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.

CCS Concepts: • Mathematics of computing → Probability and statistics; • Theory of computation → Denotational semantics; Program specifications; • Computing methodologies → Symbolic calculus algorithms.

Additional Key Words and Phrases: probabilistic programs, density functions, conditional distributions, measure kernels

ACM Reference Format:

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 automation 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 [Lunn et al. 2000; Goodman et al. 2008; Pfeffer 2009; Wingate et al. 2011; Wood et al. 2014; Carpenter et al. 2017; Wu et al. 2018], 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. Besides disintegration and its special cases, other operations on distributions have also received exact, symbolic automation, in particular simplifying the representation of a distribution while preserving its meaning [Carette and Shan 2016; Gehr et al. 2016].

This paper presents automatic program transformations that perform disintegration. The transformations are exact (under the pretense that computations on real numbers are exact) and symbolic.
(even for programs that contain free variables whose values are unknown). Although exact, they can be used to automate the tedious and error-prone process of not only formulating inference problems but also implementing approximate solutions, just as automatic differentiation is useful not only for formulating tangent-line problems but also for implementing gradient-based optimization methods.

Whereas 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.

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 $\alpha$ is a function that maps an $\alpha$-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 $\alpha$, or a generalized function, is defined as a function that either maps an $\alpha$-set to its size (a number), or equivalently, maps an $\alpha$-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 $\alpha$ 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, \(\alpha\) and \(\mathbb{M}\) are \textit{measurable spaces}, an element of \(\mathbb{R}_+\) is either a non-negative real number or \(\infty\), and a measure over \(\alpha\) is a certain kind of function that maps each \textit{measurable} \(\alpha\)-set or \textit{measurable} \(\alpha\)-to-\(\mathbb{R}_+\) function to an element of \(\mathbb{R}_+\).)

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

\textbf{Example 2.1.} The normal distribution is written \texttt{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

\[
\text{dnorm} 0 1 = \lambda x. \exp(-x^2/2) / \sqrt{2\pi} : \mathbb{R} \rightarrow \mathbb{R}_+ ,
\]

the normal distribution is the measure

\[
\text{normal} 0 1 = \lambda f. \int_\mathbb{R} (\text{dnorm} 0 1)(x) \cdot f(x) \, dx : \mathbb{M} \mathbb{R}.
\]

This integrator maps any (measurable) function \(f : \mathbb{R} \rightarrow \mathbb{R}_+\) 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 \((\text{normal} 0 1)(f) = 1/2\), because the probability is 1/2 that a real drawn randomly from the normal distribution is positive.

Another measure is the Lebesgue measure, written \texttt{lebesgue} in core Hakaru:

\[
\text{lebesgue} = \lambda f. \int_\mathbb{R} f(x) \, dx : \mathbb{M} \mathbb{R}.
\]

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

\textbf{Definition 2.2.} The total \(|\mu|\) of a measure \(\mu\) is its integral of the constant-1 function. That is, \(|\mu| = \mu(\mathbb{R}_+ \cdot 1)\). We say \(\mu\) is \textit{finite} or \textit{infinite} if \(|\mu|\) is. For example, \(|\text{normal} 0 1| = \int_\mathbb{R} (\text{dnorm} 0 1)(x) \, dx = 1\).

Such a finite measure, whose total is 1, is called a \textit{probability distribution} and \textit{normalized}. In contrast, \(|\text{lebesgue}| = \int_\mathbb{R} 1 \, dx = \infty\), so \texttt{lebesgue} is infinite and thus \textit{unnormalized}.

The equation below relates the bell curve \texttt{dnorm 0 1} to the measures \texttt{normal 0 1} and \texttt{lebesgue}:

\[
\text{normal} 0 1 = \lambda f. \text{lebesgue}(\lambda x. (\text{dnorm} 0 1)(x) \cdot f(x)).
\]

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

\[
\text{normal} 0 1 = \textbf{do} \{ x \leftarrow \text{lebesgue}; (\text{dnorm} 0 1)(x) \odot \text{return} x \}.
\]

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

What we have just seen is that \texttt{dnorm 0 1} is a density of \texttt{normal 0 1} with respect to the base measure \texttt{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 $\kappa$ is a density (or Radon-Nikodym derivative) of $\xi$ with respect to the base measure $\mu$ if

$$
\xi = \lambda f \cdot \mu(\lambda x. \kappa(x) \cdot f(x)),
$$

or in core Hakaru,

$$
\xi = \text{do } \{ x \sim \mu; \kappa(x) \odot \text{return } x \}.
$$

We write $\xi = \kappa =\odot \mu$ for short.

A measure over $\mathbb{R}$ is called continuous if it has a density with respect to the Lebesgue measure. More generally, for many spaces $\alpha$, convention stipulates a unique, default stock measure over $\alpha$—for example, the stock measure over $\mathbb{R}$ is the Lebesgue measure—and a measure over $\alpha$ is called continuous if it has a density with respect to the stock measure over $\alpha$. (The existence of a density is related to the notion of absolute continuity by the Radon-Nikodym theorem.) 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 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 $\lambda f \cdot f(42) = \lambda f \cdot \int_{\mathbb{R}} \kappa(x) \cdot f(x) \, dx$. 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 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

$$
\xi = \lambda f \cdot \int_0^1 (\text{dnorm } 0 1)(x) \cdot f(x) \, dx \cdot f(0)
+ \int_0^1 (\text{dnorm } 0 1)(x) \cdot f(x) \, dx
+ \int_1^{+\infty} (\text{dnorm } 0 1)(x) \cdot f(1),
$$

and core Hakaru can express it as

$$
\xi = \text{do } \{ x \sim \text{normal } 0 1; \text{return } \max\{0, \min\{1, x\}\} \}
\odot \text{return } 0
\odot \text{do } \{ x \sim \text{normal } 0 1; \text{if } 0 \leq x \leq 1 \text{ then return } x \text{ else fail} \}
\odot \int_1^{+\infty} (\text{dnorm } 0 1)(x) \, dx \odot \text{return } 1.
$$

In equation (9), the binary operator $\odot$ 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 $\leq$ or $<$ is used.

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

$$
\mu = \text{lebesgue} \odot \text{return } 0 \odot \text{return } 1 = \lambda f \cdot f(0) + \int_{\mathbb{R}} f(x) \, dx + f(1),
$$

Praveen Narayanan and Chung-chieh Shan

Symbolic disintegration with a variety of base measures

Fig. 2. Densities of two normal distributions, \texttt{normal 0 1} and \texttt{normal 3 2}, with respect to the Lebesgue measure

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

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

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

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 \( \xi \) in (9) into \( \kappa \) in (11) with respect to \( \mu \) in (10).

2.2 Using density for inference

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

\textit{Example 2.6.} Suppose we observe a real \( x \) drawn randomly from a black box, and we wonder whether the black box is \texttt{normal 0 1} or \texttt{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 \texttt{normal 0 1} because \( (\text{dnorm} \ 0 \ 1)(1) > (\text{dnorm} \ 3 \ 2)(1) \), whereas if we observe the real 2, then we should favor the hypothesis \texttt{normal 3 2}, because \( (\text{dnorm} \ 0 \ 1)(2) < (\text{dnorm} \ 3 \ 2)(2) \). These densities can be found symbolically using Bhat et al.’s density calculation procedure [2012, 2013].
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 \((2 \odot \text{lebesgue})\) 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 \texttt{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 \texttt{dnorm 0 1} and \texttt{dnorm 3 2} is itself a density—of \texttt{normal 0 1} with respect to \texttt{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 \(\kappa\) is a density of \(\xi\) with respect to \(\mu\), and \(\lambda\) is a density of \(\mu\) with respect to \(\nu\), then the pointwise multiplication \(\lambda x. (\kappa(x) \cdot \lambda(x))\) is a density of \(\xi\) with respect to \(\nu\). (Thus, the existence of a density is a preorder among measures.)

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

**Proof.** (1) We reason equationally:

\[
\begin{align*}
\xi &= \text{do } \{ x \sim \mu; \, \kappa(x) \odot \text{return } x \} \\
&= \text{do } \{ x \sim \nu; \, \lambda(x) \odot \text{return } x \} ; \, \kappa(x) \odot \text{return } x \\
&= \text{do } \{ x \sim \nu; \, \lambda(x) \odot (\kappa(x) \odot \text{return } x) \}
\end{align*}
\]

\(\kappa\) is a density of \(\xi\) with respect to \(\mu\), \(\lambda\) is a density of \(\mu\) with respect to \(\nu\), and \(\lambda \odot (k \odot m) = (l \odot k) \odot m\), by monad laws.

(2) We reason equationally:

\[
\begin{align*}
\text{do } \{ x \sim \xi; \, (1/\kappa(x)) \odot \text{return } x \} \\
= \text{do } \{ x \sim \mu; \, \kappa(x) \odot \text{return } x \} ; \, (1/\kappa(x)) \odot \text{return } x \\
= \text{do } \{ x \sim \mu; \, \kappa(x) \odot ((1/\kappa(x)) \odot \text{return } x) \}
\end{align*}
\]

\(\kappa\) is a density of \(\xi\) with respect to \(\mu\), \(\kappa\) is \(\mu\)-almost everywhere finite and nonzero, and \(m = 1 \odot m\), by monad laws.

**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 \texttt{normal 0 1} or of \texttt{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 \(\mu\) in (10). Our disintegrator is the first transformation to automate this reasoning, because these clamped distributions are not continuous. Similarly, Wu et al.’s GPA problem [2018] is to adjudicate whether a certain black box is the clamping of a normal distribution to the interval \([0, 4]\) or the clamping of a normal distribution to the interval \([0, 10]\), and our disintegrator is the first to automate this reasoning exactly.

**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 logarithm of a certain density [Cover and Thomas 2006]. The need to estimate mutual information for 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 \( \xi : \mathbb{M} \to \mathbb{R} \) by [MacKay 1998]

\[
\xi = (\lambda x. \exp(0.4(x - 0.4)^2 - 0.08x^4)) \circ \text{lebesgue},
\]

but it is not obvious how to sample from it.

In these situations, it can help to find a density \( \kappa \) of the target distribution \( \xi \) with respect to some similar proposal distribution \( \mu \) whose sampling method is known. Instead of sampling from \( \xi \), we can draw samples from \( \mu \) 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 \( \mu \) to approximate \( \xi \) correctly, we must use their weights \( \kappa(x) \) to compensate for the difference between \( \mu \) and \( \xi \). For example, instead of estimating the mean of \( \xi \) by averaging samples from \( \xi \) (which may be difficult to draw), we can average samples from \( \mu \) weighted by \( \kappa \). And instead of plotting a histogram of samples from \( \xi \), we can plot a histogram of samples from \( \mu \), 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 \( \xi \) in (12). For this target \( \xi \), we can use \texttt{normal 3 2} as the proposal, because it is well known how to sample from a normal distribution. To find the density of \( \xi \) with respect to \texttt{normal 3 2}, we can divide their densities with respect to \texttt{lebesgue}, using Proposition 2.7. The form of (12) manifests a density of \( \xi \) with respect to \texttt{lebesgue}, plotted in Figure 4. And a density of \texttt{normal32} is already plotted in Figure 2. So to estimate the expectation of \( f \) with respect to \( \xi \), we can sample \( x \) from \texttt{normal32} and average \( f(x) \) weighted by \( \exp(0.4(x - 0.4)^2 - 0.08x^4)/(dnorm \ 3 \ 2)(x) \).

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, \texttt{lexicographic} likelihood weighting and \texttt{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.
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:

\[
\mu \otimes \nu = \lambda f. \left( \mu (\lambda x. f(x, y)) \right) = \left\{ \begin{array}{l}
\text{do } x \sim \mu; y \sim \nu; \text{return } (x, y) \\
\end{array} \right. 
\]

(13)

Intuitively, to draw from \( \mu \otimes \nu \) is to draw from \( \mu \) and from \( \nu \) 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 (13) and (14) are defined and equal, Staton [2017] established the invariant that all measure expressions denote s-finite kernels (Section 3.3).

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

\[
\mu \bowtie= \nu = \lambda f. \left( \mu (\lambda x. \nu (f(x, y))) \right) = \left\{ \begin{array}{l}
\text{do } x \sim \mu; y \sim \nu(x); \text{return } (x, y) \\
\end{array} \right. 
\]

(15)

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

\[
\mu \bowtie \nu = \lambda f. \left( \nu (\lambda y. \mu (f(x, y))) \right) = \left\{ \begin{array}{l}
\text{do } y \sim \nu; x \sim \mu(y); \text{return } (x, y) \\
\end{array} \right. 
\]

(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 \( \kappa : \alpha \rightarrow \mathbb{R}_+ \) is a density of \( \xi : \mathbb{M} \alpha \) with respect to \( \mu : \mathbb{M} \alpha \), and \( \lambda : \beta \rightarrow \mathbb{R}_+ \) 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 \( \kappa (x) \cdot \lambda (y) : (\alpha \times \beta) \rightarrow \mathbb{R}_+ \) with respect to \( \mu \otimes \nu : \mathbb{M} (\alpha \times \beta) \).

