# 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 probabilistic programs that generalizes density calculation and conditioning, two operations widely used for exact and approximate 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 with 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 calculation 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:**
Praveen Narayanan and Chung-chieh Shan. 2019. Symbolic disintegration with a variety of base measures. *ACM Trans. Program. Lang. Syst.* 1, 1 (February 2019), 59 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 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 calculation*, *conditioning*, *marginalization*, and *expectation*. Probabilistic programming represents distributions as programs and mechanizes these operations, whether exactly or approximately, through symbolic and numerical computations.

Density calculation and conditioning are two important operations on distributions, used to define both inference problems and approximate solutions. For example, densities are used to define *maximum likelihood estimates* and *Metropolis-Hastings samplers*, whereas conditional distributions are used to define *posterior belief updates* and *Gibbs samplers*. In fact, density calculation and conditioning are special cases of *disintegration*. Recent years have seen the exact automation of both density calculation [Bhat et al. 2012, 2013; Mohammed Ismail and Shan 2016; Cusumano-Towner et al. 2019; Roberts et al. 2019] 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 enable the sampling user to specify a proposal distribution as a probabilistic program. Further, symbolic computation enables the user to specify an observation or proposal by applying deterministic operations such as square root and addition to the outcome of random choices. 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; Walia et al. 2019].

2019. 0164-0925/2019/2-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 calculation, 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* [Bhat et al. 2012], an independent product of Lebesgue and counting measures. But as illustrated in Sections 1.2 and 2, this assumption fails 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.

### 1.1 Contributions and organization

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

(B1) mixtures of the Lebesgue measure and point masses,
(B2) dependent products, and
(B3) 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 reparametrizations of, mixture distributions.

In Section 2, we specify disintegration as a measure-preserving program transformation and motivate variable-base disintegration using instances of density calculation and conditioning. In Section 3, we define a probabilistic language and illustrate the reasoning we automate. In Sections 4 to 7, we then present our variable-base disintegrator by progressively defining four measure-preserving program transformations.

- In Section 4, we refactor Shan and Ramsey's original disintegrator [2017] to expose and isolate its use of a base measure that is uniquely determined by its type: the Lebesgue base measure, an independent product, or a disjoint sum (B3). To this end, we semantically specify and equationally derive a set of meta-functions that constrain expressions, invert and differentiate numeric operations, and reparametrize and divide base measures.
- In Section 5, we let the base measure vary in two ways: we generalize the Lebesgue base measure to mixtures of the Lebesgue measure and point masses (B1), and we generalize independent products to dependent products (B2).
- In Section 6, we automate inferring an unknown base measure, by 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 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*.
- In Section 7, we allow the user to give a base measure as a probabilistic program, in the same language as the measure to disintegrate.

In Section 8, we return to our motivating applications and confirm empirically that they are handled by our new disintegrator. Sections 9 and 10 situate our contributions in related and future work.

### 1.2 An initial sketch of our problem and approach

To give a taste of our work, we informally discuss a Bayesian inference problem that Wu et al. [2018] devised to motivate concern for mixture distributions, one of our many applications. Suppose we observe the grade point average (GPA) of a student and want to guess the nationality of their institution. Our probabilistic model of how the nationality affects the grade might go as follows. A priori, the student may have studied in India or in the US. An Indian institution awards a GPA by drawing a random real number uniformly between 0 and 10, except 5% of the time the student achieves the perfect 10. Similarly, a US institution awards a GPA by drawing a random real number uniformly between 0 and 4, except 5% of the time the student achieves the perfect 4.

Observing different GPAs helps us infer the unknown nationality. For example, observing the GPA 7 tells us for sure that the nationality is India, because 7 is too high for the US. On the other hand, observing the GPA 3 suggests that the nationality is US, because the US interval $[0, 4]$ is more concentrated (in other words, less spread out) than the India interval $[0, 10]$. At first glance, these guesses might seem warranted by the ratio of the two "weight functions" graphed in Figure 1.

A weight is a non-negative quantity, as in a weighted average. Figure 1 graphs two functions from GPAs to weights. Graphed at the top is a function one might naïvely define to model GPAs in India; it is a $95\% : 5\%$ mixture of a uniform distribution over $[0, 10]$ and a deterministic distribution at 10. Graphed at the bottom is a similar function one might define to model GPAs in the US; it is a $95\% : 5\%$ mixture of a uniform distribution over $[0, 4]$ and a deterministic distribution at 4. Evaluated at 7, the India function returns a nonzero weight whereas the US function returns zero, which matches the India conclusion. And evaluating the functions at 3 yields the greater weight $95\%/4$ in the US than $95\%/10$ in India, which matches the US suggestion.

So far, so good. But what if we observe the GPA 4 exactly? We should conclude the US almost surely, because the probability of US GPAs "bunches up" at 4: out of 1000 US GPAs, approximately 50 (that is, 5%) would be exactly 4, whereas out of 1000 India GPAs, none would be exactly 4, even though some might be close to 4. Unfortunately, evaluating the functions in Figure 1 at 4 yields the greater weight $95\%/10$ in India than 5% in the US, which suggests the wrong guess!

What went wrong in the naïve probability weight functions in Figure 1? Intuitively, the "unit" of the weights $95\%/10$ and $95\%/4$ is different from the "unit" of the weight 5%; the former is "continuous" whereas the latter is "discrete". When evaluated at the GPA 4, the India function returns a continuous weight whereas the US function returns a discrete weight. To avoid the error of comparing weights of different units, we need to keep track of whether a weight is continuous or discrete at each point along the real line. That is what a *base measure* over the reals does. To compare two weight functions, they must be *densities* with respect to a *common* base measure.

To solve this GPA problem, then, we need a base measure such that both the India GPA distribution and the US GPA distribution have densities with respect to it. The Lebesgue measure, which is continuous everywhere and commonly assumed in prior work, is not suitable because neither distribution has a density with respect to it. Instead, one suitable base measure is discrete at both 4 and 10 and continuous everywhere else (B1). With respect to this base measure, however, the weight function at the top of Figure 1 is not quite a density of the India GPA distribution: we need to revise the weight at 4 from $95\%/10$ to 0, as graphed in Figure 2. Comparing the revised weights at 4 then gives the correct conclusion, that the nationality is US almost surely.

Given descriptions of the two GPA distributions as probabilistic programs, our disintegrator infers the common base measure just described—as a *principal* base measure, in fact. It then finds the densities and thus automates solving this Bayesian inference problem exactly.

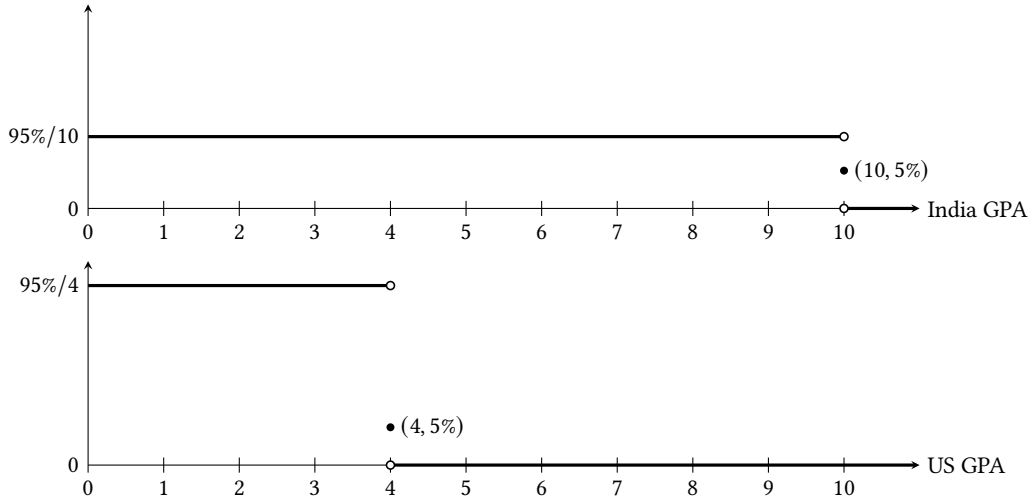


Fig. 1. Naïve probability weight functions that model GPAs in India (top) and in the US (bottom). A hollow circle indicates a point that does not lie in the graph of the function, whereas a solid circle indicates a point that does.

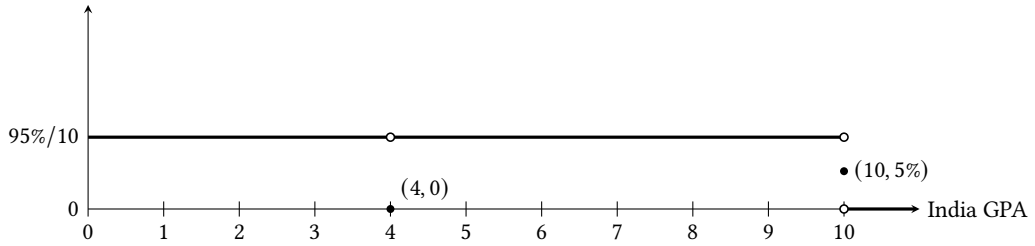


Fig. 2. Revised probability weight function that models GPAs in India and can be compared against the function for the US at the bottom of Figure 1.

