rather than in the relatively spartan base language of Figure 14. Indeed, most applications of disintegration described in Section 2 specify such a base measure. Just to recall one example from Section 2.3, Metropolis-Hastings sampling requires the density of $\zeta =\!\!\otimes \xi$ with respect to $\xi \otimes\!\!= \zeta$ [Tierney 1998], where ξ and ζ are specified as probabilistic programs.

To disintegrate one core Hakaru term with respect to another, we use Proposition 2.7 and define

$$\mathit{disint} : \lceil \mathbb{M}\,(\alpha \times \beta) \rceil \to \lceil \mathbb{M}\,\alpha \rceil \to \lfloor \alpha \rfloor \to \{\lfloor \mathbb{M}\,\beta \rfloor\} \qquad \boxed{m_1 \sqsupseteq m_2 \otimes\!\!= \mathit{disint}\ m_1\ m_2} \tag{47}$$

$$\mathit{disint}\ m_1\ m_2\ t = |\mathit{check}\ m_2\ b\ t|^{-1} \odot \mathit{check}\ m_1\ b\ t \qquad \text{where } b = \mathit{infer}\ m_2$$

That is, we use base inference to find the intermediate base measure μ in Proposition 2.7(1). One caveat of this approach is that it assumes the density $|\mathit{check}\ m_2\ b\ t|$ (whose reciprocal is taken above) is almost never 0 or ∞.

8 EVALUATION

Our new disintegrator successfully returns (proven-correct) results in all the applications claimed in Section 2:

(1) a clamped normal distribution with respect to a **mix** tt base (Example 2.5),
(2) clamped normal distributions with respect to each other (Example 2.8),
(3) mutual information (Example 2.9) in a joint distribution that is a discrete-continuous mixture,
(4) importance sampling of a posterior distribution with respect to a prior distribution that is a discrete-continuous mixture (Example 2.11),
(5) Metropolis-Hastings sampling (Example 2.14) using single-site proposals and using reversible-jump proposals,
(6) belief update using a clamped observation (Example 2.17),
(7) Gibbs sampling (Example 2.18) of a joint distribution that is a discrete-continuous mixture.

However, because the notion of term equality and matching we implemented in Figure 14 and Section 6.3.1 ("known to be equal") is mere α-equivalence and is not modulo β-equivalence for sum types, our disintegrator fails (by returning $\bot$) on a Metropolis-Hastings reversible-jump proposal that sometimes jumps without adding noise (for example, from **inr** (x, y) to **inl** $((x + y)/2)$ exactly). We leave it to future work to extend the notion of term equality and matching and handle this case.

Another direction for future work is to handle array programs without unrolling them [Narayanan and Shan 2017]. In other words, we would like to add plates (n-ary products, where n is a symbolic array size) to our base-measure language. Such plates of mixture bases may enable a disintegrator to scale up and to produce full conditional distributions for Gibbs sampling from an array distribution.

ACKNOWLEDGMENTS

This research was supported by DARPA contract FA8750-14-2-0007.

REFERENCES

Nathanael Leedom 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*. IEEE Computer Society Press, 107–116.

Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. 2016. On Computability and Disintegration. *Mathematical Structures in Computer Science* (2016), 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.

Sooraj Bhat, Ashish Agarwal, Richard Vuduc, and Alexander Gray. 2012. A Type Theory for Probability Density Functions. In *POPL '12: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 545–556.

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*, Nir Piterman and Scott A. Smolka (Eds.). Springer, 508–522.

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

Thomas M. Cover and Joy A. Thomas. 2006. *Elements of Information Theory* (second ed.). Wiley.

Jean Dieudonné. 1947–1948. Sur le Théorème de Lebesgue-Nikodym (III). *Annales de l'université de Grenoble* 23 (1947–1948), 25–53. http://eudml.org/doc/84619

Weihao Gao, Sreeram Kannan, Sewoong Oh, and Pramod Viswanath. 2017. Estimating Mutual Information for Discrete-Continuous Mixtures. In *Advances in Neural Information Processing Systems*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5986–5997.