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

Proof. We show only the proof of the "more generally" part. It is again equational reasoning:

\[
\xi \bowtie \zeta = \left\{ \begin{array}{l}
\text{do } x \sim \xi; y \sim \zeta(x); \text{return } (x, y) \\
\end{array} \right. 
\]

definition of \( \bowtie \)

\[
= \left\{ \begin{array}{l}
\text{do } x \sim \xi; \\
y \sim \text{do } y \sim \nu(x); \lambda(x(y)) \otimes \text{return } y; \\
\text{return } (x, y) \\
\end{array} \right. 
\]

\( \lambda (x) \) is a density of \( \zeta (x) \) with respect to \( \nu(x) \) for \( \xi \)-almost all \( x \)
Symbolic disintegration with a variety of base measures

\begin{verbatim}
= do { x ~ do { x ~ μ; κ(x) ⊗ return x }; y ~ do { y ~ ν(x); λ(x)(y) ⊗ return y }; return (x, y) } \quad κ \text{ is a density of } ξ \text{ with respect to } μ
= do { x ~ μ; κ(x) ⊗ do { y ~ ν(x); λ(x)(y) ⊗ return (x, y) } } \quad \text{monad laws}
= do { x ~ μ; y ~ ν(x); κ(x) ⊗ ( λ(x)(y) ⊗ return (x, y) ) } \quad \text{linearity (trivial commutativity)}
= do { x ~ μ; y ~ ν(x); (κ(x) · λ(x)(y)) ⊗ return (x, y) } \quad l \otimes (k \otimes m) = (l \cdot k) \otimes m
= do { (x, y) ~ (μ ⊗= ν); (κ(x) · λ(x)(y)) ⊗ return (x, y) } \quad \text{definition of } ⊗= \text{ and monad laws} □

Metropolis-Hastings sampling [Metropolis et al. 1953; Hastings 1970] is a popular inference technique that depends on a target distribution \( ξ : \mathbb{M} \alpha \) and a proposal kernel \( ζ : \alpha \rightarrow \mathbb{M} \alpha \). 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 \( ξ \Rightarrow ξ : \mathbb{M} \alpha^2 \) with respect to \( ξ \otimes ζ : \mathbb{M} \alpha^2 \) [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 \( ξ \Rightarrow ξ \) and \( ξ \otimes ζ \) with respect to some common base measure, using Proposition 2.7.

Example 2.14. The target distribution \( ξ : \mathbb{M} \mathbb{R} \) 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 \( ζ = λ x . \text{normal} x 1 : \mathbb{R} \rightarrow \mathbb{M} \mathbb{R}, \) so \( ζ(x) \) has a density with respect to lebesgue for all \( x \). By Proposition 2.13, with respect to the base measure lebesgue ⊗ lebesgue, the measure \( ξ \Rightarrow ξ \) has the density \( κ : \mathbb{R}^2 \rightarrow \mathbb{R}_+ \) defined by

\[ κ = \lambda(x, y). \exp(0.4(x - 0.4)^2 - 0.08x^4) \cdot \exp(-(y - x)^2/2)/\sqrt{2\pi}, \]

and the measure \( ζ \Rightarrow ξ \) has the density \( κ \circ \text{swap} \) where \( \text{swap}(y, x) = (x, y) \). Thus by Proposition 2.7, the acceptance ratio is

\[ \lambda(x, y). \exp(0.4(y - 0.4)^2 - 0.08y^4) \cdot \exp(-(x - y)^2/2)/\sqrt{2\pi} \cdot \exp(0.4(x - 0.4)^2 - 0.08x^4) \cdot \exp(-(y - x)^2/2)/\sqrt{2\pi}. \]

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 \( α_1 \times α_2 \), a single-site kernel \( ζ : α \rightarrow \mathbb{M} α \) can be built out of sub-kernels \( ζ_1 : α_1 \rightarrow \mathbb{M} α_1 \) and \( ζ_2 : α_2 \rightarrow \mathbb{M} α_2 \) as follows [Goodman et al. 2008; Wingate et al. 2011]: given \((x_1, x_2) : α_1 \times α_2\), 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

\[ ζ = \lambda(x_1, x_2). \frac{1}{2} \otimes do \{ x'_1 \sim ζ_1(x_1, x_2); return (x'_1, x_2) \} \]

\[ \otimes \frac{1}{2} \otimes do \{ x'_2 \sim ζ_2(x_1, x_2); return (x_1, x'_2) \}. \]

On the other hand, when \( α \) is a sum type \( α_1 + α_2 \), a reversible-jump kernel \( ζ : α \rightarrow \mathbb{M} α \) can be built out of sub-kernels \( ζ_1 : α_1 \rightarrow \mathbb{M} α_1 \) and \( ζ_2 : α_2 \rightarrow \mathbb{M} α_2 \) 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 \( α = \mathbb{R}^2 \), the target \( ξ \) has a density with respect to lebesgue ⊗ lebesgue, and the single-site kernel \( ζ \) in (19) is built out of sub-kernels \( ζ_1, ζ_2 : \mathbb{R}^2 \rightarrow \mathbb{M} \mathbb{R} \) that always return continuous measures. Still, the measures \( ξ \otimes ζ \) and \( ξ \otimes ξ \) do not have densities with respect to

\[ (\text{lebesgue} \otimes \text{lebesgue}) \otimes (\text{lebesgue} \otimes \text{lebesgue}) : \mathbb{M} (\mathbb{R}^2)^2. \]
Rather, they have densities with respect to the dependent product

$$(\text{lebesgue} \otimes \text{lebesgue}) \otimes = \lambda(x_1, x_2). \ (\text{lebesgue} \otimes \text{return } x_1) \otimes (\text{lebesgue} \otimes \text{return } x_2) : \mathbb{M} (\mathbb{R}^2)^2. \ (21)$$

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

$$\mu_1 \otimes \mu_2 = \{x_1 \sim \mu_1; \ \text{return } (\text{inl } x_1)\} \otimes \{x_2 \sim \mu_2; \ \text{return } (\text{inr } x_2)\}, \ (22)$$

where the measure $\mu_1$ is over $\alpha_1$, the measure $\mu_2$ is over $\alpha_2$, and the measure $\mu_1 \otimes \mu_2$ is over $\alpha_1 + \alpha_2$.

The challenge of computing acceptance ratios using these trickier base measures has motivated several publications [Green 1995; Tierney 1998; Wingate et al. 2011]. 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.

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

$$\text{fmap } f \text{st } \xi = \{x, y \sim \xi; \ \text{return } x\} = \lambda f. \xi(\lambda x.y.\ f(x)). \ (23)$$

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

$$\xi = \{x \sim \text{fmap } f \text{st } \xi; \ y \sim \kappa(x); \ \text{return } (x, y)\}, \ (24)$$

or in short, $\xi = \text{fmap } f \text{st } \xi \otimes = \kappa$. Typically, $\xi$ is normalized. In that case, so are $\text{fmap } f \text{st } \xi$ 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:

$$\xi = \{x \sim \text{normal } 3 \ 2; \ y \sim \text{normal } x \ 1; \ \text{return } (x, y)\} \ (25)$$

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

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

$$\{x \sim \text{normal } 3 \ 2; \ y \sim \text{normal } x \ 1; \ \text{return } x\} = \{x \sim \text{normal } 3 \ 2; \ 1 \otimes \text{return } x\} = \text{normal } 3 \ 2. \ (26)$$

Thus, we can read off from (25) that the conditional distribution over $y$ given $x$ is just $\lambda x. \text{normal } x \ 1$. 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 rewrite (25) using equation (5), then commute the binding of $y$ to the front:

$$\xi = \{x \sim \text{normal } 3 \ 2; \ y \sim \text{lebesgue}; \ (\text{dnorm } x \ 1)(y) \otimes \text{return } (x, y)\}
\quad \quad \quad = \{y \sim \text{lebesgue}; x \sim \text{normal } 3 \ 2; \ (\text{dnorm } x \ 1)(y) \otimes \text{return } (x, y)\}
\quad \quad \quad = \kappa' \otimes \text{lebesgue} \quad \text{where } \kappa' = \lambda y. \{x \sim \text{normal } 3 \ 2; \ (\text{dnorm } x \ 1)(y) \otimes \text{return } x\} \ (27)$$

Thus, the marginal distribution (fmap snd $\xi$) over $y$ is $(\lambda y. \kappa'(y)) \otimes \text{lebesgue}$, and the conditional distribution over $x$ given $y$ is $\lambda y. \kappa'(y)^{-1} \otimes \kappa'(y)$, the normalization of $\kappa'$. For this model, the marginal turns out equal to normal $3 \sqrt{5}$ and the conditional turns out equal to $\lambda y. \text{normal } 3 \frac{y^2}{\sqrt{5}}$. 

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.

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 $\text{normal } 3, 2$, and form our posterior belief about $x$, namely $\text{normal } \frac{3 + 4y}{ \frac{2}{\sqrt{5}} }$. This update illustrates the use of conditional distributions for making inferences.

The choice of the base measure $\text{lebesgue}$ in the calculation (27) is conventional because marginal distributions are commonly continuous, but arbitrary. The conditioning only depends on the normalization of $\kappa'$, 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 $\text{lebesgue}$ as the base measure, then $\kappa'$ 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 $\text{normal } x, 1$ to the interval $[0, 1]$, we can condition the model

$$\text{do } \{ x \leftarrow \text{normal } 3, 2; \ y \leftarrow \text{normal } x, 1; \ \text{return } (x, \max\{0, \min\{1, y\}\}) \}$$

(28)
on $y$ like in (27), but using the base measure in (10) rather than $\text{lebesgue}$. 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 $\kappa : \alpha \rightarrow \mathbb{M}\beta$ such that $\xi = \mu \otimes \kappa$.

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

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 $\lambda\nu\ .\ |\nu|$ is easy to implement as a program transformation, though it can produce integrals and sums that witness the fundamental intractability of probabilistic inference.

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 $\alpha$. 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 and the equational reasoning that justifies its correctness. Along the way, we review the syntax and semantics 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 $\kappa : \alpha \rightarrow \mathbb{M}\beta$ 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 $\kappa : \alpha \rightarrow \mathbb{M}\beta$ 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 $\alpha$ 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 $\odot$.

After this extension, for any given type $\alpha$ 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.

Third, in Section 6 we build a base-inferring disintegrator by adding $\mathbb{B}\mathbb{R}$ variables to the base measures produced by the genBase function. The restricted checking disintegrator turns a base measure containing these variables into constraints on them. 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 to infer $b : \mathbb{B} \alpha$ from $\mu : \mathbb{M} \alpha$. (The type $\mathbb{M} \alpha$ is isomorphic to $\mathbb{M} (\alpha \times 1)$, as described in Section 2.5.) It then disintegrates both $\xi$ and $\mu$ with respect to $b$, and divides the results to cancel $b$ out and produce $\kappa$.

### 3.1 Program syntax

Figure 7 shows the syntax of core Hakaru [Shan and Ramsey 2017], the language of probabilistic programs that all our disintegrators take as input and produce as output. The language is first-order and terminating, and features random choice and scoring as monadic side effects. Although this language is small and specific, the use of random choice and scoring as side effects to express distributions is established in probabilistic programming [Borgström et al. 2016; Narayanan et al. 2016; Culpepper and Cobb 2017; Staton 2017; Ścibior et al. 2018; Wand et al. 2018; Vákár et al. 2019], so we expect all of our results to extend easily to the first-order and terminating parts of other probabilistic languages.

The definition of terms at the top of Figure 7 does not include many operations on reals, because we use $\text{sqrt}$ (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 $\times$ to illustrate how to handle piecewise-invertible and coordinate-wise-invertible functions (such as absolute value and $+$.)

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 \( \nu \) is either an atomic term \( \mu \) or a constructor application. Thus, although the heap does not affect what terms are, it does affect which terms are atomic or in head normal form. Regardless, head normal forms always include measure terms and real literals \( r \).

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 measures over \( \alpha \).

The measure terms lebesgue, return \( e \), fail, and \( m_1 \odot m_2 \) 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 \( \text{do} \{ g; M \} \), where \( g \) is one of several kinds of bindings, also called guards. The typical such measure term is monadic bind, \( \text{do} \{x \leftarrow m; M\} \), 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 \( \mathbb{1} \). Also, we abbreviate \( x \leftarrow \text{return} \ e \) to let \( x = e \). Another kind of guard builds the measure term \( \text{do} \{ \text{factor} \ e; M \} \), which means to scale the measure \( M \) by the weight \( e \). We write this term as \( e \odot M \) for short. We also write \( \text{normal} \ e_1, e_2 \) as syntactic sugar for \( \text{dnorm} \ e_1 e_2 = \odot \text{lebesgue} \). (In turn, \( \odot \) is syntactic sugar defined in Definition 2.3.)

Following Shan and Ramsey [2017], to make the disintegrator easier to explain, sum types in core Hakaru are deconstructed by bindings like let \( \text{inl} \ x = e; M \) just means \( M\{x \mapsto e_1\} \), but if \( e \) is \( \text{inl} \ e_1 \), then the same term \( \text{do} \{ \text{let} \ \text{inl} \ x = e; M \} \) means the zero measure fail. Ordinary pattern matching on sum types can be recovered by duplicating a measure context \( M \):

\[
M \left[ \begin{array}{c}
\text{case } e \text{ of } \text{inl} \ x_1 \rightarrow e_1 \\
\text{inr} \ x_2 \rightarrow e_2
\end{array} \right] = \ \text{do} \{ \text{let} \ \text{inl} \ x_1 = e; M[e_1]\} \odot \ \text{do} \{ \text{let} \ \text{inr} \ x_2 = e; M[e_2]\}.
\] (29)

Booleans true, false can be encoded using the sum type \( \text{bool} = \mathbb{1} + \mathbb{1} \) 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 \( \text{observe} \ e \) to mean the guard \( \text{let} \ true = e \).