## 2 BACKGROUND

We introduce disintegration as a measure-theoretic relation and probabilistic-program transformation, by generalizing from density calculation 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

To explain why it is useful to find densities, we need to first define what a density is. And to define what a density is, we need to first explain how a weight function differs from a measure. A weight function over a type $\alpha$ is a function that maps each $\alpha$-value to its weight. We write the type of such a function as

$$\alpha \to \mathbb{R}_+, \tag{1}$$

where the type of weights $\mathbb{R}_+$ consists of the non-negative real numbers and $\infty$. Weights cannot be negative. For example, the bell curve graphed in Figure 3 is a weight function over $\mathbb{R}$. In contrast, a

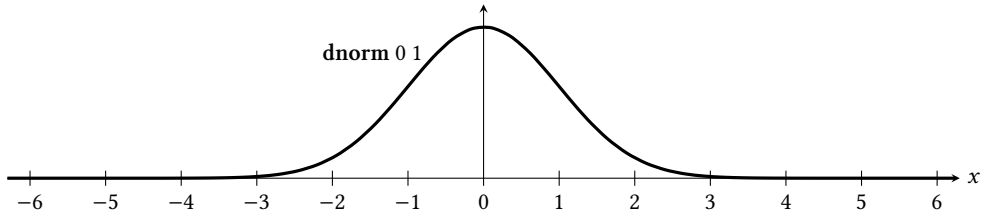


Fig. 3. A density of the normal distribution **normal** 0 1 with respect to the Lebesgue measure

measure over $\alpha$ can be defined as a function that maps an $\alpha$-set to its size. We could write the type of such a measure as

$$\mathbb{M}\,\alpha \quad = \quad (\alpha \to \mathit{bool}) \to \mathbb{R}_+, \tag{2}$$

where the type constructor $\mathbb{M}$ stands for "measure", and a function from $\alpha$ to *bool* is an indicator that represents an $\alpha$-set by its membership predicate. Instead of (2) however, in this paper we use another definition of measures that is equivalent but more convenient [de Finetti 1970, Sections 2.11 and 3.8; de Finetti 1972, Section 6.3; Pollard 2001, Section 1.4]: we define a measure over $\alpha$ as a function that maps an $\alpha \to \mathbb{R}_+$ function to its integral. That is, we define a measure as an *integrator*:

$$\mathbb{M}\,\alpha \quad = \quad (\alpha \to \mathbb{R}_+) \to \mathbb{R}_+. \tag{3}$$

The integrated function must be non-negative.

(More precisely, $\alpha$ and $\mathbb{M}\,\alpha$ are *measurable spaces*, and a measure over $\alpha$ is a certain kind of function that maps *measurable* $\alpha$-to-$\mathbb{R}_+$ functions to elements of $\mathbb{R}_+$. Every integrator (3) can be used as a set-measurer (2) because a *measurable* $\alpha$-set can be regarded as a measurable function from $\alpha$ to the binary weights $\{0, 1\} \subset \mathbb{R}_+$. Conversely, every set-measurer (2) can be used as an integrator (3) by the standard construction of the *Lebesgue integral* [Royden 1988, Chapters 4 and 11], which intuitively approximates an area under a curve by slicing it into horizontal strips.)

Weight functions (1) and measures (3) are different, even though they are often conflated in the popular imagination. In fact, measures are sometimes called *generalized* functions.

*Example 2.1.* The standard normal distribution is written **normal** 0 1 in core Hakaru, because its mean is 0 and its standard deviation is 1. Whereas the bell curve in Figure 3 is the weight function

$$\mathbf{dnorm}\ 0\ 1 \quad = \quad \lambda x.\, \exp(-x^2/2)/\sqrt{2\pi} \quad : \quad \mathbb{R} \to \mathbb{R}_+, \tag{4}$$

the normal distribution is the measure

$$\mathbf{normal}\ 0\ 1 \quad = \quad \lambda f.\, \textstyle\int_{\mathbb{R}} (\mathbf{dnorm}\ 0\ 1)(x) \cdot f(x)\, dx \quad : \quad \mathbb{M}\,\mathbb{R}. \tag{5}$$

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

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

$$\mathbf{lebesgue} \quad = \quad \lambda f.\, \textstyle\int_{\mathbb{R}} f(x)\, dx \quad : \quad \mathbb{M}\,\mathbb{R}. \tag{6}$$

This integrator maps any (measurable) function $f : \mathbb{R} \to \mathbb{R}_+$, which must be non-negative, to the area under it above the $x$-axis. For example, if $f$ maps positive reals $x$ to $(\mathbf{dnorm}\ 0\ 1)(x)$ and other reals to 0, then $\mathbf{lebesgue}(f) = 1/2$, because the area under the bell curve **dnorm** 0 1, above the $x$-axis and to the right of the $y$-axis, is $1/2$.

*Definition 2.2.* The *total* $|\mu|$ of a measure $\mu$ is its integral of the constant-1 function. That is, $|\mu| = \mu(\lambda\_.\, 1)$. We say $\mu$ is *finite* or *infinite* if $|\mu|$ is. For example, $|\mathbf{normal}\ 0\ 1| = \int_{\mathbb{R}} (\mathbf{dnorm}\ 0\ 1)(x)\, dx = 1$. Such a finite measure, whose total is 1, is called a *probability distribution* and *normalized*. In contrast, $|\mathbf{lebesgue}| = \int_{\mathbb{R}} 1\, dx = \infty$, so **lebesgue** is infinite and thus *unnormalized*.

A density is a weight function that characterizes one measure with respect to another measure. For example, the bell curve **dnorm** 0 1 is a density of the measure **normal** 0 1 with respect to the measure **lebesgue**, because

$$\mathbf{normal}\ 0\ 1 \quad = \quad \lambda f.\, \mathbf{lebesgue}\big(\lambda x.\, (\mathbf{dnorm}\ 0\ 1)(x) \cdot f(x)\big). \tag{7}$$

Checking this equation is a matter of $\beta$-reduction. In core Hakaru, the right-hand side is expressed as follows:

$$\mathbf{normal}\ 0\ 1 \quad = \quad \mathbf{do}\ \{x \leftsquigarrow \mathbf{lebesgue};\ (\mathbf{dnorm}\ 0\ 1)(x) \odot \mathbf{return}\ x\}. \tag{8}$$

The core Hakaru forms **do** $\{x \leftsquigarrow \cdots; \cdots\}$ and **return** denote bind and unit in the measure monad [Giry 1982; Ramsey and Pfeffer 2002].

- The unit operation **return** turns any value $x$, of type $\alpha$ (above $\mathbb{R}$), into the probability distribution concentrated at $x$, of type $\mathbb{M}\,\alpha$. Another name for **return**$x$ is the *Dirac distribution* or *deterministic distribution* at $x$. It denotes the integrator $\lambda f.\, f(x)$.
- The bind operation **do** $\{x \leftsquigarrow \mu;\ \nu(x)\}$ integrates $\nu : \alpha \to \mathbb{M}\,\beta$ (a family of measures over $\beta$ parameterized by $x : \alpha$) with respect to $\mu : \mathbb{M}\,\alpha$ (a measure over $\alpha$) to form a measure over $\beta$ (no longer parameterized by $x : \alpha$). This *Kock integral* [Kock 2012; Ścibior et al. 2018] applies not to weights but to measures over $\beta$. It denotes the integrator $\lambda f.\, \mu\big(\lambda x.\, \nu(x)(f)\big)$.

The monadic intuition behind these operations is that unit lets us treat a deterministic value as a trivially nondeterministic computation, and bind lets us treat a sequence of two nondeterministic computations as one. Finally, the $\odot$ form in (8) scales the measure **return** $x$ by the weight $(\mathbf{dnorm}\ 0\ 1)(x)$. So, roughly (8) says that the normal distribution can be obtained by scaling the **lebesgue** measure at each real $x$ by the weight $(\mathbf{dnorm}\ 0\ 1)(x)$, while leaving the value $x$ intact.

The definition of a density simply generalizes equations (7) and (8).

*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

$$\xi \quad = \quad \lambda f.\, \mu\big(\lambda x.\, \kappa(x) \cdot f(x)\big), \tag{9}$$

or in core Hakaru,

$$\xi \quad = \quad \mathbf{do}\ \{x \leftsquigarrow \mu;\ \kappa(x) \odot \mathbf{return}\ x\}. \tag{10}$$

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

As this definition makes clear, the notion of a density is a ternary relation between a weight function $\kappa$ and two measures $\xi$ and $\mu$. Implemented as a program transformation, this relation can take on a variety of directions (modes). In one direction, given any core Hakaru expressions for $\mu$ and $\kappa$, it is trivial to express a measure $\xi$ in core Hakaru such that $\kappa$ is a density of $\xi$ with respect to $\mu$: just use equation (10). A harder direction is to turn core Hakaru expressions for $\xi$ and $\mu$ into $\kappa$: the answer is neither unique nor guaranteed to exist, as Examples 2.4 to 2.6 below show. This latter direction is called *density calculation* and the concern of this paper.

It is popular to set $\mu =$ **lebesgue**: 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 default *stock measure* [Bhat et al. 2012] 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.

Some measures are continuous, and some are not.

*Example 2.4.* The standard normal distribution **normal** 0 1 is continuous; after all, it has the density **dnorm** 0 1 with respect to the Lebesgue measure. In fact, this density is not unique: the weight functions

$$\lambda x. \begin{cases} 12 & \text{if } x = 34 \\ 56 & \text{if } x = 78 \\ \textbf{dnorm}\ 0\ 1\ x & \text{otherwise} \end{cases} \qquad \text{and} \qquad \lambda x. \begin{cases} 0 & \text{if } x \in \mathbb{Z} \\ \infty & \text{if } x \in \mathbb{Q} \setminus \mathbb{Z} \\ \textbf{dnorm}\ 0\ 1\ x & \text{otherwise} \end{cases} \tag{11}$$

are just two other densities of the standard normal distribution with respect to the Lebesgue measure. In general, given $\xi$ and $\mu$, the density $\kappa$ is only unique up to *$\mu$-almost-sure equivalence*: we can revise $\kappa$ at any set of points that is *negligible* (that is, has size 0 according to $\mu$).

*Example 2.5.* The measure **return** 42 : $\mathbb{M}\,\mathbb{R}$ is not continuous: If it were, then equation (9) would yield $\lambda f.\, f(42) = \lambda f.\, \int_{\mathbb{R}} \kappa(x) \cdot f(x)\, dx$. When we take the integrators on both sides and apply them to the function $f$ that maps 42 to 37 and all other reals to 0, we get $37 = 0$, a contradiction.