Alan E. Gelfand, Adrian F. M. Smith, and Tai-Ming Lee. 1992. Bayesian Analysis of Constrained Parameter and Truncated Data Problems Using Gibbs Sampling. *J. Amer. Statist. Assoc.* 87, 418 (1992), 523–532.

Stuart Geman and Donald Geman. 1984. Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6, 6 (1984), 721–741.

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*, Bernhard Banaschewski (Ed.). Springer, 68–85.

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*, David Allen McAllester and Petri Myllymäki (Eds.). AUAI Press, 220–229.

N. J. Gordon, D. J. Salmond, and A. F. M. Smith. 1993. Novel Approach to Nonlinear/Non-Gaussian Bayesian State Estimation. *IEE Proceedings F (Radar and Signal Processing)* 140, 2 (April 1993), 107–113.

Peter J. Green. 1995. Reversible Jump Markov Chain Monte Carlo Computation and Bayesian Model Determination. *Biometrika* 82, 4 (1995), 711–732.

W. Keith Hastings. 1970. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika* 57, 1 (1970), 97–109.

Gérard Huet. 1976. *Résolution d'Équations dans des Langages d'Ordre* $1, 2, \ldots, \omega$. Thèse de doctorat es sciences mathématiques. Université Paris VII.

Gérard Huet and Bernard Lang. 1978. Proving and Applying Program Transformations Expressed with Second-Order Patterns. *Acta Informatica* 11, 1 (March 1978), 31–55.

David J. C. MacKay. 1998. Introduction to Monte Carlo Methods. In *Learning and Inference in Graphical Models*, Michael I. Jordan (Ed.). Kluwer. Paperback: *Learning in Graphical Models*, MIT Press.

Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. 1953. Equation of State Calculations by Fast Computing Machines. *Journal of Chemical Physics* 21, 6 (1953), 1087–1092.

Wazim Mohammed Ismail and Chung-chieh Shan. 2016. Deriving a Probability Density Calculator (Functional Pearl). In *ICFP '16: Proceedings of the ACM International Conference on Functional Programming*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM Press, 47–59.

Praveen Narayanan and Chung-chieh Shan. 2017. Symbolic Conditioning of Arrays in Probabilistic Programs. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 11:1–11:25.

Luke Ong and Matthijs Vákár. 2018. S-finite Kernels and Game Semantics for Probabilistic Programming. Workshop on probabilistic programming semantics. https://pps2018.sice.indiana.edu/2018/01/06/s-finite-kernels-and-game-semantics-for-probabilistic-programming/

Norman Ramsey and Avi Pfeffer. 2002. Stochastic Lambda Calculus and Monads of Probability Distributions. In *POPL '02: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 154–165.

Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. 2018. Denotational Validation of Higher-Order Bayesian Inference. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018), 60:1–60:29.

Chung-chieh Shan and Norman Ramsey. 2017. Exact Bayesian Inference by Symbolic Disintegration. In *POPL '17: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 130–144.

Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *Programming Languages and Systems: Proceedings of ESOP 2017, 26th European Symposium on Programming (Lecture Notes in Computer Science)*, Yang Hongseok (Ed.). Springer, 855–879.

Luke Tierney. 1998. A Note on Metropolis-Hastings Kernels for General State Spaces. *The Annals of Applied Probability* 8, 1 (Feb. 1998), 1–9.

James Tobin. 1958. Estimation of Relationships for Limited Dependent Variables. *Econometrica* 26, 1 (1958), 24–36.

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 (JMLR Workshop and Conference Proceedings)*, Geoffrey Gordon, David Dunson, and Miroslav Dudík (Eds.). 770–778.

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of AISTATS 2014: 17th International Conference on Artificial Intelligence and Statistics (JMLR Workshop and Conference Proceedings)*. 1024–1032.

Yi Wu, Siddharth Srivastava, Nicholas Hay, Simon Du, and Stuart Russell. 2018. Discrete-Continuous Mixtures in Probabilistic Programming: Generalized Semantics and Inference Algorithms. In *Proceedings of ICML 2018: 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. 5339–5348.

Robert Zinkov and Chung-chieh Shan. 2017. Composing Inference Algorithms as Program Transformations. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*. Corvallis, Oregon.