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:

\[
\text{do} \{ g_1; \ldots; g_n; M \} = \ \text{do} \{ g_1; \ldots \} \ \text{do} \{ g_n; M; \ldots \}.
\] (30)

### 3.2 Base-measure language

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

\[
\text{genBase}(\mathbb{R}) = \text{lebesgue} \quad \text{genBase}(\alpha \times \beta) = \text{genBase}(\alpha) \odot \text{genBase}(\beta) \\
\text{genBase}(\mathbb{1}) = \text{return} () \quad \text{genBase}(\alpha + \beta) = \text{genBase}(\alpha) \odot \text{genBase}(\beta)
\] (31)

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

### 3.3 Disintegration by equational reasoning

The problem of disintegrating a core Hakaru program appears intractable at first glance, because any mathematics is fair game to use to equate the two sides \( \xi \) and \( \mu \otimes \kappa \). Fortunately, it turns out that a few equational reasoning principles suffice for all the applications claimed in Section 2. Moreover, the reasoning can be generalized to variable base measures, as well as automated as
program transformations that do not perform unbounded search. In this subsection, we illustrate
the reasoning and its variable-base generalization using concrete example programs. The rest of
the paper then describes and justifies the automation for arbitrary input programs.

3.3.1 Semantics. Before we discuss equational reasoning, we first define the denotations being
equated. As is standard, a core Hakaru term denotes a function from environments to values, and
an environment is a tuple of values. For example, the term $2 \cdot x : \mathbb{R}$, in the scope of $y : \mathbb{R}$ and $x : \mathbb{R}$,
denotes the function $\lambda(y,x).2x$ from $\mathbb{R}^2$ to $\mathbb{R}$. To take another example, the closed term $\text{return } \mathbb{Z}$
denotes (the function that maps $()$ to) the Dirac distribution at $9$. These denotations are defined
compositionally, by induction on typing derivations as usual.

In this probabilistic language, the functions denoted by terms are measurable. Moreover, following
Staton [2017], a term of measure type denotes an $s$-finite kernel.

Definition 3.1. A kernel $\kappa$ from a measurable space $\alpha$ to a measurable space $\beta$ is a measurable
function from $\alpha$ to measures over $\beta$. We notate this as $\kappa : \alpha \rightarrow M \beta$.

A finite kernel is a kernel with a uniform bound on the total of the measure returned. That is,
$\kappa$ is finite if there exists $r < \infty$ such that for all $x \in \alpha$ we have $|\kappa(x)| < r$.

An $s$-finite kernel is a countable sum of finite kernels.

So in particular, a closed term of measure type denotes an $s$-finite measure, which is a countable
sum of finite measures. For example, the closed term $\text{lebesgue} : \mathbb{M} \mathbb{R}$ denotes the Lebesgue measure,
which is $s$-finite because it is the sum of the uniform distributions over the intervals $[n, n+1]$ where
$n \in \mathbb{Z}$. And the closed term $\text{do } (_\sim \text{lebesgue}; \text{return } ()) : \mathbb{M} \mathbb{1}$ denotes the infinite measure over
the unit type, which is $s$-finite because it is the sum of countably infinite copies of $\text{return } ()$.

Staton’s $s$-finiteness invariant provides us with two crucial assurances. The first assurance is
that monadic bindings denote. Treating measures as integrators, we write the semantic definition
$$[\text{do } \{ x \sim m; M \}]\ (\rho) = \lambda f. \ [m] (\rho) (\lambda x. \ [M] (\rho, x) (f)),$$
where the pair $(\rho, x)$ extends the environment $\rho$ with the value $x$ drawn from the measure $[m] (\rho)$.

The second assurance is that monadic bindings commute. We have the equation
$$\text{do } \{ x \sim m_1; y \sim m_2; M \} = \text{do } \{ y \sim m_2; x \sim m_1; M \}$$
as long as scope is respected: $M$ can use $x$ and $y$, but $m_1$ cannot use $y$ and $m_2$ cannot use $x$. This
commutativity equation, reminiscent of Fubini’s and Tonelli’s theorems, lets us treat a sequence of
bindings as a directed acyclic graph of dependencies, which is equivalent to a Bayes net. We use
commutativity pervasively below.

3.3.2 Fixed-base disintegration. In addition to commutativity, disintegration requires just two equa-
tional reasoning principles: density and reparameterization. To see how, consider the disintegration
shown in Figure 9. (This example is equivalent to (27), but the variables have been renamed so
that $y$ and $x$ are both drawn from normal $0 \ 1$.) On the left is the input program $\xi$, which draws
two random variables $x, y$ and then observes a quantity $t$ determined by them. On the right, to
condition on $t$, the program has been rewritten in the form $\mu \otimes\approx \kappa$, where the base measure $\mu$ is
lebesgue and the kernel $\kappa$ is $\lambda t. \text{do } \{ \ldots ; \text{return } (3 + 2 \cdot x) \}$. 

As shown in Figure 9, the disintegration is justified by a density step, a reparameterization step,
and elided applications of commutativity and monad laws. The density step (equating the left-hand
side to the middle) finds a density of normal $0 \ 1$ with respect to lebesgue, using equation (5). The
reparameterization step (equating the middle to the right-hand side) changes whether $x$ takes scope
over $t$ or vice versa, using the rule from integral calculus for changing the variable of integration:
$$\forall f, \quad \int_\mathbb{R} f(x, 3 + 2 \cdot x + y) \: dx = \int_\mathbb{R} \frac{1}{2} f((t - 3 - y)/2, t) \: dt$$ (34)
Fig. 9. Disintegration with respect to the Lebesgue base measure. Two reasoning steps equate the input program on the left to the output program on the right, which has the disintegrated form \( \text{lebesgue} \odot \cdots \). Dotted lines show the bindings affected by each step.

\[
\begin{align*}
\text{do } t &\sim \text{normal } 0 \ 1; \\
&x \sim \text{normal } 0 \ 1; \\
&\text{let } t = 3 + 2 \cdot x + y; \\
&\text{return } (t, 3 + 2 \cdot x) \\
\end{align*}
\]

Fig. 10. Disintegration with respect to a base measure that is a discrete-continuous mixture

\[
\begin{align*}
\text{do } t &\sim \text{normal } 0 \ 1; \\
&x \sim \text{normal } 0 \ 1; \\
&\text{let } t = 3 + 2 \cdot x + y; \\
&\text{observe } t < 0; \\
&\text{return } (t', 3 + 2 \cdot x) \\
\end{align*}
\]

Note that this step introduces the Jacobian \( \text{factor} \ 1/2 \) in the final result.

Automatic disintegration actually begins with the desired form of the right-hand side of Figure 9 and asks, how can the input program equal a program that begins with \( t \sim \text{lebesgue} \)? Driven by this desired form, and noticing that the input program determines \( t \) in terms of \( x \), the disintegrator attempts the reparameterization step, which leads it to attempt the density step. The success of the density step then fleshes out the middle program, which finally allows the success of the reparameterization step to flesh out the right-hand side. This intuitive description is detailed formally in Section 4 below.

3.3.3 Base-checking disintegration. What if the base measure is not \( \text{lebesgue} \)? For example, suppose we change the input program so that the observed quantity is clamped to be non-negative. That is, suppose we observe not \( t \) but \( \max(0, t) \). Because the probability of observing 0 is no longer zero, disintegration with respect to the Lebesgue base measure is no longer possible. Instead, we need a discrete-continuous mixture, at least \( \text{lebesgue} \odot \text{return} \ 0 \).

Figure 10 shows that the same equational reasoning principles suffice for this new base measure. To start, we decompose the input as the sum of two measures: one component (top) where \( t < 0 \)
The output, to the right in Figure 11, invokes the densities \( \text{divide}\,(\text{return}\,0)\,B\) and \( \text{divide}\,\text{lebesgue}\,B\), so they better exist. Thus, these occurrences of \( \text{divide}\) constrain the unknown base measure \( B\) such that \( \text{return}\,0\) and \( \text{lebesgue}\) both have densities with respect to it. In general, we can read off from the output of disintegration a set of \( \text{divide}\) constraints on base-measure variables, just as we can read off from the output of typing a set of constraints on type variables. Solving our two constraints yields the principal base measure \( \text{lebesgue} \odot \text{return}\,0\). That is, we have inferred that a base measure \( B\) works if and only if we can find a density of \( \text{lebesgue} \odot \text{return}\,0\) with respect to it. In particular, setting \( B = \text{lebesgue} \odot \text{return}\,0\) in Figure 11 recovers Figure 10.
4 A FIXED-BASE DISINTEGRATOR

Armed with motivation, grammar, and intuition, we are ready for our formal development. 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. (The differences between our disintegrator and Shan and Ramsey’s are described in Section 4.2.) Our new proof of soundness culminates in Theorem 4.2 and takes advantage of commutativity as described in Section 4.4.

Figure 12 shows the top four functions that make up our restricted base-checking disintegrator: check, \(<, \leq, \triangleleft\). 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. The semantic specification is the main theorem we prove about the function. Additional functions used are explained in the following subsections:

- Section 4.2 explains \( \in \text{dom}, \in \text{rng}, \circ, \text{inv}, \text{jacobian}, \text{reparam}, \) and \( \div \).
- Section 4.3 explains \( \triangleright, \triangleright\rightleftarrows, \text{fst}, \text{snd}, \text{outl}, \) and \( \text{outr} \).

At the top of Figure 12 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 12, the (meta) type \( \llbracket \alpha \times \beta \rrbracket \) means a core Hakaru term of type \( \llbracket \alpha \times \beta \rrbracket \), and the (meta) type \( \llbracket \alpha \rrbracket \) means a head normal form of type \( \alpha \).

Nondeterminism. The return type of check is \( \{ \llbracket \beta \rrbracket \} \), which means a set (of head normal forms of type \( \llbracket \beta \rrbracket \), in the empty heap \( [] \) initializing check). 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 12, the symbol \( \triangledown \) appears to relate two terms but actually relates two sets of terms. The right-hand side is the set of all terms \( \text{do } \{ t \leftarrow b; p \leftarrow e; \text{return } (t, p) \} \) where the subterm \( e \) belongs to the set check \( m b t \). The left-hand side is the singleton set containing the term \( m \).

Definition 4.1. Let \( S \) and \( T \) be two sets of core Hakaru terms. We write \( S \triangledown T \) 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 check \( m b t \) (if any), the denotation of \( \text{do } \{ t \leftarrow b; p \leftarrow e; \text{return } (t, p) \} \) equals the denotation of \( m \).

In other definitions (such as the next function \( \leftarrow \) in Figure 12), we write \( \perp \) for the empty set of terms and \( \triangledown \) for the union of two sets of terms. These are the only two ways to incur nondeterminism. Note that \( \perp \) is different from fail, which is (a singleton set of) a term that denotes the zero measure, and \( \triangledown \) is different from \( \otimes \), which constructs a term that sums two measures.

Continuations. For reasons explained in Section 4.1, the disintegrator is written in continuation-passing style: many functions take and return continuations of type \( \text{heap} \rightarrow \{ \llbracket M \rrbracket \} \) and maintain the heap as a piece of mutable state. We use the metavariable \( e \) for these continuations. They typically have the form \( \overline{M} \), defined as the function

\[
\overline{M} = \lambda h. \text{do } \{ h; M \},
\]

(35)
Constrain outcome

\[
\forall : [M \alpha] \rightarrow B \alpha \rightarrow [\alpha] \rightarrow (heap \rightarrow \{M \gamma\}) \rightarrow heap \rightarrow \{M \gamma\}
\]

\[
\ll e \ ll (do) v c h = \ll
\]

where \( u \) is atomic

\[
\ll (lebesgue) v c h = do \{(\ll (lebesgue \div b) v) v c h\}
\]

\[
\ll (return) e v c h = \ll e b v c h
\]

\[
\ll (fail) v c h = fail
\]

\[
\ll (do \{g; m\}) v c h = \ll m v c h
\]

\[
\ll e v c h = \ll (\lambda m. \ll m b v c) h
\]

where \( e \) is not in head normal form

Constrain op

\[
\ll do \{h; observe (e \in dom f); let t = f \odot e; M\} \ll do \{t \odot b; \ll f e b t M h\}
\]

\[
\ll : invertible \rightarrow [R] \rightarrow B R \rightarrow [R] \rightarrow (heap \rightarrow \{M \gamma\}) \rightarrow heap \rightarrow \{M \gamma\}
\]

\[
\ll f e b v c h = do \{observe (v \in rng f); factor (jacobian f b v); \ll e (reparam f b) (inv f @ v) c h\}
\]

Fig. 12. The restricted base-checking disintegrator
which implicitly builds a singleton set by wrapping the given heap $h$ around some measure term $M$.
Thus, the third line of Figure 12 defines \texttt{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 \texttt{check} initializes the heap to the empty $[]$, invokes $\Rightarrow$ on $m$ to get $v$,
invokes $\langle$ on $\texttt{fst} \; v$, and wraps the final heap around $\texttt{return} \; (\texttt{snd} \; v)$ to form the result.