*Example 2.6.* Non-continuous measures often arise from *clamping* continuous measures. Clamping means replacing out-of-bounds outcomes by the bound. For example, a sensor that can measure only reals between 0 and 1 (like a camera pixel) might sense reals less than 0 as 0 and reals greater than 1 as 1. Clamping **normal** 0 1 in this way yields a measure whose outcome is exactly 0 half of the time and exactly 1 roughly 16% of the time. More precisely, as an integrator it is

$$\begin{aligned} \xi &= \lambda f.\, \textstyle\int_{\mathbb{R}} (\textbf{dnorm}\ 0\ 1)(x) \cdot f(\max\{0, \min\{1, x\}\})\, dx \\ &= \lambda f.\, \textstyle\int_{-\infty}^{0} (\textbf{dnorm}\ 0\ 1)(x)\, dx \cdot f(0) \\ &\qquad + \textstyle\int_{0}^{1} (\textbf{dnorm}\ 0\ 1)(x) \cdot f(x)\, dx \\ &\qquad + \textstyle\int_{1}^{+\infty} (\textbf{dnorm}\ 0\ 1)(x)\, dx \cdot f(1), \end{aligned} \tag{12}$$

and core Hakaru can express it as

$$\begin{aligned} \xi &= \textbf{do}\ \{x \leftsquigarrow \textbf{normal}\ 0\ 1;\ \textbf{return}\ \max\{0, \min\{1, x\}\}\} \\ &= \textbf{do}\ \{x \leftsquigarrow \textbf{normal}\ 0\ 1;\ \textbf{if}\ \ x < 0\ \textbf{then return}\ 0\ \textbf{else fail}\} \\ &\quad \oplus\ \textbf{do}\ \{x \leftsquigarrow \textbf{normal}\ 0\ 1;\ \textbf{if}\ 0 \le x \le 1\ \textbf{then return}\ x\ \textbf{else fail}\} \\ &\quad \oplus\ \textbf{do}\ \{x \leftsquigarrow \textbf{normal}\ 0\ 1;\ \textbf{if}\ 1 < x\ \ \textbf{then return}\ 1\ \textbf{else fail}\}. \end{aligned} \tag{13}$$

In equation (13), the associative binary operator $\oplus$ denotes summing (*mixing*) measures of the same type. Its identity **fail** denotes the zero measure. The totals of the three summands are 1/2, roughly 34%, and roughly 16%. The same measure is denoted whether $\le$ or $<$ is used.

Arguments similar to that in Example 2.5 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

$$\mu \;=\; \textbf{return}\ 0 \oplus \textbf{lebesgue} \oplus \textbf{return}\ 1 \;=\; \lambda f.\, f(0) + \textstyle\int_{\mathbb{R}} f(x)\, dx + f(1), \tag{14}$$

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

$$\begin{aligned} \kappa(0) &= \textstyle\int_{-\infty}^{0} (\mathbf{dnorm}\ 0\ 1)(x)\, dx, \\ \kappa(x) &= (\mathbf{dnorm}\ 0\ 1)(x) && \text{if } 0 < x < 1, \\ \kappa(1) &= \textstyle\int_{1}^{+\infty} (\mathbf{dnorm}\ 0\ 1)(x)\, dx, \\ \kappa(x) &= 0 && \text{if } x < 0 \text{ or } 1 < x. \end{aligned} \tag{15}$$

Checking that this $\kappa$ is indeed a density is a matter of plugging equations (12), (14), and (15) into equation (9) and using the fact that $\int_{\mathbb{R}} \kappa(x) \cdot f(x)\, dx = \int_0^1 \kappa(x) \cdot f(x)\, dx$ because $\kappa(x) = 0$ whenever $x < 0$ or $1 < x$.

The state of the art in mechanizing density calculation on probabilistic programs [Bhat et al. 2012, 2013; Mohammed Ismail and Shan 2016; Shan and Ramsey 2017; Narayanan and Shan 2017; Cusumano-Towner et al. 2019; Roberts et al. 2019] is limited to the stock base measure. Thus, our disintegrator is the first transformation that can turn $\xi$ in (13) into $\kappa$ in (15) with respect to $\mu$ in (14), a mixture of the Lebesgue measure and point masses (B1).

*Remark* 2.7. The measures over each space form a monoid whose binary operation is $\oplus$ and whose identity is **fail**. This fact can be used to equate the two probabilistic programs in (13). First we reason equationally about the subexpressions under the scope of $x$:

$$\begin{aligned} & \mathbf{return}\ \max\{0, \min\{1, x\}\} \\ = \quad & \{\text{semantics of max and min}\} \\ & \mathbf{if}\ x < 0\ \mathbf{then\ return}\ 0\ \mathbf{else\ if}\ x \le 1\ \mathbf{then\ return}\ x\ \mathbf{else\ return}\ 1 \\ = \quad & \{\mathbf{fail} \text{ is identity of } \oplus\} \\ & \mathbf{if}\ x < 0\ \mathbf{then}\ (\mathbf{return}\ 0 \oplus \mathbf{fail} \oplus \mathbf{fail})\ \mathbf{else} \\ & \mathbf{if}\ x \le 1\ \mathbf{then}\ (\mathbf{fail} \oplus \mathbf{return}\ x \oplus \mathbf{fail})\ \mathbf{else} \\ & \qquad\qquad\qquad (\mathbf{fail} \oplus \mathbf{fail} \oplus \mathbf{return}\ 1) \\ = \quad & \{\mathbf{if} \text{ distributivity; the logic of real inequalities}\} \\ & \quad \mathbf{if}\ \ x < 0\ \mathbf{then\ return}\ 0\ \mathbf{else\ fail} \\ & \oplus\ \mathbf{if}\ 0 \le x \le 1\ \mathbf{then\ return}\ x\ \mathbf{else\ fail} \\ & \oplus\ \mathbf{if}\ 1 < x\ \ \mathbf{then\ return}\ 1\ \mathbf{else\ fail}. \end{aligned} \tag{16}$$

Then we use the fact that monadic bind distributes over $\oplus$ (and **fail**):

$$\mathbf{do}\ \{x \leftsquigarrow \mu;\ \mu_1 \oplus \mu_2\} = \mathbf{do}\ \{x \leftsquigarrow \mu;\ \mu_1\} \oplus \mathbf{do}\ \{x \leftsquigarrow \mu;\ \mu_2\}, \qquad \mathbf{do}\ \{x \leftsquigarrow \mu;\ \mathbf{fail}\} = \mathbf{fail}, \tag{17}$$

$$\mathbf{do}\ \{x \leftsquigarrow \mu_1 \oplus \mu_2;\ \mu\, x\} = \mathbf{do}\ \{x \leftsquigarrow \mu_1;\ \mu\, x\} \oplus \mathbf{do}\ \{x \leftsquigarrow \mu_2;\ \mu\, x\}, \quad \mathbf{do}\ \{x \leftsquigarrow \mathbf{fail};\ \mu\, x\} = \mathbf{fail}. \tag{18}$$

Left distributivity (17) follows from right distributivity (18) and commutativity (Section 3.2.1). Right distributivity (18) characterizes this monoid as an *algebraic effect* [Plotkin and Power 2003].

## 2.2 Using densities for inference

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

*Example 2.8.* Suppose we observe a real $x$ drawn randomly from a black box, and we wonder whether the black box is **normal** 0 1 or **normal** 3 2. We would like to compare the likelihood of drawing the observed real from each of the two normal distributions. Unfortunately, the probability of drawing any real from any normal distribution is zero, and comparing zero against zero does not help us adjudicate between the two hypotheses.

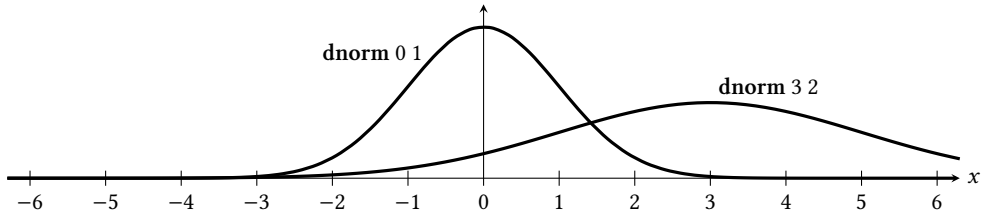


Fig. 4. Densities of two normal distributions, **normal** 0 1 and **normal** 3 2, with respect to the Lebesgue measure

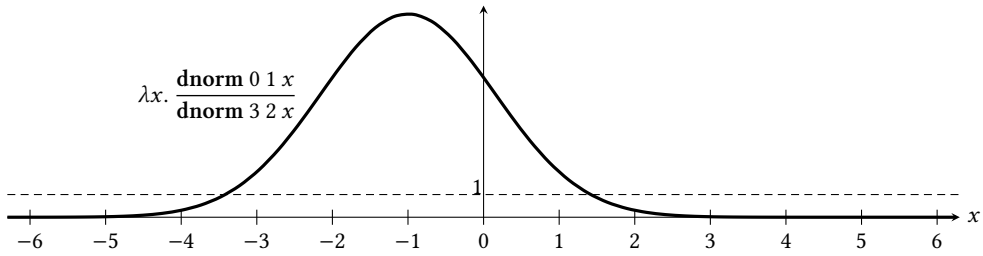


Fig. 5. A density of **normal** 0 1 with respect to **normal** 3 2, the ratio of the two densities in Figure 4

Instead, we can compare the densities of the two normal distributions with respect to the Lebesgue measure, at the observed real. Figure 4 plots those densities. As the plot shows, if we observe the real 1, then we should favor the hypothesis **normal** 0 1 because (**dnorm** 0 1)(1) > (**dnorm** 3 2)(1), whereas if we observe the real 2, then we should favor the hypothesis **normal** 3 2, because (**dnorm** 0 1)(2) < (**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 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 base measure, as long as both densities exist. For example, if we double the Lebesgue measure (2 ⊙ **lebesgue**) as the base measure, then the two densities would each be halved, but their ratio would remain the same. In fact, we can just pick **normal** 3 2 as the base measure, and take advantage of the following fact:

Proposition 2.9. *The constant-*1 *function is a density of every measure with respect to itself. (Thus, the existence of a density is a reflexive relation among measures.)*

Proof. Let $\mu$ be the measure. We use the fact that scaling a measure by 1 does not change it:

$$\begin{aligned} & \textbf{do}\ \{x \leftsquigarrow \mu;\ 1 \odot \textbf{return}\ x\} \\ = \quad & \{\text{scaling: } m = 1 \odot m\} \\ & \textbf{do}\ \{x \leftsquigarrow \mu;\ \textbf{return}\ x\} \\ = \quad & \{\text{monad right-identity law}\} \\ & \mu. \end{aligned}$$

□

Thus, on one hand, the constant-1 function is a density of **normal** 3 2 with respect to itself. On the other hand, the ratio between **dnorm** 0 1 and **dnorm** 3 2 is a density of **normal** 0 1 with respect to the same base measure **normal** 3 2. This ratio is plotted in Figure 5 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 2.10. (1) *If $\kappa$ is a density of $\xi$ with respect to $\mu$, and $\kappa'$ is a density of $\mu$ with respect to $\nu$, then the pointwise product $\lambda x.\,(\kappa(x) \cdot \kappa'(x))$ is a density of $\xi$ with respect to $\nu$. (This fact and Proposition 2.9 together mean that 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. We prove both parts by equational reasoning. For both parts, we use the fact that scaling a measure by two consecutive weights is same as scaling the measure by the product of the weights: $l \odot (k \odot m) = (l \cdot k) \odot m$. Also, to apply the monad associativity law, we note that the scaling expression $l \odot M$ is actually defined in Section 3.1 to be a mere abbreviation for **do** $\{$**factor** $l$; $M\}$.

(1)
$$
\begin{aligned}
& \xi \\
= \;& \{\kappa \text{ is a density of } \xi \text{ with respect to } \mu\} \\
& \mathbf{do}\ \{x \leftsquigarrow \mu;\ \kappa(x) \odot \mathbf{return}\ x\} \\
= \;& \{\kappa' \text{ is a density of } \mu \text{ with respect to } \nu\} \\
& \mathbf{do}\ \{x \leftsquigarrow \mathbf{do}\ \{x \leftsquigarrow \nu;\ \kappa'(x) \odot \mathbf{return}\ x\};\ \kappa(x) \odot \mathbf{return}\ x\} \\
= \;& \{\text{monad laws}\} \\
& \mathbf{do}\ \{x \leftsquigarrow \nu;\ \kappa'(x) \odot (\kappa(x) \odot \mathbf{return}\ x)\} \\
= \;& \{\text{scaling: } l \odot (k \odot m) = (l \cdot k) \odot m\} \\
& \mathbf{do}\ \{x \leftsquigarrow \nu;\ (\kappa'(x) \cdot \kappa(x)) \odot \mathbf{return}\ x\}.
\end{aligned}
$$

(2)
$$
\begin{aligned}
& \mathbf{do}\ \{x \leftsquigarrow \xi;\ (1/\kappa(x)) \odot \mathbf{return}\ x\} \\
= \;& \{\kappa \text{ is a density of } \xi \text{ with respect to } \mu\} \\
& \mathbf{do}\ \{x \leftsquigarrow \mathbf{do}\ \{x \leftsquigarrow \mu;\ \kappa(x) \odot \mathbf{return}\ x\};\ (1/\kappa(x)) \odot \mathbf{return}\ x\} \\
= \;& \{\text{monad laws}\} \\
& \mathbf{do}\ \{x \leftsquigarrow \mu;\ \kappa(x) \odot ((1/\kappa(x)) \odot \mathbf{return}\ x)\} \\
= \;& \{\text{scaling: } l \odot (k \odot m) = (l \cdot k) \odot m\} \\
& \mathbf{do}\ \{x \leftsquigarrow \mu;\ (\kappa(x) \cdot (1/\kappa(x))) \odot \mathbf{return}\ x\} \\
= \;& \{\kappa \text{ is } \mu\text{-almost everywhere finite and nonzero}\} \\
& \mathbf{do}\ \{x \leftsquigarrow \mu;\ 1 \odot \mathbf{return}\ x\} \\
= \;& \{\text{Proposition 2.9}\} \\
& \mu.
\end{aligned}
$$
□

*Example 2.11.* The reasoning in Example 2.8 can just as well be used to compare non-continuous distributions. For example, to adjudicate whether a certain black box is the clamping of **normal** 0 1 or of **normal** 3 2 to the interval $[0, 1]$, we can find a density of one clamped distribution with respect to the other, namely the ratio of the densities of the two clamped distributions with respect to the common base measure $\mu$ in (14). Our disintegrator is the first transformation to automate this reasoning, because these clamped distributions are not continuous (B1). Similarly, Wu et al.'s GPA problem (Section 1.2) is to adjudicate whether a certain black box is the clamping of a continuous distribution to the interval $[0, 10]$ or the clamping of a continuous distribution to the interval $[0, 4]$, and our disintegrator is the first to automate this reasoning exactly.

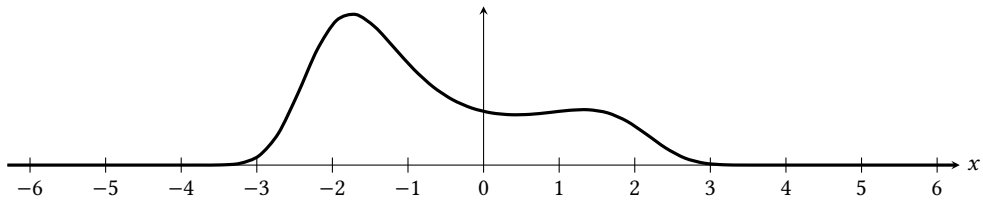


Fig. 6. The target density in Examples 2.13 and 2.17 with respect to the Lebesgue measure

*Example 2.12.* Comparing hypotheses is not the only calculation on models that densities are 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 densities 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 from MacKay [1998], it is easy to precisely define a distribution $\xi : \mathbb{M}\,\mathbb{R}$

$$\xi \;=\; \left(\lambda x.\, \exp(0.4(x-0.4)^2 - 0.08x^4)\right) \mathrel{=\!\!\odot} \textbf{lebesgue}, \tag{19}$$

but it is not obvious how to sample from it. (For one thing, it is not obvious how to compute or invert the integral of $\exp(0.4(x-0.4)^2 - 0.08x^4)$ so as to apply the *inverse transform method* [Devroye 1986, Theorem 2.1].)

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* [Kahn and Harris 1951]. It is also known as *likelihood weighting* in the special case where the proposal and target distributions are the prior and posterior distributions in Bayesian inference.

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 2.13.* Suppose we have a function $f : \mathbb{R} \to \mathbb{R}_+$ and we want to estimate its expectation with respect to $\xi$ in (19). For this target $\xi$, we can use **normal** 3 2 as the proposal, because it is well known how to sample from a normal distribution. To find the density of $\xi$ with respect to **normal** 3 2, we can divide their densities with respect to **lebesgue**, using Proposition 2.10. The form of (19) manifests a density of $\xi$ with respect to **lebesgue**, plotted in Figure 6. And a density of **normal** 3 2 is already plotted in Figure 4. So to estimate the expectation of $f$ with respect to $\xi$, we can sample $x$ from **normal** 3 2 and average $f(x)$ weighted by $\exp(0.4(x-0.4)^2 - 0.08x^4)/(\textbf{dnorm}\ 3\ 2)(x)$.

*Example 2.14.* 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

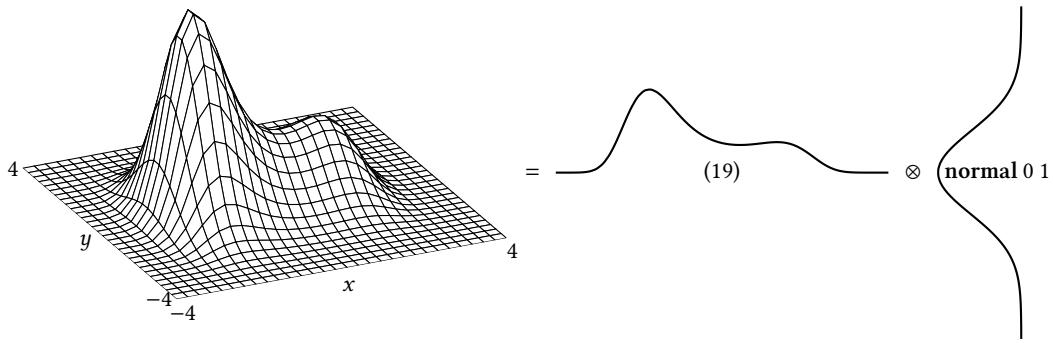


Fig. 7. The product measure of Figures 6 and 3

Lebesgue measure. As above, our disintegrator is the first transformation to find such densities (B1). Wu et al. [2018] developed two algorithms, *lexicographic* likelihood weighting and *lexicographic* particle filtering, that handle these distributions, but they carry out particular inference techniques rather than finding densities in general, and they do not allow customizing the proposal distribution.