In other definitions (such as the function $\langle$ in Figure 12), we write $h_0$ for the heap-to-heap
function
\[
\overline{h}_0 = \lambda h. [h; h_0],
\] (36)
which wraps the given heap $h$ around some heap $h_0$. We also define the reverse function-composition
operator $\swarrow$ by $(g \swarrow c)(h) = c(g(h))$, so that $\overline{h}_0 \swarrow M = \texttt{do} \; \{h_0; M\}$. To draw an example from the definition
of $\langle$, if $c = \overline{M}$ then $[\texttt{let} \; x = v; h_2] \swarrow c = \texttt{do} \; \{\texttt{let} \; x = v; h_2; M\}$. 

4.1 From specifications to implementations

All the specifications boxed in Figure 12 have the form
\[
\ldots \equiv \texttt{do} \; \{t \leftarrow b; \ldots\}, \tag{37}
\]
where the right-hand side invokes the function being specified and the left-hand side is made
of inputs to the function. Recall from Definition 4.1 that the symbol $\equiv$ relates two sets of terms; here the left-hand side is a singleton set, so the specification says that every term produced
nondeterministically on the right denotes the same as the term on the left. 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 $t \leftarrow b$. Whereas the
external interface \texttt{check} rewrites an arbitrary measure term $m$, the other functions $\langle$ (“constrain value”), $\ll$ (“constrain outcome”), and $\ll$ (“constrain op”) rewrite measure terms of specific forms,
focusing on an expression $e$, action $m$, or operation $f$ in the context of a heap $h$ and final action $M$.
These specifications assure soundness, not completeness: they all admit implementations that return no answer ($\bot$).

All the implementations in Figure 12 are derived from the semantic specifications by equational
proofs. By \textit{derive} [Hughes 1995; Hutton and Meijer 1996], we mean that the right-hand side of
each implementation equation in Figure 12 can be figured out by the following process. First, we
prove an instance of the specification in which the function call is replaced by a term. Then, we
match the function call against the term to obtain the right-hand side. This way, we figure out the
implementation and prove that it satisfies the specification at the same time. (Some technicalities
of the proofs are discussed in Section 4.4.)

For example, the case $\ll \texttt{fail}$ is derived and proven from the specification
\[
\texttt{do} \; \{h; t \leftarrow m; M\} \equiv \texttt{do} \; \{t \leftarrow b; \ll m \; b \; t \overline{M} \; h\} \tag{38}
\]
by substituting $\texttt{fail}$ for $m$ and equating both sides to $\texttt{fail}$:
\[
\texttt{do} \; \{h; t \leftarrow \texttt{fail}; M\} = \texttt{fail} = \texttt{do} \; \{t \leftarrow b; \texttt{fail}\} \quad \text{fail distributivity}
\]
\[
= \texttt{do} \; \{t \leftarrow b; \ll \texttt{fail} \; b \; t \overline{M} \; h\} \quad \text{definition of $\ll$} \tag{39}
\]
The first line of (39) proves an instance of the specification (38) in which the function call $\ll \texttt{fail} b t \overline{M} \; h$
is replaced by the term $\texttt{fail}$. Matching that function call against that term gives the implementation
equation $\ll \texttt{fail} \; b \; v \; c \; h = \texttt{fail}$ in Figure 12. Filling in that implementation equation allows us to
proceed with the second line of (39) and finish this case of the proof that $\ll$ satisfies (38).
Similarly, the case $\triangleq (m_1 \otimes m_2)$ is derived and proven from (38) by substituting $m_1 \otimes m_2$ for $m$ and using the induction hypotheses derived also from (38) by substituting $m_1$ and $m_2$ for $m$:

$$\text{do } \{h; t \leftarrow m_1 \otimes m_2; M\} = \text{do } \{h; t \leftarrow m_1; M\} \otimes \text{do } \{h; t \leftarrow m_2; M\}$$

$\triangleright$ distributivity

$$\exists \text{ do } \{t \leftarrow b; \triangleq m_1 b t \overline{M} h\}$$

$\triangleright$ induction hypothesis

$$\exists \text{ do } \{t \leftarrow b; \triangleq m_2 b t \overline{M} h\}$$

$\triangleright$ induction hypothesis

$$= \text{ do } \{t \leftarrow b; (\triangleq m_1 b t \overline{M} h \otimes \triangleq m_2 b t \overline{M} h)\}$$

$\triangleright$ distributivity

$$= \text{ do } \{t \leftarrow b; \triangleq (m_1 \otimes m_2) b t \overline{M} h\}$$

$\triangleright$ definition of $\triangleq$  (40)

All but the last line of (40) proves an instance of the specification (38) in which the function call $\triangleq (m_1 \otimes m_2) b t \overline{M} h$ is replaced by the term $\triangleq m_1 b t \overline{M} h \otimes \triangleq m_2 b t \overline{M} h$. Matching that function call against that term gives the implementation equation $\triangleq (m_1 \otimes m_2) b v c h = \triangleq m_1 b v c h \otimes \triangleq m_2 b v c h$ in Figure 12. Filling in that implementation equation allows us to proceed with the last line of (40) and finish this case of the proof that $\triangleq$ satisfies (38). (Duplicating the continuation to deal with $m_1$ and $m_2$ separately, as justified above, is the first of two reasons to write the disintegrator in continuation-passing style [Danvy and Filinski 1990].)

An example where one function calls another is the case $\triangleq (\text{return } e)$, which is derived and proven from (38) by substituting return $e$ for $m$ and using the induction hypothesis that is the specification of $\triangleq$:

$$\text{do } \{h; t \leftarrow \text{return } e; M\} = \text{do } \{h; \text{let } t = e; M\}$$

$\triangleright$ abbreviation

$$\exists \text{ do } \{t \leftarrow b; \triangleq e b t \overline{M} h\}$$

$\triangleright$ specification of $\triangleq$

$$= \text{ do } \{t \leftarrow b; \triangleq (\text{return } e) b t \overline{M} h\}$$

$\triangleright$ definition of $\triangleq$  (41)

Similarly, the definition of top-level check is derived and proven from the specifications of $\triangleright$ and $\triangleq$:

$$\textbf{Theorem 4.2.} \text{ For all } m : \mathbb{M}(\alpha \times \beta) \text{ and } b : \mathbb{B} \alpha, \text{ we have }$$

$$m \triangleright \text{ do } \{t \leftarrow b; p \leftarrow \text{check } m b t; \text{return } (t, p)\}. \tag{42}$$

In other words, for every term $M$ produced nondeterministically by check $m b t$, the denotations of $m$ and $\text{do } \{t \leftarrow b; p \leftarrow M; \text{return } (t, p)\}$ are equal. Here $t$ and $p$ are fresh variables.

$$\textbf{Proof.} \text{ We reason equationally from the right-hand side of (42) to the left-hand side, using many separately proven lemmas stated in figures:}$$

$$\text{do } \{t \leftarrow b; p \leftarrow \text{check } m b t; \text{return } (t, p)\}$$

$$= \text{ do } \{t \leftarrow b; \}$$

$$p \triangleright m(\lambda v. \triangleq(\text{fst } v) b t (\text{return } (\text{snd } v))) [];$$

$$\text{return } (t, p)\}$$

$\triangleright$ definition of check (Figure 12)

$$= \text{ do } \{t \leftarrow b;$$

$$\triangleright m(\lambda vh. \text{do } \{p \leftarrow (\text{fst } v) b t (\text{return } (\text{snd } v)) h;$$

$$\text{return } (t, p)\}) []\}$$

$\triangleright$ associativity of $\triangleright$ (Figure 17)

$$= \text{ do } \{t \leftarrow b;$$

$$\triangleright m(\lambda vh. \triangleq(\text{fst } v) b t (\lambda h'. \text{do } \{p \leftarrow \text{do } \{h'; \text{return } (\text{snd } v)\};$$

$$\text{return } (t, p)\}) h) []\}$$

$\triangleright$ associativity of $\triangleq$ (Figure 17)

$$= \text{ do } \{t \leftarrow b; \triangleright m(\lambda vh. \triangleq(\text{fst } v) b t (\text{return } (t, \text{snd } v)) h) []\}$$

$\triangleright$ monad laws
Three crucial aspects of this disintegrator are new compared to Shan and Ramsey’s.

A sample run. For concreteness, Figure 13 shows the automatic disintegrator performing essentially the same reasoning as Figure 9 depicts equationally. Amid all the detail, the main pattern to notice is that the result is emitted gradually by several meta-functions invoking each other:

- On line 7, the meta-function $\Rightarrow$ (“perform”) emits the binding of $z$ (an alias for $y$). The rest of the run proceeds under the scope of this $z$. (Emitting such bindings is the second of two reasons to write the disintegrator in continuation-passing style [Bondorf 1992; Lawall and Danvy 1994].)
- On line 8, the meta-function $\Leftarrow$ emits $\text{factor}$ $1/2$ to carry out the reparameterization step in Figure 9.
- On line 10, the meta-function $\Leftarrow$ emits $\text{dnorm}$ to carry out the density step in Figure 9.

4.2 Constraining with respect to a base measure

Three crucial aspects of this disintegrator are new compared to Shan and Ramsey’s.
Invertibles
\[ f ::= \text{mul} \ v \mid \text{div} \ v \mid \text{sqr}^+ \mid \text{sqr}^- \mid \text{sqrt}^+ \mid \text{sqrt}^- \]

Domain
\( (\in dom) : [\mathbb{R}] \rightarrow \text{invertible} \rightarrow [\mathbb{bool}] \)
\( (\in @) : \text{invertible} \rightarrow [\mathbb{R}] \rightarrow [\mathbb{R}] \)
\( e \in dom (\text{mul} \ v) = \text{true} \)
\( e \in dom (\text{div} \ v) = \text{true} \)
\( e \in dom \text{sqr}^+ = 0 \leq e \)
\( e \in dom \text{sqr}^- = e < 0 \)
\( e \in dom \text{sqrt}^+ = 0 \leq e \)
\( e \in dom \text{sqrt}^- = 0 < e \)

Apply invertible
\( \text{Invert} (\text{inv } f @) = (f @)^{-1} \)

Range
\( (\in rng) : [\mathbb{R}] \rightarrow \text{invertible} \rightarrow [\mathbb{bool}] \)
\( e \in rng \ f = e \in dom (\text{inv } \ f) \)
\( e \in rng \ f = e \in dom (\text{inv } \ f) \)
\( \text{diff} : \text{invertible} \rightarrow [\mathbb{R}] \rightarrow [\mathbb{R}] \)
\( \text{diff} (\text{mul} \ v) = v \)
\( \text{diff} (\text{div} \ v) = v^{-1} \)
\( \text{diff} \text{sqr}^+ = 2 \cdot v \)
\( \text{diff} \text{sqr}^- = 2 \cdot v \)
\( \text{diff} \text{sqrt}^+ = (2 \cdot \text{sqrt} \ v)^{-1} \)
\( \text{diff} \text{sqrt}^- = (2 \cdot \text{sqrt} \ v)^{-1} \)

Fig. 14. Invertible functions and operations on them

First, we add a base-measure argument \( b : \mathbb{R} \alpha \) to every function in Figure 12. The case \( \ll e \ (\text{return} \ ()) \) fleshes out Shan and Ramsey’s remark that “on countable spaces, life is easy.” The case \( \ll e \ (b_1 \otimes b_2) \) fleshes out their remark that “other types \( \alpha \) such as \( \alpha = \mathbb{R} \times \mathbb{R} \) can be handled by successive disintegration.” The case \( \ll e \ (b_1 \otimes b_2) \) adds handling for disjoint sums, exactly as defined and motivated in equation (22). The fact that only \( \ll \) inspects \( b \), and only if \( \alpha \) is not \( \mathbb{R} \), helps us handle unknown base measures over \( \mathbb{R} \) in Section 6.

Second, we introduce the function \( \ll ("\text{constrain op}") \) to encapsulate a repeated pattern in how invertible operations are handled. Figure 14 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 \( \ll \) encounters an operation that is invertible, such as \( \text{sqrt} \ e = \text{sqrt}^+ \otimes e \), it hands it to \( \ll \); when \( \ll \) 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 \( \ll \).

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 \( \text{jacobian} \) and \( \text{reparam} \) 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 \( \ll \). The function \( \div ("\text{divide}") \) finds a density of one base measure over \( \mathbb{R} \) with respect to another; it is used by \( \ll \). We now turn to these three functions. Figure 15 shows their informal types, semantic specifications (boxed), and implementations.

### 4.2.1 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...
Reparameterize a base measure with a Jacobian

\[
\begin{align*}
\text{jacobian} : \text{invertible} & \rightarrow \mathbb{R} \rightarrow [\mathbb{R}] \rightarrow [\mathbb{R}] \\
\text{reparam} : \text{invertible} & \rightarrow \mathbb{R} \rightarrow \mathbb{R} \\
jacobian f \lebesgue & = \lambda y. |\text{diff} (\text{inv } f) y| \\
\text{reparam } f \lebesgue & = \lebesgue
\end{align*}
\]

Divide base measures

\[
(\div) : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \{[\mathbb{R}] \rightarrow \mathbb{M}, 1\}
\]

\[
\lebesgue \div \lebesgue = \lambda v. \text{return } ()
\]

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 \$\lhd \$, which in turn calls \text{jacobian} and \text{reparam}, defined in Figure 15.

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 \text{normal } 0 1 but with its outcome multiplied by 3. According to equation (6), we seek \$\kappa : \mathbb{R} \rightarrow \mathbb{R}_+ \$ such that

\[
(\text{normal } 0 1)(\lambda x. f(3 \cdot x)) = \lebesgue(\lambda y. \kappa(y) \cdot f(y))
\]

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

\[
\int_\mathbb{R} (\text{dnorm } 0 1)(x) \cdot f(3 \cdot x) \, dx = \int_\mathbb{R} \kappa(y) \cdot f(y) \, dy.
\]

To match the two sides, we need to change the integration variable from \$x \rightarrow y = 3 \cdot x \$. Equivalently, we need to express—or \text{reparameterize}—the Lebesgue measure over \$y \$ in terms of \$x = y/3 \$. Using integral calculus, we calculate

\[
\int_\mathbb{R} (\text{dnorm } 0 1)(x) \cdot f(3 \cdot x) \, dx = \int_\mathbb{R} (\text{dnorm } 0 1)(y/3) \cdot f(y) \cdot (dx/\!dy) \, dy,
\]

in which \$dx/\!dy = 1/3 \$. Hence, we find the density \$\kappa = \lambda y. (\text{dnorm } 0 1)(y/3) \cdot (1/3) \$. Using the disintegrator to find the same density, the top-level call

\[
\text{check } (\text{do } \{ x \sim \text{normal } 0 1; \; \text{return } (3 \cdot x, ()) \}) \lebesgue y
\]

eventually turns into

\[
\lhd (3 \cdot x) \lebesgue y (\text{return } ()) [x \sim \text{normal } 0 1]
\]

and then into

\[
\lhd (\text{mul } 3) x \lebesgue y (\text{return } ()) [x \sim \text{normal } 0 1],
\]

which calls \text{jacobian} (\text{mul } 3) \lebesgue. The \text{jacobian} function differentiates \$\text{inv } (\text{mul } 3) = \text{div } 3 \$ and returns \$\lambda y. 1/3 \$, hence computing the Jacobian factor necessary to change the integration variable. After \$\lhd \$ emits this \text{factor } 1/3, it proceeds to constrain the value of \$x \$ to be \$y/3 \$, by calling

\[
\lhd x \lebesgue (y/3)(\text{return } ()) [x \sim \text{normal } 0 1].
\]
4.2.2 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 $\alpha$. That is the distribution of $\text{fst} \; v$ in check in Figure 12, 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 $\xi$ with respect to a given base measure $v$. This reduction takes place in Figure 12, as check uses $\leftarrow$, and $\leftarrow$ in turn uses $\equiv$:

- In Figure 12, the case $\equiv \text{lebesgue} \; b$ faces the problem of finding a density of lebesgue with respect to $b$.
- In the simplified transcript in Figure 13, the call to $\equiv (\text{normal} \; 0 \; 1) \; \text{lebesgue}$ on line 9 faces the problem of finding a density of normal 0 1 with respect to lebesgue.

The density of a primitive measure with respect to a given base is found by the call $\equiv \xi \; v$ 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, normal, Beta, or Gamma distributions. For each primitive measure $\xi$, the $\equiv$ function should choose an intermediate measure $\mu$ that is represented in the base-measure language, and emit a density of $\xi$ with respect to $\mu$ (unless $\xi = \mu$, as in the case $\equiv \text{lebesgue}$). It then remains to find a density of the base measure $\mu$ with respect to the base measure $v$, and that is the job of the auxiliary function $\div$, defined in Figure 15.

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 $\mathbb{M} \; 1$, so $\div$ returns not 1 but its isomorphic value return ()

4.3 Partial evaluation

The functions $\Rightarrow$ ("evaluate") and $\Rightarrow$ ("perform"), used in Figure 12, perform lazy partial evaluation [Jørgensen 1992; Fischer et al. 2008]. Figure 16 shows their informal types, semantic specifications (boxed), and implementations. This part of the disintegrator is standard and unchanged from Shan and Ramsey’s—they do not even take a new base-measure argument.

The job of partial evaluation is to bring a given term into head normal form (indicated by the meta-type $[\alpha]$ and the metavariable $v$) while preserving its meaning. This job serves two purposes in automatic disintegration, each illustrated in the sample run in Figure 13.

First, given a measure over a product space $\alpha \times \beta$, disintegration needs to decompose the pair into an $\alpha$ to be constrained and a $\beta$ to be inferred, but the pair expression may be buried deep inside the input program and even occur on multiple control paths. Partial evaluation digs out this pair expression; hence check calls $\Rightarrow$ in Figure 12. For example, on lines 2 and 3 of Figure 13, partial evaluation digs out the pair expression $(3 + 2 \cdot x + y, 3 + 2 \cdot x)$, so that the first component $3 + 2 \cdot x + y$ can be constrained by $\leftarrow$ and the second component $3 + 2 \cdot x$ can be inferred by return.

Second, given an invertible operation to constrain, $\leftarrow$ needs to emit a jacobian factor outside the scope of the heap, so the invertible must not use any variable bound in the heap. The grammar of

invertibles in Figure 14 enforces this requirement, because the $v$ in mul $v$ and div $v$ must be a head normal form of type $\mathbb{R}$, and (an easy induction on the grammar of head normal forms shows that) no such term uses any variable bound in the heap. Partial evaluation establishes this requirement; hence $\leftarrow (e_1 \cdot e_2)$ calls $\Rightarrow$ in Figure 12. For example, on lines 5 and 6 of Figure 13, partial evaluation turns the heap-bound variable $y$ into the non-heap-bound variable $z$, so that a jacobian factor can be emitted outside the scope of $y$.

To preserve semantics, partial evaluation may accumulate bindings onto the heap as well as emit bindings into the output. Both possibilities are illustrated in the sample run in Figure 13: between lines 2 and 3, the bindings $y \leftarrow \text{normal} \; 0 \; 1$; $x \leftarrow \text{normal} \; 0 \; 1$ are accumulated onto the heap; between lines 6 and 7, the binding $z \leftarrow \text{normal} \; 0 \; 1$ is emitted into the output.
The semantic specifications (boxed) above, including Theorem 4.2 (the soundness of the external
interface check), follow from three inductively proven properties of the 4 workhorse functions
\(\lll, \llll, \rrr, \rrrr\). These properties are specified in Figure 17.

4.4 Proof technicalities

The semantic specifications (boxed) above, including Theorem 4.2 (the soundness of the external
interface check), follow from three inductively proven properties of the 4 workhorse functions
\(\lll, \llll, \rrr, \rrrr\). These properties are specified in Figure 17.

These sets of terms are partially ordered by the subset relation, and these functions (including parametricity, and then the semantic specifications) is then proven by induction on the step-index as the least fixed point of \( \text{contract} \). 

Technically, Figures 12 and 16 define a function \( \text{contract} \) as the least fixed point of \( \text{contract} \) in the step-index \( n \). We can thus define the 4-tuple of functions \( \langle \text{contract}, \text{contract}, \text{contract}, \text{contract} \rangle \) as the least fixed point of \( \text{contract} \) in the step-index \( n \). Each property of the 4 functions (commutativity, associativity, parametricity, and then the semantic specifications) is then proven by induction on the step-index \( n \).

### Commutativity

\[
\begin{align*}
\text{do} \{ g \triangleright e \ b \ p c h \} & = \langle e b p \rangle (\lambda h'. \text{do} \{ g \; \triangleright c h' \}) h \\
\text{do} \{ g \triangleright m b v c h \} & = \langle e b p \rangle (\lambda h'. \text{do} \{ g \; \triangleright c h' \}) h \\
\text{do} \{ g \triangleright e k h \} & = \langle e b p \rangle (\lambda h'. \text{do} \{ g \; \triangleright c h' \}) h \\
\text{do} \{ g \triangleright m k h \} & = \langle e b p \rangle (\lambda h'. \text{do} \{ g \; \triangleright c h' \}) h \\
\end{align*}
\]

### Associativity

\[
\begin{align*}
\text{do} \{ p \triangleleft e b p c h; M \} & = \langle e b p \rangle (\lambda h'. \text{do} \{ p \triangleleft c h' \}; M) h \\
\text{do} \{ p \triangleleft m b v c h; M \} & = \langle e b p \rangle (\lambda h'. \text{do} \{ p \triangleleft c h' \}; M) h \\
\text{do} \{ p \triangleleft e k h; M \} & = \langle e b p \rangle (\lambda h'. \text{do} \{ p \triangleleft c h' \}; M) h \\
\text{do} \{ p \triangleleft m k h; M \} & = \langle e b p \rangle (\lambda h'. \text{do} \{ p \triangleleft c h' \}; M) h \\
\end{align*}
\]

### Parametricity

\[
\begin{align*}
\langle e b p \rangle (\lambda v \mathcal{M} \{ v \mapsto e' \}) h & = \langle e b p \rangle (\lambda v \mathcal{M} \{ v \mapsto e' \}) h \\
\langle e b p \rangle (\lambda v \mathcal{M} \{ v \mapsto e' \}) h & = \langle e b p \rangle (\lambda v \mathcal{M} \{ v \mapsto e' \}) h \\
\end{align*}
\]

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

Fig. 17. Inductively proven properties of the 4 workhorse functions \( \triangleright, \triangleleft, \triangleright, \triangleright \)

(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 commute \( g \) with whatever a workhorse functions emits.

(2) Associativity says that \( [\text{constraining or evaluating } e \text{ or } m \text{ with the continuation } c \text{ or } k] \) followed by the final action \( M \) is same as constraining or evaluating \( e \) or \( m \) with the continuation \( c \text{ or } k \) followed by the final action \( M \). This property is reminiscent of what Thielecke [2003] calls naturality and Ahmed and Blume [2011] call continuation shuffling.

(3) Parametricity says that the backward functions \( \triangleright \) and \( \triangleright \) 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 \( \triangleright, \triangleright, \triangleright, \triangleright \) uses recursion and nondeterminism, so assembling all the equational derivations into an overall soundness proof is not entirely trivial.
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

\[
\text{lebesgue } \odot \text{return } 0 \odot \text{return } 1 \\
\text{lebesgue } \odot x. \text{return } x \\
(\text{lebesgue } \odot \text{lebesgue}) \odot x. ((\text{lebesgue } \odot \text{return } (\text{fst } x)) \odot \\
(\text{lebesgue } \odot \text{return } (\text{snd } x)))
\]

in (10),

\[
(\text{lebesgue } \odot \text{lebesgue}) \odot x. ((\text{lebesgue } \odot \text{return } (\text{fst } x)) \odot \\
(\text{lebesgue } \odot \text{return } (\text{snd } x))) : \text{M} (\text{R}^2)^2
\]

(a diagonal), or

In contrast, the base-measure language in Section 3.2 only allows independent products and disjoint sums of Lebesgue and unit measures. Accordingly, we generalize the base-measure language in two ways.

1. Base measures over \( \text{R} \) have the form \( \text{mix } l [e_1, \ldots] \), distinct from any measure term in the term language. Here \( [e_1, \ldots] \) 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 \( \text{mix } ff [e_1, \ldots] \) means \( \text{return } e_1 \odot \cdots \), and the base measure \( \text{mix } tt [e_1, \ldots] \) means \( \text{lebesgue } \odot \text{return } e_1 \odot \cdots \). Hence \( \text{lebesgue} \) can be expressed as \( \text{mix } tt [\ ] \), and \( \text{return } x \) can be expressed as \( \text{mix } ff [x] \).