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

*Definition 2.15.* A measure over a product $\alpha \times \beta$ is called a *joint measure*. Given two measures $\mu : \mathbb{M}\,\alpha$ and $\nu : \mathbb{M}\,\beta$, we can construct their *product measure* $\mu \otimes \nu : \mathbb{M}\,(\alpha \times \beta)$ as follows:

$$\mu \otimes \nu \;=\; \lambda f.\, \mu\big(\lambda x.\, \nu\big(\lambda y.\, f(x,y)\big)\big) \;=\; \mathbf{do}\ \{x \leftsquigarrow \mu;\ y \leftsquigarrow \nu;\ \mathbf{return}\ (x,y)\} \tag{20}$$

$$\;=\; \lambda f.\, \nu\big(\lambda y.\, \mu\big(\lambda x.\, f(x,y)\big)\big) \;=\; \mathbf{do}\ \{y \leftsquigarrow \nu;\ x \leftsquigarrow \mu;\ \mathbf{return}\ (x,y)\}. \tag{21}$$

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 7 depicts the product measure of two distributions over $\mathbb{R}$. In order to ensure *commutativity*, which means that the measures in (20) and (21) are defined and equal, Staton [2017] established the invariant that all measure expressions denote *s-finite kernels* (Section 3.2.1).

We also write $\mu^n$ for the $n$-ary product $\mu \otimes \cdots \otimes \mu : \mathbb{M}\,\alpha^n$.

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

$$\mu \otimes\!\!= \nu \;=\; \lambda f.\, \mu\big(\lambda x.\, \nu(x)\big(\lambda y.\, f(x,y)\big)\big) \;=\; \mathbf{do}\ \{x \leftsquigarrow \mu;\ y \leftsquigarrow \nu(x);\ \mathbf{return}\ (x,y)\}. \tag{22}$$

Intuitively, $\mu \otimes\!\!= \nu$ is like $\mu \otimes \nu$, except $\nu$ depends on the outcome of $\mu$. Symmetrically, we can construct the dependent product $\nu =\!\!\otimes \mu : \mathbb{M}\,(\beta \times \alpha)$ where $\nu$ again depends on $\mu$:

$$\nu =\!\!\otimes \mu \;=\; \lambda f.\, \mu\big(\lambda y.\, \nu(y)\big(\lambda x.\, f(x,y)\big)\big) \;=\; \mathbf{do}\ \{y \leftsquigarrow \mu;\ x \leftsquigarrow \nu(y);\ \mathbf{return}\ (x,y)\}. \tag{23}$$

These dependent products are useful for Metropolis-Hastings sampling, as we illustrate in Example 2.17 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 2.16. *Let $\kappa : \alpha \to \mathbb{R}_+$ be a density of $\xi : \mathbb{M}\,\alpha$ with respect to $\mu : \mathbb{M}\,\alpha$.*

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

*More generally, if we have instead $\kappa' : \alpha \to \beta \to \mathbb{R}_+$ and $\zeta, \nu : \alpha \to \mathbb{M}\,\beta$ such that $\kappa'(x)$ is a density of $\zeta(x)$ with respect to $\nu(x)$ for $\xi$-almost all $x$, then the dependent product measure $\xi \otimes\!\!= \zeta : \mathbb{M}\,(\alpha \times \beta)$ has the density $\lambda(x,y).\,(\kappa(x) \cdot \kappa'(x)(y)) : (\alpha \times \beta) \to \mathbb{R}_+$ with respect to $\mu \otimes\!\!= \nu : \mathbb{M}\,(\alpha \times \beta)$. And symmetrically for $\zeta =\!\!\otimes\, \xi$ with respect to $\nu =\!\!\otimes\, \mu$.*

Proof. We prove this standard result using equational reasoning on programs again. As in Proposition 2.10, we use the fact that scaling a measure by two weights is same as scaling by their product. We show only the proof of the "more generally" part:

$$
\begin{aligned}
& \xi \otimes\!\!= \zeta \\
= \;& \{\text{definition of } \otimes\!\!= \text{ (22)}\} \\
& \textbf{do } \{x \leftsquigarrow \xi;\ y \leftsquigarrow \zeta(x);\ \textbf{return } (x,y)\} \\
= \;& \{\kappa'(x) \text{ is a density of } \zeta(x) \text{ with respect to } \nu(x) \text{ for } \xi\text{-almost all } x\} \\
& \textbf{do } \{x \leftsquigarrow \xi;\ y \leftsquigarrow \textbf{do } \{y \leftsquigarrow \nu(x);\ \kappa'(x)(y) \odot \textbf{return } y\};\ \textbf{return } (x,y)\} \\
= \;& \{\kappa \text{ is a density of } \xi \text{ with respect to } \mu\} \\
& \textbf{do } \{x \leftsquigarrow \textbf{do } \{x \leftsquigarrow \mu;\ \kappa(x) \odot \textbf{return } x\};\ y \leftsquigarrow \textbf{do } \{y \leftsquigarrow \nu(x);\ \kappa'(x)(y) \odot \textbf{return } y\};\ \textbf{return } (x,y)\} \\
= \;& \{\text{monad laws, and again the abbreviation } \odot \text{ defined in Section 3.1}\} \\
& \textbf{do } \{x \leftsquigarrow \mu;\ \kappa(x) \odot \textbf{do } \{y \leftsquigarrow \nu(x);\ \kappa'(x)(y) \odot \textbf{return } (x,y)\}\} \\
= \;& \{\text{linearity (trivial commutativity)}\} \\
& \textbf{do } \{x \leftsquigarrow \mu;\ y \leftsquigarrow \nu(x);\ \kappa(x) \odot (\kappa'(x)(y) \odot \textbf{return } (x,y))\} \\
= \;& \{\text{scaling: } l \odot (k \odot m) = (l \cdot k) \odot m\} \\
& \textbf{do } \{x \leftsquigarrow \mu;\ y \leftsquigarrow \nu(x);\ (\kappa(x) \cdot \kappa'(x)(y)) \odot \textbf{return } (x,y)\} \\
= \;& \{\text{definition of } \otimes\!\!= \text{ and monad laws}\} \\
& \textbf{do } \{(x,y) \leftsquigarrow (\mu \otimes\!\!= \nu);\ (\kappa(x) \cdot \kappa'(x)(y)) \odot \textbf{return } (x,y)\}.
\end{aligned}
$$

□

*Metropolis-Hastings sampling* [Metropolis et al. 1953; Hastings 1970] is a popular inference technique that depends on a *target distribution* $\xi : \mathbb{M}\,\alpha$ and a *proposal kernel* $\zeta : \alpha \to \mathbb{M}\,\alpha$. The proposal kernel $\zeta$ specifies a search strategy by which to explore the target distribution $\xi$. The user of the technique specifies $\xi$ and $\zeta$ then calculates the *acceptance ratio*, a density of $\zeta \mathrel{=\!\!\otimes} \xi : \mathbb{M}\,\alpha^2$ with respect to $\xi \otimes\!\!= \zeta : \mathbb{M}\,\alpha^2$ [Tierney 1998]. This density is then used in the probabilistic body of a loop. The density is called a ratio because it is usually calculated by dividing densities of $\zeta \mathrel{=\!\!\otimes} \xi$ and $\xi \otimes\!\!= \zeta$ with respect to some common base measure, using Proposition 2.10.

*Example 2.17.* The target distribution $\xi : \mathbb{M}\,\mathbb{R}$ in equation (19) above gives a small instance of Metropolis-Hastings sampling whose acceptance ratio can be calculated using Bhat et al.'s procedure [2012, 2013]. Let us choose the proposal kernel $\zeta = \lambda x.\, \textbf{normal}\ x\ 1 : \mathbb{R} \to \mathbb{M}\,\mathbb{R}$, so $\zeta(x)$ has a density with respect to **lebesgue** for all $x$. By Proposition 2.16, with respect to the base measure $\textbf{lebesgue}^2$, the measure $\xi \otimes\!\!= \zeta$ has the density $\kappa : \mathbb{R}^2 \to \mathbb{R}_+$ defined by

$$\kappa \;=\; \lambda(x,y).\, \exp(0.4(x-0.4)^2 - 0.08x^4) \cdot \exp(-(y-x)^2/2)/\sqrt{2\pi}, \tag{24}$$

and the measure $\zeta \mathrel{=\!\!\otimes} \xi$ has the density $\kappa \circ \mathit{swap}$, where $\circ$ denotes function composition as usual and $\mathit{swap}(y,x) = (x,y)$. Thus by Proposition 2.10, the acceptance ratio is

$$\lambda(x,y).\, \frac{\exp(0.4(y-0.4)^2 - 0.08y^4) \cdot \exp(-(x-y)^2/2)/\sqrt{2\pi}}{\exp(0.4(x-0.4)^2 - 0.08x^4) \cdot \exp(-(y-x)^2/2)/\sqrt{2\pi}}. \tag{25}$$

Often in Metropolis-Hastings sampling, the target distribution ranges over a type $\alpha$ that is a product or sum type, and the proposal kernel is built from sub-kernels on the constituent types.

- Often $\alpha$ is a product type $\alpha_1 \times \alpha_2$. A value of a product type is a pair. For example, a value of type $\mathbb{R} \times \mathbb{R}$ is a pair of reals $(x_1, x_2)$. Perhaps they are the times of two events (*change points*). For a product type $\alpha$, a *single-site* kernel $\zeta : \alpha \to \mathbb{M}\,\alpha$ can be built out of sub-kernels

$\zeta_1 : \alpha \to \mathbb{M}\,\alpha_1$ and $\zeta_2 : \alpha \to \mathbb{M}\,\alpha_2$ as follows [Goodman et al. 2008; Wingate et al. 2011]: given $(x_1, x_2) : \alpha_1 \times \alpha_2$, flip a coin and either use $\zeta_1$ to update $x_1$ while keeping $x_2$ or use $\zeta_2$ to update $x_2$ while keeping $x_1$. This composite kernel can be expressed as

$$\begin{aligned} \zeta \;&=\; \lambda(x_1,x_2).\,\big(\tfrac{1}{2} \odot \mathbf{do}\ \{x_1' \leftsquigarrow \zeta_1(x_1,x_2);\ \mathbf{return}\ (x_1',x_2)\}\big) \\ &\qquad \oplus\ \big(\tfrac{1}{2} \odot \mathbf{do}\ \{x_2' \leftsquigarrow \zeta_2(x_1,x_2);\ \mathbf{return}\ (x_1,x_2')\}\big). \end{aligned} \tag{26}$$

- Often $\alpha$ is a sum type $\alpha_1 + \alpha_2$. A value of a sum type is a tagged value, and the summand types $\alpha_1$ and $\alpha_2$ may differ, so the type of the value tagged depends on whether the tag is **inl** or **inr**. For example, a value of type $\mathbb{R} + \mathbb{R}^2$ is either $\mathbf{inl}\ x_1$ or $\mathbf{inr}\ x_2$, where $x_1$ is a real and $x_2$ is a pair of reals. Perhaps $x_1$ is the time of one event and $x_2$ is the times of two events, and the choice of tag represents how many events there are. For a sum type $\alpha$, a *reversible-jump* kernel $\zeta : \alpha \to \mathbb{M}\,\alpha$ can be built out of sub-kernels $\zeta_1 : \alpha_1 \to \mathbb{M}\,\alpha$ and $\zeta_2 : \alpha_2 \to \mathbb{M}\,\alpha$ by case discrimination [Green 1995].

Despite the prevalence of single-site and reversible-jump proposal kernels, existing density calculation procedures cannot find their acceptance ratios, because the necessary base measure is not an independent product but rather a dependent product or disjoint sum respectively. For example, suppose that $\alpha = \mathbb{R}^2$, the target $\xi$ has a density with respect to $\mathbf{lebesgue}^2$, and the single-site kernel $\zeta$ in (26) is built out of sub-kernels $\zeta_1, \zeta_2 : \mathbb{R}^2 \to \mathbb{M}\,\mathbb{R}$ that always return continuous measures. Still, the measures $\xi \otimes\!= \zeta$ and $\zeta =\!\otimes\, \xi$ do not have densities with respect to $(\mathbf{lebesgue}^2)^2 : \mathbb{M}\,(\mathbb{R}^2)^2$. Rather, they have densities with respect to the dependent product

$$\mathbf{lebesgue}^2 \otimes\!= \lambda(x_1,x_2).\ (\mathbf{lebesgue} \oplus \mathbf{return}\ x_1) \otimes (\mathbf{lebesgue} \oplus \mathbf{return}\ x_2) : \mathbb{M}\,(\mathbb{R}^2)^2. \tag{27}$$

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

*Definition 2.18.* Let $\mu_1$ be a measure over $\alpha_1$ and $\mu_2$ be a measure over $\alpha_2$. The *disjoint sum measure* $\mu_1 \oplus \mu_2$ over the sum type $\alpha_1 + \alpha_2$ is defined by

$$\begin{aligned} \mu_1 \oplus \mu_2 \;&=\; (\mathbf{inl} \diamond \mu_1) \qquad\qquad\qquad\quad \oplus\ (\mathbf{inr} \diamond \mu_2) \\ &=\; \mathbf{do}\ \{x_1 \leftsquigarrow \mu_1;\ \mathbf{return}\ (\mathbf{inl}\ x_1)\} \oplus \mathbf{do}\ \{x_2 \leftsquigarrow \mu_2;\ \mathbf{return}\ (\mathbf{inr}\ x_2)\} \end{aligned} \tag{28}$$

using the shorthand $\diamond$ (also spelled *fmap* in Haskell) defined by $f \diamond \mu = \mathbf{do}\ \{x \leftsquigarrow \mu;\ \mathbf{return}\ (f\ x)\}$.

The challenge of computing acceptance ratios using these trickier base measures has motivated several publications [Green 1995; Tierney 1998; Wingate et al. 2011; Roberts et al. 2019]. Our disintegrator is the first to allow (indeed, infer) dependent products (B2) and disjoint sums (B3) as base measures, and thus to find these acceptance ratios automatically from programs such as (26).

### 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 2.19.* Given a joint distribution $\xi : \mathbb{M}\,(\alpha \times \beta)$, the *marginal* distribution $(\mathbf{fst} \diamond \xi) : \mathbb{M}\,\alpha$ over $\alpha$ is defined by

$$\mathbf{fst} \diamond \xi \;=\; \mathbf{do}\ \{(x,y) \leftsquigarrow \xi;\ \mathbf{return}\ x\} \;=\; \lambda f.\,\xi\big(\lambda(x,y).\,f(x)\big). \tag{29}$$

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

$$\xi \;=\; \mathbf{do}\ \{x \leftsquigarrow (\mathbf{fst} \diamond \xi);\ y \leftsquigarrow \kappa(x);\ \mathbf{return}\ (x,y)\}, \tag{30}$$

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

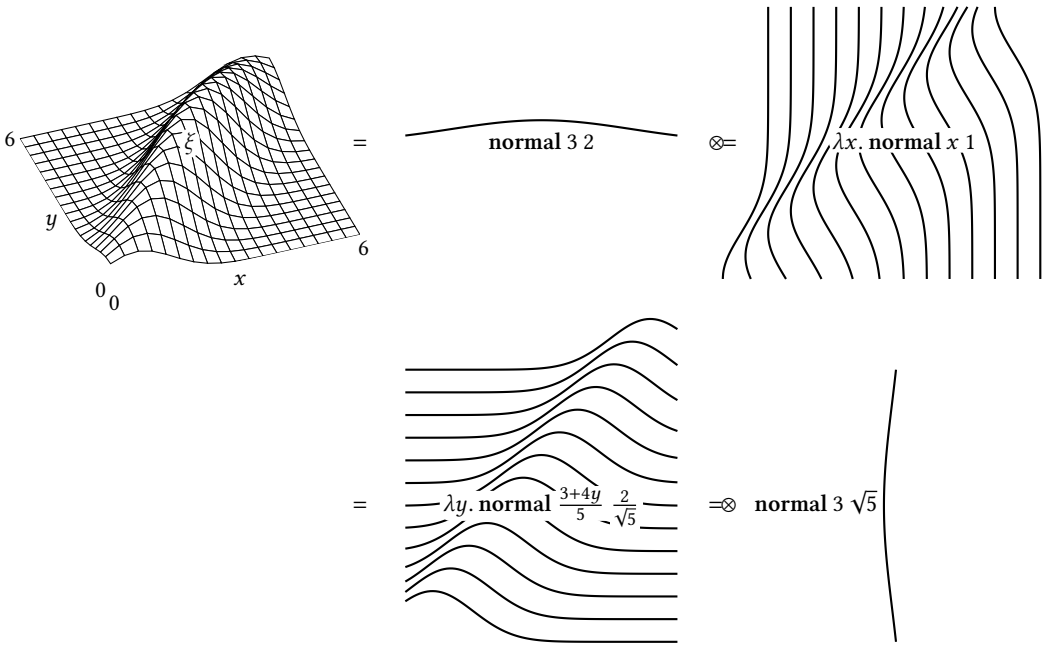


Fig. 8. Conditioning the joint distribution (31) on $x$ (top) and on $y$ (bottom). The left column shows the joint distribution; the middle column shows distributions over $x$ horizontally; the right column shows distributions over $y$ vertically. Each curve depicts a distribution, so each family of curves depicts a family of distributions.

*Example 2.20.* 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:

$$\xi \;=\; \mathbf{do}\ \{x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ y \leftsquigarrow \mathbf{normal}\ x\ 1;\ \mathbf{return}\ (x, y)\}. \tag{31}$$

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

The marginal distribution ($\mathbf{fst} \diamond \xi$) over $x$ is

$$\begin{aligned} & \mathbf{do}\ \{x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ y \leftsquigarrow \mathbf{normal}\ x\ 1;\ \mathbf{return}\ x\} \\ =\ & \mathbf{do}\ \{x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ |\mathbf{normal}\ x\ 1| \odot \mathbf{return}\ x\} \\ =\ & \mathbf{do}\ \{x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ 1 \odot \mathbf{return}\ x\} \;=\; \mathbf{normal}\ 3\ 2. \end{aligned} \tag{32}$$

Thus, we can read off from (31) that one conditional distribution over $y$ given $x$ is just $\lambda x.\ \mathbf{normal}\, x\, 1$. This decomposition is shown at the top of Figure 8. Hence, we have already used conditional distributions when specifying the model.

To condition on $y$, we rewrite (31) using equation (8), then commute the binding of $y$ to the front:

$$\begin{aligned} \xi \;&=\; \mathbf{do}\ \{x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ y \leftsquigarrow \mathbf{lebesgue};\ (\mathbf{dnorm}\ x\ 1)(y) \odot \mathbf{return}\ (x, y)\} \\ &=\; \mathbf{do}\ \{y \leftsquigarrow \mathbf{lebesgue};\ x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ (\mathbf{dnorm}\ x\ 1)(y) \odot \mathbf{return}\ (x, y)\} \\ &=\; \kappa' \mathbin{=\!\!\otimes} \mathbf{lebesgue} \qquad \text{where } \kappa' = \lambda y.\, \mathbf{do}\ \{x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ (\mathbf{dnorm}\ x\ 1)(y) \odot \mathbf{return}\ x\}. \end{aligned} \tag{33}$$

Thus, the marginal distribution ($\mathbf{snd} \diamond \xi$) over $y$ is $(\lambda y.\, |\kappa'(y)|) \mathbin{=\!\!\odot} \mathbf{lebesgue}$, and one conditional distribution over $x$ given $y$ is $\lambda y.\, |\kappa'(y)|^{-1} \odot \kappa'(y)$, the normalization of $\kappa'$. For this model, the marginal turns out equal to **normal** 3 $\sqrt{5}$ and the conditional turns out equal to $\lambda y.\ \mathbf{normal}\ \frac{3+4y}{5}\ \frac{2}{\sqrt{5}}$.