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

These changes are summarized at the top of Figure 18. The rest of the figure updates the functions \( \triangleleft, \triangleleft \ll, \text{jacobian}, \text{reparam} \) 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 \( \odot \) take advantage of the extended base-measure language. Constraining the second element of a pair \( \text{snd } e \) now uses a base measure that can depend on what the first element was constrained to be (\( \text{fst } v \)). 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 same sequence of justifications derive both \( \text{mix } tt [\ ] \div \text{mix } tt [0] \) and \( \text{mix } ff [0] \div \text{mix } tt [0] \). We show the former case, which means dividing \( \text{lebesgue} \) by \( \text{lebesgue } \odot \text{return } 0 \):

\[
\text{lebesgue } = \text{lebesgue } \odot \text{fail} \\
= \{ \text{do } t \leftarrow \text{lebesgue}; \text{observe } (t \neq 0); \text{return } t \} \\
\odot \{ \text{do } t \leftarrow \text{return } 0; \text{observe } (t \neq 0); \text{return } t \} \\
= \{ \text{do } t \leftarrow \text{lebesgue } \odot \text{return } 0; \text{observe } (t \neq 0); \text{return } t \} \odot \text{distributivity}
\]

The fourth case of \( \div \) invokes a term-equality checker, a meta-function that takes two real terms as input and either declares them equal or declines to declare them equal. We require a term-equality checker that is sound—if it declares two terms equal, then their denotations must actually be equal—but it need not be complete. We also require in Section 6 below that this checker computes an equivalence relation that is preserved by substitution and \( \odot \):

1. It must declare every term equal to itself.
2. Whenever it declares \( e_1 \) equal to \( e_2 \), it must also declare \( e_2 \) equal to \( e_1 \).
3. Whenever it declares \( e_1 \) equal to \( e_2 \) and \( e_2 \) equal to \( e_3 \), it must also declare \( e_1 \) equal to \( e_3 \).
Symbolic disintegration with a variety of base measures

Bases \( b := \text{mix } l \{ e, \ldots \} \mid \text{return } () \mid b \otimes \lambda x. b \mid b \otimes b \)

Continuity \( l := \text{ff } \mid \text{tt } \)

\[ \begin{array}{c}
\text{return } () : B 1 \\
\text{mix } l \{ e, \ldots \} : B \ R \\
\end{array} \]

\[ \begin{array}{c}
b_1 : B \alpha \\
b_2 : B \beta \\
b_1 \otimes \lambda x. b_2 : B (\alpha \times \beta) \\
b_1 \otimes b_2 : B (\alpha + \beta) \\
\end{array} \]

\( \langle : [\alpha] \rightarrow B \alpha \rightarrow [\alpha] \rightarrow (heap \rightarrow \{ [M \ y] \}) \rightarrow heap \rightarrow \{ [M \ y] \} \)

\( \langle e \rangle \quad (b_1 \otimes \lambda x. b_2) v \ c \ h = \langle (\text{fst } e) b_1 \ (\text{fst } v) \ (\langle (\text{snd } e) (b_2 \{ x \mapsto \text{fst } v \}) \ (\text{snd } v) \ c) \ h \)

\( \langle u \rangle \quad b \ v \ c \ h = \text{do } \{ () \sim (\text{mix ff } [u] \div b) v, c \ h \} \quad \text{where } u : R \text{ is atomic} \)

\( \langle r \rangle \quad b \ v \ c \ h = \text{do } \{ () \sim (\text{mix ff } [r] \div b) v, c \ h \} \quad \text{where } r : R \text{ is a literal} \)

\( \langle : \{ [M \alpha] \rightarrow B \alpha \rightarrow [\alpha] \rightarrow (heap \rightarrow \{ [M \ y] \}) \rightarrow heap \rightarrow \{ [M \ y] \} \)

\( \langle \text{lebesgue } b \rangle \quad v \ c \ h = \text{do } \{ () \sim (\text{mix tt } [\lambda v. b] \div b) v, c \ h \} \)

\( \text{jacobian } \circ \text{invertible } \rightarrow B R \rightarrow [R] \rightarrow [R] \quad \text{reparam } \circ \text{invertible } \rightarrow B R \rightarrow B R \)

\( \text{jacobian } f (\text{mix ff } \_1) = \lambda v. 1 \quad \text{reparam } f (\text{mix } l \{ e_1, \ldots \} ) = \text{mix } l \{ \text{inv } f @ e_1, \ldots \} \)

\( \text{jacobian } f (\text{mix tt } [e_1, \ldots ]) = \lambda v. \text{if } v = e_1 \nu \ldots \text{ then } 1 \text{ else } \text{diff } (\text{inv } f) v \)

\( (\div) : B R \rightarrow B R \rightarrow \{ [R] \rightarrow M 1 \} \)

\( \text{mix tt } [\_1] \div \text{mix ff } \_1 = \perp \)
\( \text{mix tt } [e_1, \ldots ] = \lambda v. \text{do } \{ \text{observe } (v \neq e_1 \land \cdots ) ; \text{return } () \} \)

\( \text{mix ff } [\_1] \div \_1 = \text{fail} \)
\( \text{mix ff } [e_1, \ldots ] = \lambda v. \text{do } \{ \text{observe } (v = e_1) ; \text{factor } 1 \# [i | v = e_1] ; \text{return } () \} \)

\( \text{mix ff } [e] \div \_1 [e_1, \ldots ] = \perp \quad \text{otherwise} \)

\( \text{mix ff } [e] \div \_1 [e_1, \ldots ] = [\lambda v. (\text{mix ff } [e] \div b) v \ominus (\text{mix } l \{ e_1, \ldots \} \div b) v] \)

Fig. 18. Changes to handle base measures that are either mixtures of the Lebesgue measure and point masses or dependent products. The typing rules introduce a new judgment form \( b : B \alpha \), which says that \( b \) is a base measure over \( \alpha \). In the new definition of \( \div \), the syntactic sugar \( \text{observe } \cdots \) for \( \text{let true } \cdots \) is defined in Section 3.1, and whether two terms are “known to be equal” is checked as described at the end of Section 5.

(4) Whenever it declares \( e_1 \) equal to \( e_2 \), it must also declare \( e_1 \{ x \mapsto e \} = e_2 \{ x \mapsto e \} \) and \( f @ e_1 \) equal to \( f @ e_2 \), for all variables \( x \), terms \( e \), and invertibles \( f \).

Syntactic equality is one basic checker that satisfies these requirements, and that is what our implementation uses, but a more complete checker would allow \( \div \) and the overall disintegration to succeed more often.

6 A BASE-INFERRING DISINTEGRATOR

Recall that disintegration is a ternary relation between a joint measure, a base measure, and a kernel (Definition 2.19). The joint measure is an input, and the kernel is an output, but the base measure can be either an input or an output. In this section, we turn from checking the base measure as an input to inferring it as an output.

On one hand, the ideal base-checking disintegrator would answer the question:

Given a joint measure and a base measure, what is a kernel?
To this end, our check disintegrator above is sound but incomplete. On the other hand, the ideal base-inferring disintegrator would answer the question,

Given a joint measure, with respect to which base measures does a kernel exist?

To this end, our base-inferring disintegrator precisely answers a different question,

Given a joint measure, with respect to which base measures can check find a kernel?

Example 6.1. A simple example of base inference is the clamped measure whose disintegration is discussed in Sections 3.3.3 and 3.3.4. That measure has a disintegration not with respect to the Lebesgue base \( \text{mix tt} \{ \} \) but with respect to the mixture base \( \text{mix tt} \{ 0 \} \). The check disintegrator can find that density, as well as another density with respect to \( \text{mix tt} \{ 0, 1 \} \). However, it may fail with respect to \( \text{mix tt} \{ 0^2 \} \), depending on whether the term-equality checker used in \( \div \) is complete enough to affirm that \( 0^2 = 0 \). In general, check finds a kernel with respect to precisely those bases \( \text{mix tt} \{ e_0, \ldots \} \) where \( e_0 \) is known to equal 0. This fact about check is what our base-inferring disintegrator computes.

This example highlights the fact that the binary function \( \div \) is the only place where base checking can fail—by returning \( \bot \), the empty set of solutions, in Figure 18. It also illustrates that the base measures for which a kernel can be found are not unique. Quite to the contrary, given a joint measure, whenever a kernel can be found with respect to a base \( b \), one can be found with respect to every base above \( b \) in the preorder defined below.

Definition 6.2. Let \( b, b' : \mathbb{B} \alpha \) be two bases over the same space \( \alpha \), following the grammar at the top of Figure 18. We define when \( b \) is divisible by \( b' \) (notated \( b <: b' \) ) inductively:

1. If \( b = \text{mix tt} \{ e_1, \ldots \} \) and \( b' = \text{mix tt} \{ e_1', \ldots \} \) over \( \mathbb{R} \), then \( b <: b' \) if and only if \( b \div b' \neq \bot \).
   
   In other words, \( b <: b' \) if and only if \( l \) implies \( l' \) and every element of \( \{ e_1, \ldots \} \) is known to equal some element of \( \{ e_1', \ldots \} \).

2. If \( b = b' = \text{return} () \) over \( \mathbb{R} \), then \( b <: b' \) always.

3. If \( b = b_1 \oplus \lambda x. b_2 \) and \( b' = b'_1 \oplus \lambda x. b'_2 \) over \( \alpha_1 \times \alpha_2 \), then \( b <: b' \) if and only if \( b_1 <: b'_1 \) and \( b_2 <: b'_2 \). The fresh variable \( x \) may appear free in \( b_2 \) and \( b'_2 \).

4. If \( b = b_1 @ b_2 \) and \( b' = b'_1 @ b'_2 \) over \( \alpha_1 + \alpha_2 \), then \( b <: b' \) if and only if \( b_1 <: b'_1 \) and \( b_2 <: b'_2 \).

Proposition 6.3. The relation \( <: \) is a preorder. Moreover, it is preserved by substitution and @.

Proof. By induction. In case 1 of Definition 6.2, we use the assumption that the term-equality checker used in \( \div \) in Figure 18 computes a preorder that is preserved by substitution and @.

We illustrate a small slice of this preorder as a graph in Figure 19, where nodes represent base measures, and an edge from \( b \) to \( b' \) means that \( b <: b' \). With regard to Example 6.1, the figure shows that \( \text{mix tt} \{ \} <: \text{mix tt} \{ 0 \} <: \text{mix tt} \{ 0, 1 \} \).

Fig. 19. The preorder \(<:\) ("is divisible by"), where an edge from \( b \) to \( b' \) means \( b <: b' \)
Symbolic disintegration with a variety of base measures

PROPOSITION 6.4. If one of the functions check, ⊢, ⇐, < succeeds on a base b, then it also succeeds on every base b′ such that b ⊆ b′.

PROOF. By induction on the step-index (Section 4.4), using Proposition 6.3.

Due to Proposition 6.4, there is often an infinite number of bases to choose from. Fortunately, it turns out in Section 6.5 below that the bases to choose from can be summarized by a principal base.

Definition 6.5. Given a joint measure m, we say that a base measure b is principal if for every base measure b′, the check disintegrator can find a kernel with respect to b′ (that is, check m b′ t ≠ ⊥, where t is a fresh variable) if and only if b ⊆ b′. (In particular, check m b t ≠ ⊥ because b ⊆ b.)

In the rest of this section, we explain why base inference is useful, then describe how to infer a principal base.

6.1 Motivating base inference

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 18. That unrestricted disintegrator is described in Section 7 below.

6.2 Implementing base inference

We implement base inference like constraint-based type inference [Wand 1987a,b; Pierce 2002, chapter 22]: 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 20. This example is motivated by Metropolis-Hastings sampling (Section 2.3): the input program, in the upper-left corner of Figure 20, can alternatively be written as (κ = Ω ξ) ⊗ return (), composed of the target distribution

\[ \xi = \text{normal} 0 1 \]

and the proposal kernel

\[ \kappa = \lambda x. (1/2) \odot \text{normal} x 0.1) \odot ((1/2) \odot \text{return} x) \]

(similar to the composite kernel in (19) but along one \( \mathbb{R} \) dimension only).

Our first step of base inference is to represent the unknown base measure by introducing base variables. Because the input measure is over \( (\mathbb{R} \times \mathbb{R}) \times \mathbb{1} \), the base measure must be over \( \mathbb{R} \times \mathbb{R} \). The grammar of bases in Figure 18 mandates that all base measures over \( \mathbb{R} \times \mathbb{R} \) must have the form

\[ \text{mix} \ldots \odot = \lambda x. \text{mix} \ldots \]

in which the term variable \( x : \mathbb{R} \) can appear in the second \( \text{mix} \). We represent this form by \( B_1[\_] \odot = \lambda x_1, B_2[x_1], \) returned by \text{genBase} in Figure 18. Here the base variables \( B_1 \) and \( B_2 \) each range over a base measure \( \text{mix} \ldots \) over \( \mathbb{R} \). The notation \( B_2[x_1] \) means that \( B_2 \) can contain zero or more occurrences of a hole, plugged by a fresh term variable \( x_1 : \mathbb{R} \).
do \( \{ x \leftarrow \text{normal}\ 0\ 1; \}
1/2 \odot \text{do } \{ y \leftarrow \text{normal}\ x\ 0.1; \text{ return } ((y, x), ()) \} \)
\odot 1/2 \odot \text{ return } ((x, x), ())

\begin{align*}
\text{genBase} & \quad :  \mathbb{M} \left[ \mathbb{R} \times \mathbb{R} \right] \\
\text{do} \quad \{ \lambda : \text{divide} \ (\text{mix} \ \check{tt} \ B_1) \ (\text{fist} \ t); \\
\lambda : \text{divide} \ (\text{mix} \ \check{tt} \ B_2) \ (\text{snd} \ t); \}
\text{factor} \ (\text{dnorm} \ 0 \ 1) \ (\text{snd} \ t); \}
\text{factor} \ 1/2; \\
\text{factor} \ (\text{dnorm} \ (\text{snd} \ t) \ 0.1) \ (\text{fist} \ t); \}
\text{return} () \}
\odot \text{do} \quad \{ \lambda : \text{divide} \ (\text{mix} \ \check{tt} \ B_1) \ (\text{fist} \ t); \\
\lambda : \text{divide} \ (\text{mix} \ \check{tt} \ B_2) \ (\text{snd} \ t); \}
\text{factor} \ (\text{dnorm} \ 0 \ 1) \ (\text{snd} \ t); \}
\text{factor} \ 1/2; \\
\text{return} () \}
\end{align*}

\begin{align*}
\text{mix} \ \check{tt} \ \check{tt} \ \lambda : \text{substitute} \ 
\begin{align*}
\text{mix} \ \check{tt} \ \check{tt} \ \lambda : \\
\text{mix} \ \check{tt} \ \check{tt}
\end{align*}
\end{align*}

Fig. 20. Inferring a principal base measure (lower left) for an input program (upper left)

Our second step of base inference is to invoke the \textit{check} disintegrator with respect to the unknown base. Whenever \textit{check} matches a base variable against a pattern \textit{mix} \ldots : \mathbb{B} \mathbb{R}, it suspends that part of the program transformation and produces a \textit{residualized} term. This suspension only happens in the functions \textit{jacobian}, \textit{reparam}, and \div. For example, the call \textit{mix} \ \check{tt} \ \check{tt} \div B_1) produces the term \textit{divide (mix tt \check{tt} B_1)} in Figure 20, which represents an unknown density of the Lebesgue measure with respect to the unknown base \textit{B_1}. Each occurrence of \textit{divide} in the output of \textit{check} expresses a constraint on a base variable, such as \textit{mix} \ \check{tt} \ \check{tt} \div B_1). Collecting these constraints yields a set of pairs such as (\textit{mix} \ \check{tt} \ \check{tt} \div B_1)). The \textit{check} disintegrator succeeds if and only if every constraint holds.