This decomposition is shown at the bottom of Figure 8. Hence, we have used our observation $y$ to update our *prior* belief about $x$, namely **normal** 3 2, and form our *posterior* belief about $x$, namely **normal** $\frac{3+4y}{5}$ $\frac{2}{\sqrt{5}}$. This update illustrates the use of conditional distributions for making inferences.

The choice of the base measure **lebesgue** in the calculation (33) 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 **lebesgue** as the base measure, then $\kappa'$ would be halved, but its normalization would remain the same.

Just as densities are not unique (as illustrated in Example 2.4), conditional distributions are not unique. Because $\lambda x.$ **normal** $x$ 1 is a conditional distribution over $y$ given $x$, so is

$$\lambda x. \begin{cases} \textbf{return } 12 & \text{if } x = 34 \\ 5 \odot \textbf{return } 6 & \text{if } x = 78 \\ \textbf{normal } x\ 1 & \text{otherwise.} \end{cases} \tag{34}$$

And because $\lambda y.$ **normal** $\frac{3+4y}{5}$ $\frac{2}{\sqrt{5}}$ is a conditional distribution over $x$ given $y$, so is

$$\lambda y. \begin{cases} \textbf{fail} & \text{if } y \in \mathbb{Z} \\ \textbf{lebesgue} & \text{if } y \in \mathbb{Q} \setminus \mathbb{Z} \\ \textbf{normal } \frac{3+4y}{5}\ \frac{2}{\sqrt{5}} & \text{otherwise.} \end{cases} \tag{35}$$

In general, given $\xi$, the conditional $\kappa$ is only unique up to marginal-almost-sure equivalence.

*Example 2.21.* The reasoning in Example 2.20 can just as well be used to make inferences from non-continuous observations. For example, to update our belief about $x$ using an observation of the clamping of **normal** $x$ 1 to the interval $[0, 1]$, we can condition the model

$$\xi \;=\; \textbf{do } \{x \leftsquigarrow \textbf{normal } 3\ 2;\ y \leftsquigarrow \textbf{normal } x\ 1;\ \textbf{return } (x, \max\{0, \min\{1, y\}\})\} \tag{36}$$

on its **snd** dimension like in (33), but using the base measure in (14) rather than **lebesgue**. This clamped model is a simple instance of a *Tobit model* [Tobin 1958; the relation between the names Tobin and Tobit is discussed in Shiller 1999]. Our disintegrator is the first transformation to automate this reasoning, because the clamped marginal over $y$ is not continuous (B1).

*Example 2.22.* Like densities, conditional distributions are 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 their 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 densities and conditional distributions and illustrated their applications using small examples and varying base measures. Disintegration [Chang and Pollard 1997] generalizes densities and conditional distributions.

*Definition 2.23.* A *disintegration* of a joint measure $\xi : \mathbb{M}\,(\alpha \times \beta)$ is a *base measure* $\mu : \mathbb{M}\,\alpha$ and a *kernel* $\kappa : \alpha \to \mathbb{M}\,\beta$ such that $\xi = \mu \otimes\!\!= \kappa$.

It is easy to see that disintegration generalizes conditional distributions (Definition 2.19): let $\mu = \textbf{fst} \diamond \xi$. To see that disintegration also generalizes densities (Definition 2.3), let $\beta$ be the unit type $\mathbb{1}$ and note that not only are the types $\alpha$ and $\alpha \times \mathbb{1}$ isomorphic, but $\mathbb{M}\,\mathbb{1}$ and $\mathbb{R}_+$ are also

isomorphic: map the measure $\nu : \mathbb{M}\,\mathbb{1}$ to the total $|\nu| : \mathbb{R}_+$, and map the total $r : \mathbb{R}_+$ to the measure $r \odot \mathbf{return}\ () : \mathbb{M}\,\mathbb{1}$. Just as densities and conditional distributions are not unique (as illustrated in Examples 2.4 and 2.20), disintegrations are not unique: even given both $\xi$ and $\mu$, the kernel $\kappa$ is only unique up to $\mu$-almost-sure equivalence.

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.

Definition 2.23 is Shan and Ramsey's [2017] specialization of Chang and Pollard's [1997] definition of disintegration, to the concrete case where the map $T$ is the projection $\mathbf{fst} : \alpha \times \beta \to \alpha$. Chang and Pollard's problem of disintegrating some $\lambda$ with respect to $T$ and $\mu$ reduces easily to our problem of disintegrating some $\xi$ with respect to $\mu$: just let $\xi = \mathbf{do}\ \{y \leftsquigarrow \lambda;\ \mathbf{return}\ (T(y), y)\}$.

## 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 (14), dependent products as in (27), and disjoint sums as in (28). 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 $\kappa : \alpha \to \mathbb{M}\,\beta$ such that $\xi = \mu \otimes\!\!= \kappa$.
- The *base-inferring disintegrator* takes the input program $\xi : \mathbb{M}\,(\alpha \times \beta)$ and returns a set of pairs of output programs $\mu : \mathbb{M}\,\alpha$ and $\kappa : \alpha \to \mathbb{M}\,\beta$ such that $\xi = \mu \otimes\!\!= \kappa$.

These transformations return (possibly empty) sets because they perform nondeterministic search and may find zero, one, or more solutions. Nevertheless, 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 $\mathit{genBase}(\alpha) : \mathbb{B}\,\alpha$. This base measure is Bhat et al.'s *stock measure* [2012].

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

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

After this extension, for any given type $\alpha$ 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 $\mu : \mathbb{M}\,\alpha$.

$$
\begin{array}{lll}
\text{Variables} & & x, y, z \\
\text{Real numbers} & & r \in \mathbb{R} \\
\text{Terms} & e, m, M ::= & x \mid r \mid () \mid (e, e) \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e \mid \mathbf{inl}\ e \mid \mathbf{inr}\ e \\
& & \mid \mathbf{lebesgue} \mid \mathbf{return}\ e \mid \mathbf{fail} \mid e \oplus e \mid \mathbf{do}\ \{g;\, e\} \\
& & \mid \mathbf{sqrt}\ e \mid e^{\mathbf{2}} \mid e \cdot e \mid e < e \mid \cdots \\
\text{Bindings (guards)} & g ::= & x \leftsquigarrow m \mid \mathbf{factor}\ e \mid \mathbf{let\ inl}\ x = e \mid \mathbf{let\ inr}\ x = e \\
\text{Heaps} & h ::= & [g; \ldots; g] \\
\text{Types} & \alpha, \beta, \gamma ::= & \mathbb{R} \mid \mathbb{1} \mid \alpha \times \beta \mid \alpha + \beta \mid \mathbb{M}\,\alpha
\end{array}
$$

Fig. 9. The syntax of core Hakaru

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

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

Fig. 10. Typing rules for measure terms. The measure term $e \odot M$ abbreviates $\mathbf{do}\ \{\mathbf{factor}\ e; M\}$. In the bottom row, $[x : \alpha] \ldots$ means that the typing derivation of $M : \mathbb{M}\,\gamma$ can use the discharged hypothesis $x : \alpha$; in other words, $M$ can use the bound variable $x$.

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 $\mathbb{M}\,(\alpha \times \mathbb{1})$, 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 9 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. We write core Hakaru syntax constructors in **bold**.

The definition of terms in Figure 9 does not include many operations on reals, because we use **sqrt** (square root) to illustrate how in general to handle invertible functions (such as negation, reciprocal, exp, log). Similarly, we use squaring $^{\mathbf{2}}$ and multiplication $\cdot$ to illustrate how in general to handle piecewise-invertible and coordinatewise-invertible functions (such as absolute value, +, min, max). The operations that the term grammar in Figure 9 lists explicitly (**sqrt**, $^{\mathbf{2}}$, $\cdot$) are treated by our disintegrator using invertibility, as spelled out in Section 4.2 and Figure 17 below. That treatment generalizes to other numeric operations, so the term grammar ends with ellipsis.

By way of explaining the measure terms, Figure 10 shows their typing rules; the rest of the type system is standard and elided. Our types are simple: as defined in Figure 9, 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 $\alpha$. We only consider well-typed terms as part of core Hakaru syntax.

The measure terms **lebesgue**, **return** $e$, **fail**, and $m_1 \oplus m_2$ denote respectively the Lebesgue measure, the Dirac measure at $e$ (in other words, monadic unit), the zero measure, and the sum of the measures $m_1$ and $m_2$. The remaining measure terms have the form **do** $\{g; M\}$, where $g$ is one of several kinds of *bindings*, also called *guards*. The typical such measure term is monadic bind, **do** $\{x \leftsquigarrow m; M\}$, where $m$ and $M$ are measure terms and $x$ takes scope over $M$. If $x$ is not used in $M$, then instead of $x$ we can write _, or write () if $x$ has type $\mathbb{1}$. Also, we abbreviate $x \leftsquigarrow$ **return** $e$ to **let** $x = e$. Another kind of guard builds the measure term **do** {**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 **normal** $e_1\, e_2$ as syntactic sugar for **dnorm** $e_1\, e_2 \mathrel{=\!\!\odot}$ **lebesgue**. (In turn, $\mathrel{=\!\!\odot}$ 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 inl** $x = e$, which may fail. If $e$ is **inl** $e_1$, then the term **do** {**let inl** $x = e$; $M$} just means $M\{x \mapsto e_1\}$, but if $e$ is **inr** $e_2$, then the same term **do** {**let inl** $x = e$; $M$} means the zero measure **fail**. Ordinary pattern matching on sum types can be recovered by duplicating a measure context $M[\ ]$:

$$M\left[\begin{array}{r}\textbf{case } e \textbf{ of } \textbf{inl } x_1 \to e_1 \\ \textbf{inr } x_2 \to e_2\end{array}\right] \quad = \quad \textbf{do } \{\textbf{let inl } x_1 = e; M[e_1]\} \oplus \textbf{do } \{\textbf{let inr } x_2 = e; M[e_2]\}. \tag{37}$$

Booleans **true**, **false** can be encoded as **inl** (), **inr** () as usual, and Boolean operations can be encoded in terms of **case**, so numeric comparisons such as equality can be encoded in terms of $<$. If $e$ is a Boolean expression, we write **observe** $e$ to mean the guard **let inl** _ $= e$.