Our third step of base inference is to reduce all the collected constraints on plugged base variables to one constraint per unplugged base variable. In Figure 20, the \textit{solve} step turns each constraint on a plugged base variable (such as the constraint (\textit{mix} \ \check{tt} \ \check{tt} B_2) on \textit{B_2}) into a constraint on the base variable unplugged (such as the constraint \textit{mix} \ \check{tt} \ \check{tt} B_3) on \textit{B_3}, in which the fresh term \textit{variable} \textit{x_3} : \mathbb{R} is local). Then, the \textit{group} step joins all the constraints on each base variable (such as the constraints (\textit{mix} \ \check{tt} \ \check{tt} B_2) and (\textit{mix} \ \check{tt} \ \check{tt} B_3) on \textit{B_2}) into one constraint (such as the constraint (\textit{mix} \ \check{tt} \ \check{tt} B_3) on \textit{B_2}, expressed as a substitution using \leftarrowrightarrow).

Substituting these conjoined constraints for the base variables in the unknown base \textit{B_1 \check{tt} \ \check{tt} \ \lambda x_1, B_2[x_1]} yields the principal base \textit{mix} \ \check{tt} \ \check{tt} \ \lambda x_3. \textit{Principality} means that the \textit{check} disintegrator finds a kernel with respect to precisely those bases that divide the principal base.

We detail each step of base inference below.

6.3 Introducing base variables

We start by adding variables \textit{B} to our language of bases in Figure 18. The \textit{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 \check{tt} part expresses that the solution is a base measure \textit{with holes}. A base with holes is like a function from terms to bases but cannot, say, distinguish the form of a term that plugs a hole.

Bases  \( b ::= \text{mix} \ell \, e, \ldots, \ell \, | \, \text{return} \, () \, | \, b \odot= \lambda x . b \, | \, b \odot b \, | \, B \)  \( B : B \, R \)

Unknown bases  \( B ::= B[\vec{e}] \, | \, \text{reparam} \, f \, B \)

Base variables  \( B[\] \)

Terms  \( e ::= \ldots \, | \, \text{jacobian} \, f \, B \, e \, | \, \text{divide} \, b \, B[\vec{e}] \, e \)

\[ \text{genBase}(\alpha) : \text{variables} \to \mathbb{B} \, \alpha \]

\[ \text{genBase}(\mathbb{R}) \quad \vec{x} = B[\vec{x}] \]

where \( B \) is fresh

\[ \text{genBase}(\mathbb{1}) \quad \vec{x} = \text{return} \, () \]

\[ \text{genBase}(\alpha \times \beta) \quad \vec{x} = \text{genBase}(\alpha) \, \vec{x} \odot= \lambda y . \text{genBase}(\beta) \, (\vec{x}, y) \quad \text{where} \ y : \alpha \text{ is fresh} \]

\[ \text{genBase}(\alpha + \beta) \quad \vec{x} = \text{genBase}(\alpha) \, \vec{x} \odot \text{genBase}(\beta) \, \vec{x} \]

\[ \text{jacobian} : \text{invertible} \to \mathbb{B} \, \mathbb{R} \to [\mathbb{R}] \to [\mathbb{R}] \quad \text{reparam} : \text{invertible} \to \mathbb{B} \, \mathbb{R} \to \mathbb{B} \, \mathbb{R} \]

\[ \lambda v . (\text{jacobian} \, f \, B \, (f \odot v) \cdot \text{jacobian} \, (\text{inv} \, f) \, b \, v)^{-1} \]

\[ \odot (\text{reparam} \, (\text{inv} \, f) \, b \odot \vec{B}) \, (f \odot v) \]

\[ \lambda v . \text{divide} \, b \, B[\vec{e}] \, v \]

Fig. 21. Introducing base-measure variables and updating the base-checking disintegrator. The first line adds unknown bases \( B \) to the grammar of bases \( b \) and adds a typing rule to say that every unknown base is over \( \mathbb{R} \).

Base variables represent bases with holes because they might occur under a binder \( \lambda x \) to the right of \( \odot= \). A hole in a base is where the bound variable \( x \) is used. To track what variables \( x \) are in scope, we need to augment base variables \( B[\] \) with a sequence of core Hakaru variables \( \vec{x} \). Moreover, the pair case \( \triangleleft \, e \, (b_1 \odot= \lambda x . b_2) \, v \) in Figure 18 requires a sequence of terms \( \vec{e} \), not just variables \( \vec{x} \). In this case, \( \triangleleft \) deconstructs the \( \odot= \) base measure and substitutes \( \text{fst} \, v \) for the bound variable \( x \) in \( b_2 \). Hence, a plugged variable must store terms that may be in scope after such a substitution.

We thus add a new construct, \( B[\vec{e}] \), 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 top of Figure 21 shows the new base language, which combines unknown bases with ground bases.

**Definition 6.6.** A ground base \( b \) is a base measure with no base variables. The (meta) type of a ground base is notated \( \mathbb{B} \, \alpha \). A ground substitution \( \sigma \) is a mapping from base variables (with holes) to ground bases (with holes). Applying a ground substitution \( \sigma \) to a base \( b \) yields a ground base \( \sigma \, b \), by replacing all (plugged) base variables in \( b \) with (plugged) ground bases. For example in Figure 20, the ground base in the lower-left corner is the result of applying the ground substitution in the lower-right corner to the base \( B[\] \odot= \lambda x_1 . B_2[x_1] \) produced by \text{genBase}.

The relation \( \triangleleft \) (Definition 6.2) is only defined on ground bases.

We only need base variables of type \( \mathbb{B} \, \mathbb{R} \), because ground base measures of the same type differ only in their subterms of type \( \mathbb{B} \, \mathbb{R} \). According to the grammar of bases in Figure 18, base measures have a unique structure for each of the other non-measure type constructors: \( \text{return} \, () \) is the only base over \( \mathbb{1} \), bases over pairs are always of the form \( b_1 \odot= \lambda x \, b_2 \), and bases over disjoint sums are always of the form \( b_1 \odot b_2 \).
6.4 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 over $\mathbb{R}$. Its meaning is that the first base must be divisible by the second. By design, the first base is ground while the second is a plugged base variable; i.e., constraints are of the form $(b, B[\bar{\epsilon}])$. 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 may contain base variables. As done above in Section 3.2, we construct the base using $\text{genBase}(\alpha)$. We modify the definition of $\text{genBase}(\alpha)$—as shown in the middle of Figure 21—to construct a base measure $b^*$ of type $\mathbb{B} \alpha$ whose leaf positions of type $\mathbb{B} \mathbb{R}$ are all populated by plugged base variables. For the example in Figure 20, we have $\alpha = \mathbb{R} \times \mathbb{R}$ and we construct $b^* = B_1[\bar{\epsilon}] \otimes \lambda x_1. B_2[x_1]$. The constructed $b^*$ represents an unknown base measure because any ground base $b$ of type $\mathbb{B} \alpha$ can be obtained by applying to $b^*$ a suitable ground substitution $\sigma$ over the base variables. The function $\text{genBase}(\alpha)$ now takes a set of core Hakaru variables $\bar{x}$ as input. This set is initially empty, extended with a fresh core Hakaru variable whenever $\alpha$ is a pair type, and stored alongside a fresh base variable whenever $\alpha$ is $\mathbb{R}$. Thus, we set $b^* = \text{genBase}(\alpha)(\bar{x})$.

Second, we need to run base-checking disintegration with respect to $b^*$. In order for this to work we need to update $\text{jacobian}$, $\text{reparam}$, and $\div$ to handle unknown bases. The necessary updates are shown at the bottom of Figure 21.

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

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

- The divisor is an unknown base of the form $\text{reparam} f \ B$. In this case we use $\text{jacobian}$ to undo the reparameterization of the unknown base $B$, and use $\text{reparam}$ to reparameterize the (ground) dividend by the inverse of $f$. The latter call $\text{reparam} (\text{inv} f) \ B$ always produces a ground base.
- The divisor is a plugged base variable, i.e., an unknown base of the form $B[\bar{\epsilon}]$. In this case we suspend (or residualize) the call using $\text{divide}$.

Thus, running base-checking disintegration with respect to a non-ground base produces a program with embedded residual $\div$ calls of the form $\text{divide} \ b \ B[\bar{\epsilon}] \ e$, where $b$ is a ground base, $B[\bar{\epsilon}]$ is a plugged base variable, and $e$ is a core Hakaru term of type $\mathbb{R}$. We walk through this program and collect the first two arguments of each $\text{divide}$ expression into a set of constraints $(b, B[\bar{\epsilon}])$. In Figure 20 this set is the output of a $\text{collect}$ transformation.

The conjunction of the constraints collected in this step characterizes precisely those ground bases with respect to which $\text{check}$ succeeds, because applying a ground substitution commutes with running $\text{check}$:

$$\text{check} \ m (\sigma \ b^*) \ t = \sigma (\text{check} \ m \ b^* \ t) \quad (53)$$

That is, applying a ground substitution $\sigma$ to produce a ground base $\sigma \ b^*$ then running $\text{check}$ is equivalent to running $\text{check}$ then applying $\sigma$ to resume the disintegration.
6.5 Solving constraints to produce the principal base measure

The final step of base inference produces a ground base $b^*$ that is principal. Like all ground bases of the correct type, $b^*$ shares the structure of the base measure $b^c$ constructed initially by $\text{genBase}$, but with a ground substitution applied to replace all (plugged) base variables with (plugged) ground bases. We compute this ground substitution by solving the constraints collected from base checking while respecting the scope of core Hakaru terms.

From the previous step we obtain a set of constraints of the form $(b, B[\vec{c}])$. This set may contain multiple constraints for the same base variable $B[\cdot]$ (plugged with possibly different term sequences $\vec{c}$).

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. Failure means concluding that base checking cannot succeed with respect to any base measure.

2. **Group** (or **unify**) constraints by their base variable.

3. **Substitute** into $b^*$ the (unified) solutions for each base variable.

6.5.1 Solving a base-variable constraint. A solution of a constraint $(b_1, B[\vec{c}])$ is a ground base replacement $b_2[\cdot]$ for the base variable $B[\cdot]$ such that $b_1 <: b_2[\vec{c}]$. Because the relation $<$ is transitive, there may be infinitely many solutions. We need a **principal solution** $b^*_1[\cdot]$ such that

$$\forall b_2[\cdot], \ b_1 <: b_2[\vec{c}] \iff b^*_1[\vec{x}] <: b_2[\vec{x}].$$

(54)

so in particular $b_1 <: b^*_1[\vec{c}]$. Here $\vec{x}$ are as many fresh variables as there are terms in $\vec{c}$. If $b_1$ contained no open core Hakaru terms, then we could just let $b^*_1[\cdot] = b_1$ and not use any hole. For example in Figure 20, the first two collected constraints $(\text{mix } tt [\cdot], B_1[\cdot])$ and $(\text{mix } tt [\cdot], B_2[\text{fst } t])$ both have the hole-less principal solution $\text{mix } tt [\cdot]$. Using such a hole-less solution is almost what we do.

What we actually do is solve the problem $b^*_1[\vec{c}] = b_2$, a special case of second-order matching [Huet and Lang 1978; Huet 1976], by checking that every core Hakaru variable in $b_2$ occurs inside some subterm of $b_1$ that equals some term in $\vec{c}$. To check this, we replace with a hole every subterm of $b_2$ that the term-equality checker declares equal to some term $e$ in $\vec{c}$. We note the result of this replacement by $b_2[\cdot] \leftrightarrow e$. 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 $b^*_1[\cdot] = b_2[\cdot] \leftrightarrow e$.

For example in Figure 20, to solve the third collected constraint $(\text{mix } ff [\text{fst } t], B_2[\text{fst } t])$, we solve the matching problem $\text{?}[\text{fst } t] = \text{mix } ff [\text{fst } t]$, by replacing $\text{fst } t$ in $\text{mix } ff [\text{fst } t]$ with a hole.

The result of this replacement is $\text{mix } ff [\cdot] = (\text{mix } ff [\text{fst } t])[\cdot] \leftrightarrow \text{fst } t$, which does not contain any free variable, so it is the principal solution for $B_2[\cdot]$. This principal solution is represented in Figure 20 as $(\text{mix } ff [\cdot], B_2[\cdot])$, using a fresh local variable $x_3 : \mathbb{R}$.

Although some second-order matching problems have multiple solutions, ours don’t and we produce a unique principal solution. This is because the base variables in our constraints are plugged with independent atomic terms—such as $\text{fst } t$ and $\text{fst } (\text{snd } t)$, produced by the first case of $<$ in Figure 18—that a sound term-equality checker will always judge to be distinct from each other and from any term not involving the observation variable $t$ passed initially to check.

6.5.2 Grouping constraints by base variables. We are home free and no longer need to worry about failure once we have successfully solved each constraint. The next step is to group the solved constraints $(b^*_1[\vec{x}], B[\vec{x}])$ by their base variables $B[\cdot]$. For this we define a binary operation $b\text{plus}$ that acts as a join in the preorder $<$: by summing together ground base measures of type $B \mathbb{R}$. Since $\text{mix}$ is the only way to construct such bases, $b\text{plus}$ has a one-line definition:

$$b\text{plus} : B \mathbb{R} \to B \mathbb{R} \to B \mathbb{R}$$

$$b\text{plus} (\text{mix } l [e_1, \ldots, \cdot]) (\text{mix } l' [e'_1, \ldots, \cdot]) = \text{mix } (l \lor l') [e_1, \ldots, e'_1, \ldots]$$

(55)
Any two solved constraints \((b^+_1[x_i], B[x_i])\) and \((b^+_2[y_j], B[y_j])\) that share the same base variable \(B[]\) now get grouped into a solved constraint \(B[z] \rightarrow bplus b^+_1[x_i] b^+_2[y_j] x_i \mid z\), where the variables \(z\) are fresh. After grouping, we have one solved constraint per base variable. (The odd unconstrained base variable \(B[]\) gets the least solution \(B[z] \mapsto \text{mix ff } \{\}\).)

For example in Figure 20, the two solved constraints \((\text{mix tt } \{x_1\}, B_2[x_2])\) and \((\text{mix ff } \{x_3\}, B_2[x_3])\) get grouped into \(B_2[x_4] \mapsto \text{mix tt } \{x_4\}\), because \(bplus\) joins \(\text{mix tt } \{\}\) and \(\text{mix ff } \{x_4\}\) into \(\text{mix tt } \{x_4\}\). Again \(x_2, x_3, x_4\) are just local variables used to represent and identify holes in unplugged bases.

### 6.5.3 Substituting solutions to form a principal base.

At this point we have one ground instantiation \(b^*_1\) for each base variable \(B_1\). We obtain our principal base measure \(b^*\) by substituting these ground bases into \(b^*\):

\[
b^* = b^*(B_1[]) \mapsto b^*_1[], \ldots
\]

In Figure 20 we produce the base \(\text{mix tt } \{x_5\} \mapsto \lambda x_5. \text{mix tt } \{x_5\}\).

### 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 18. 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 \(\zeta \Rightarrow \xi\) with respect to \(\xi \Rightarrow \zeta\) [Tierney 1998], where \(\xi\) and \(\zeta\) are specified as probabilistic programs.

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

\[
disint : [\text{M} (\alpha \times \beta)] \rightarrow [\text{M} \alpha] \rightarrow [\alpha] \rightarrow \{[\text{M} \beta]\}
\]

\[
disint m_1 \_ m_2 t = |\text{check } m_2 b t|^{-1} \odot \text{check } m_1 b t \quad \text{where } b = \text{infer } m_2
\]

That is, we use base inference to find the intermediate base measure \(\mu\) in Proposition 2.7(1). One caveat of this approach is that \(\text{disint}\) takes the reciprocal of the density \(\text{check } m_2 b t\) and thus assumes that the density is almost never 0 or \(\infty\). It would be nice to check this assumption algorithmically, but we leave that for future work.

### 8 EVALUATION AND FUTURE WORK

Because we handle distributions whose disintegration had never been automated before, there is not yet a corpus of programs written in the wild on which to evaluate the completeness of our disintegrator in practice. Nevertheless, we can report that 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 \(\text{mix tt}\) 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.

Item 5 above, Metropolis-Hastings sampling, offers a good look at the boundary of what our disintegrator can and cannot handle. Let \(\xi_R : \text{M } \mathbb{R}\) and \(\xi_{R^2} : \text{M } \mathbb{R}^2\) be some continuous distributions over \(\mathbb{R}\) and \(\mathbb{R}^2\). (The precise densities do not matter so long as they are expressible; for instance,
they can be \( \lambda x. \exp(0.4(x - 0.4)^2 - 0.08x^4) \) in (12) and \( \kappa \) in (17).) Following equation (19) for the target \( \xi_{\mathbb{R}^2} \), we can construct the single-site proposal kernel
\[
\xi_{\mathbb{R}^2} = \lambda (x_1, x_2), \quad \frac{1}{2} \odot \{ x'_1 \sim \text{normal} x_1 0.1; \text{return} (x'_1, x_2) \} \\
\odot \frac{1}{2} \odot \{ x'_2 \sim \text{normal} x_2 0.1; \text{return} (x_1, x'_2) \}. \tag{58}
\]
which flips a fair coin to decide whether to perturb \( x_1 \) and keep \( x_2 \) or to perturb \( x_2 \) and keep \( x_1 \), And for the target \( \xi_{\mathbb{R}^2} \otimes \xi_{\mathbb{R}^d} : \mathbb{M}(\mathbb{R} + \mathbb{R}^d) \), we can construct the reversible-jump proposal kernel
\[
\xi_{\mathbb{R}^2 + \mathbb{R}^d} = \lambda x. \text{case } x \text{ of } \text{inl} x_0 \rightarrow \frac{1}{2} \odot \text{fmap inl} (\text{normal} x_0 0.1) \\
\odot \frac{1}{2} \odot \text{fmap inl} (\text{normal} x_0 0.1 \otimes \text{normal} x_0 0.1) \tag{59}
\]
\[
\text{inr} (x_1, x_2) \rightarrow \frac{1}{2} \odot \text{fmap inl} (\text{normal} \frac{x_1 + x_2}{2} 0.1) \\
\odot \frac{1}{2} \odot \text{fmap inr} (\xi_{\mathbb{R}^d}(x_1, x_2)).
\]
which flips a fair coin to decide whether to stay in the same component of the sum type \( \mathbb{R} + \mathbb{R}^d \) or to jump to the other component. For \( \alpha \in \{\mathbb{R}^2, \mathbb{R} + \mathbb{R}^d\} \) and \( \beta = 1 \), our unrestricted base-checking disintegrator (Section 7) successfully finds the density of \( \zeta_\alpha \Rightarrow \xi_\alpha \) with respect to \( \xi_\alpha \otimes - = \xi_\alpha \), in other words, the Metropolis-Hastings acceptance ratio for the target \( \xi_\alpha \) and the proposal \( \zeta_\alpha \).

Unfortunately, our disintegrator returns no result if we change the reversible-jump proposal (59) so that it sometimes jumps without adding noise—for instance if we change \( \text{normal} \frac{x_1 + x_2}{2} 0.1 \) on the third line to \( \text{return} \frac{x_1 + x_2}{2} \). This failure is because the notion of term equality and matching we implemented in Figure 18 and Section 6.5.1 (“known to be equal”) is mere \( \alpha \)-equivalence and is not modulo \( \beta \)-equivalence for sum types. 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 \( \Pi \) (n-ary products, where \( n \) is a symbolic array size) [Buntine 1994] 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.

9 RELATED WORK

Our work shows how to compute densities and disintegrations, exactly and symbolically, for a broader variety of base measures than before. It thus contrasts with previous works that perform simplification instead, that compute approximations instead, and that fix the base measure instead.

**Performing simplification** Whereas disintegration seeks any program whose denoted measure differs from the given program in accordance with a semantic specification, simplification seeks a program whose denoted measure is same as the given program but whose efficiency or readability is improved. Our work is thus complementary to the work on simplification by Carette and Shan [2016] and Gehr et al. [2016]; the result of disintegration can be improved by simplification while preserving correctness, and it may also be possible to ease disintegration by first simplifying its input.

**Computing approximations** Many probabilistic programming systems can be said to compute densities or disintegrations in the form of an approximation such as a stream of density estimates [Pfeffer 2009] or of posterior samples [Lunn et al. 2000; Carpenter et al. 2017; Goodman et al. 2008; Wingate et al. 2011; Wood et al. 2014; Wu et al. 2018]. But because those computations only take programs as input and do not produce programs as output, they cannot be composed with other program transformations (such as simplification as just discussed) or used as part of a larger application or compiler pipeline (such as to generate desired samplers or plots) in a modular way. In particular, although Wu et al.’s work on
mixtures [2018] shares motivation with us such as the GPA problem (Example 2.8), their lexicographic inference algorithms produce weighted samples and so do not compose and do not allow specifying a custom proposal distribution.

**Fixing the base measure** Previous program transformations that compute densities and disintegrations can be classified by the base measures they allow. Mohammed Ismail and Shan’s density calculator [2016] and Shan and Ramsey’s disintegrator [2017] only deal explicitly with \( \text{lebesgue} \), the Lebesgue measure over \( \mathbb{R} \). Bhat et al.’s density calculator [2012, 2013] allows \( \text{lebesgue} \odot \cdots \odot \text{lebesgue} = \text{lebesgue}^n \), the Lebesgue measure over \( \mathbb{R}^n \) where \( n \) is a concrete natural number; their *stock measure* is our initial \( \text{genBase} \) function (31). Most recently, Narayanan and Shan’s disintegrator [2017] handles \( \text{lebesgue}^n \) for symbolic \( n \). Our base-measure language (Figure 18) includes discrete-continuous mixtures over \( \mathbb{R} \), dependent products, and disjoint sums; we also allow specifying the base measure as another probabilistic program (Section 7). Thus, our work subsumes all but Narayanan and Shan’s, and is the first to allow different base measures over the same type.

We prove our disintegrator sound using equational reasoning on probabilistic programs (Section 4.1), which is familiar from functional programming [Bird and de Moor 1996; Hughes 1995; Hutton and Meijer 1996], and using induction on step indices (Section 4.4), which is familiar from domain theory. Like us, Shan and Ramsey [2017] also use equational reasoning to argue for the soundness of their disintegrator, but we give more detail (Section 4.4) for a refactored proof (Section 4.2) about a more general disintegrator (Section 5). Besides disintegration, equational reasoning has been used to prove other properties of probabilistic programs [Sato et al. 2019], including inference correctness [Ścibior et al. 2018]. Our soundness result plugs into those proofs; for example, it discharges the density preconditions in Ścibior et al.’s theorems about the correctness of Metropolis-Hastings sampling.

The foundation of our equational reasoning is the semantics of core Hakaru. It is based on the probability monad [Giry 1982; Ramsey and Pfeffer 2002]. However, because disintegration requires scoring by an unbounded factor, we must consider measures that are not probability distributions, and we cannot restrict ourselves to sub-probability distributions as Borgström et al. do [2016]. Rather, in Section 3.3.1 we adopt Staton’s s-finite semantics [2017]. This semantics has been extended to richer languages with higher-order recursive terms and types [Heunen et al. 2017; Ścibior et al. 2018; Vákár et al. 2019]. The logical relations of Culpepper and Cobb [2017] and Wand et al. [2018] also support reasoning about contextual equivalence of probabilistic programs with unbounded scoring.

A density calculator and disintegrator, like ours, can be regarded as a programming analogue of Radon-Nikodym theorems, which assert the existence of a density [Nikodym 1930], and of disintegration theorems, which assert the existence of a disintegration [Dieudonné 1948; Chang and Pollard 1997]. In particular, Vákár and Ong [2018] recently proved Radon-Nikodym and disintegration theorems with respect to s-finite kernels. Our disintegrator falls under a simple special case of those theorems, because it handles s-finite measures with respect to \( \sigma \)-finite base measures. However, it is not structured like the proof of any existence theorem, and it is unfortunately not complete: there are easy ways to stymie it intentionally (such as \( \triangleright (x + x) \)), and we demonstrate its utility only empirically (Section 8). This incompleteness is not explained by Ackerman et al.’s result [2011, 2016] that a computable measure can have disintegrations that are all uncomputable, because our input language does not express all computable measures. All the statements about our disintegrator in this paragraph apply to previous density calculators and disintegrators as well.

Our disintegrator uses continuations in three ways: to maintain the heap for lazy partial evaluation [Jørgensen 1992; Fischer et al. 2008], to deal separately with mixture components [Danvy and
Symbolic disintegration with a variety of base measures

Filinski 1990], and to emit bindings in the output code [Bondorf 1992; Lawall and Danvy 1994]. Continuations can also be used to backtrack among nondeterminism possibilities [Tennent 1973], but we manage nondeterminism using sets instead for clarity.

Our notion of a principal base measure is new (Definition 6.5), but our algorithm to find it is clearly inspired by constraint-based type inference [Wand 1987a,b; Pierce 2002, chapter 22].

ACKNOWLEDGMENTS

This research was supported by DARPA contract FA8750-14-2-0007. We thank our anonymous reviewers at POPL 2019.

REFERENCES


Symbolic disintegration with a variety of base measures