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

$$\textbf{do } \{[g_1; \ldots; g_n]; M\} \quad = \quad \textbf{do } \{g_1; \ldots \textbf{do } \{g_n; M\} \ldots\}. \tag{38}$$

We also use the semicolon ; to concatenate heaps, and we omit the square brackets inside **do**. So for example, if $h = [g_1; g_2]$ then **do** $\{h; g_3; M\}$ = **do** $\{g_1;$ **do** $\{g_2;$ **do** $\{g_3; M\}\}\}$.

### 3.2 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 $\mu \mathrel{\otimes\!\!=} \kappa$. 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.

*3.2.1 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 **return** $3^2$ denotes (the function that maps () to) the Dirac distribution at 9. These denotations are defined compositionally, by induction on typing derivations as usual.

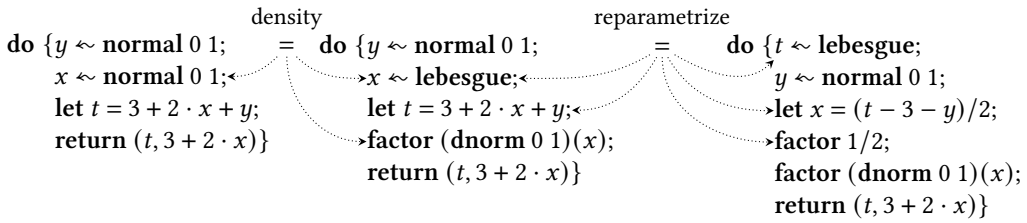


Fig. 11. Disintegration with respect to the Lebesgue base measure. Two reasoning steps equate the input program on the left to the output program on the right, which has the disintegrated form **lebesgue** $\otimes\!=\cdots$. Dotted lines show the bindings affected by each step.

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 $\kappa : \alpha \to \mathbb{M}\,\beta$.

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 **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 **do** {_ ↜ **lebesgue**; **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 **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

$$[\![\mathbf{do}\ \{x \leftsquigarrow m; M\}]\!](\rho) \quad = \quad \lambda f.\, [\![m]\!](\rho)\big(\lambda a.\, [\![M]\!](\rho, a)(f)\big), \tag{39}$$

where the pair $(\rho, a)$ extends the environment tuple $\rho$ with the value $a$. The second assurance is that monadic bindings commute. We have the equation

$$\mathbf{do}\ \{x \leftsquigarrow m_1; y \leftsquigarrow m_2; M\} \quad = \quad \mathbf{do}\ \{y \leftsquigarrow m_2; x \leftsquigarrow m_1; M\} \tag{40}$$

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.

*3.2.2 Fixed-base disintegration.* In addition to commutativity, disintegration requires just two equational reasoning principles: density and reparametrization. To see how, consider the disintegration shown in Figure 11. (This example is equivalent to (33), 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 $\mu \otimes\!= \kappa$, where the base measure $\mu$ is **lebesgue** and the kernel $\kappa$ is $\lambda t.$ **do** {...; **return** $(3 + 2 \cdot x)$}.

As shown in Figure 11, the disintegration is justified by a *density* step, a *reparametrization* 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 (8). The

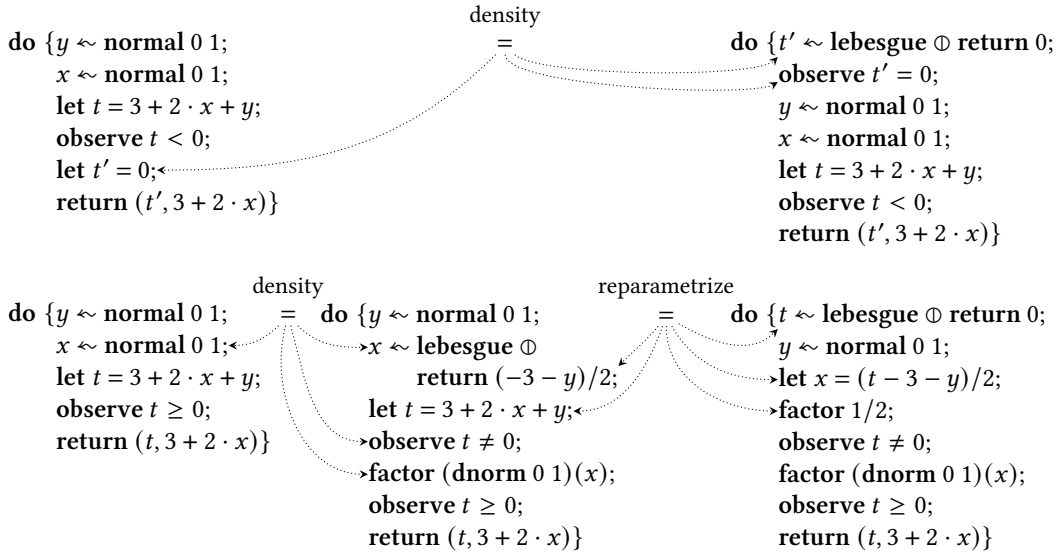


Fig. 12. Disintegration with respect to a base measure that is a discrete-continuous mixture

reparametrization 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:

$$\forall f, \quad \int_{\mathbb{R}} f\big(x, 3 + 2 \cdot x + y\big)\, dx \;=\; \int_{\mathbb{R}} \tfrac{1}{2} f\big((t - 3 - y)/2, t\big)\, dt. \tag{41}$$

Note that this step introduces the Jacobian **factor** $1/2$ in the final result.

Automatic disintegration actually begins with the desired form of the right-hand side of Figure 11 and asks, how can the input program equal a program that begins with $t \leftsquigarrow$ **lebesgue**? Driven by this desired form, and noticing that the input program determines $t$ in terms of $x$, the disintegrator attempts the reparametrization 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 reparametrization step to flesh out the right-hand side. This intuitive description is detailed formally in Section 4 below.

*3.2.3 Base-checking disintegration.* What if the base measure is not **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 **lebesgue** ⦶ **return** 0.

Figure 12 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 $t < 0$ so the observation is 0, and one component (bottom) where $t \geq 0$ so the observation is $t$. The results of disintegrating each component separately with respect to the same base measure can be summed to yield a disintegration of the original input. Each component is disintegrated using similar reasoning as before. The top component only needs a density step. The bottom component needs the same two steps as in Figure 11, with slightly larger terms. In particular, the density step requires **observe** $t \neq 0$ in the middle, and it is only when $t \neq 0$ that the Jacobian **factor** $1/2$ introduced by reparametrization is correct.

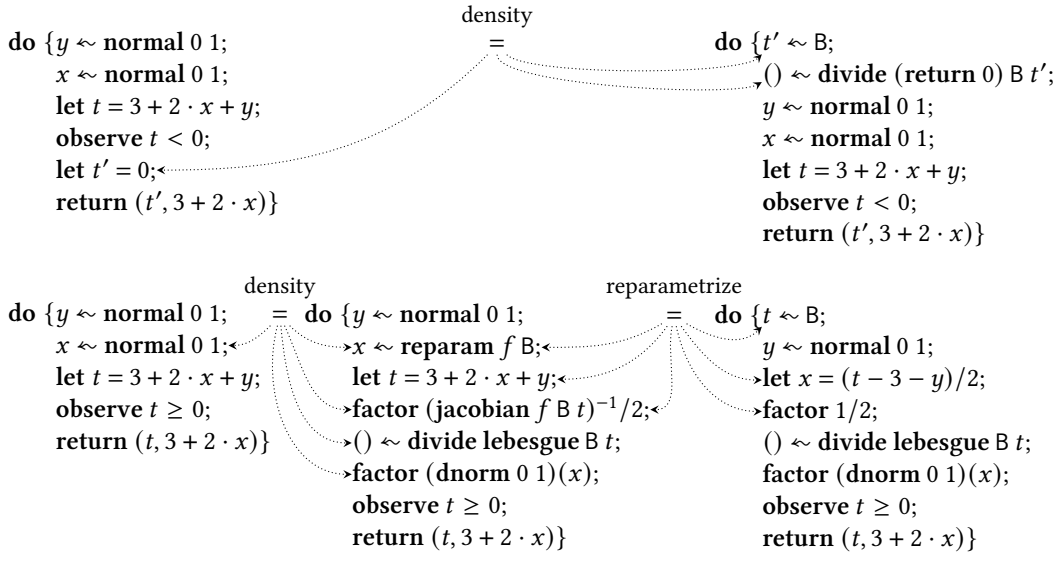


Fig. 13. Disintegration with respect to an unknown base measure. The invertible $f$ denotes $x \mapsto 3 + 2 \cdot x + y$.

*3.2.4 Base-inferring disintegration.* Manually specifying the base measure (such as in (27)) soon gets unwieldy, so we want a disintegrator that produces a base measure as output rather than taking it as input. To this end, we introduce base-measure variables B, analogous to the type variables classically used to turn a type checker into a type inferrer.

Figure 13 shows the same reasoning as in Figure 12 on the same input, but with respect to an unknown base measure B. The base-measure variable B appears in subterms that depend on the unknown base measure: To generalize **lebesgue** ⓘ **return** 0 in Figure 12 to become B in Figure 13, we must also generalize

- **observe** $t' = 0$ to become $() \leftsquigarrow$ **divide** (**return** 0) B $t'$, to denote a density of **return** 0 with respect to B at $t'$;
- **observe** $t \neq 0$ to become $() \leftsquigarrow$ **divide lebesgue** B $t$, to denote a density of **lebesgue** with respect to B at $t$; and
- similarly other expressions to become **reparam** $f$ B and **jacobian** $f$ B.

The output, to the right in Figure 13, invokes the densities **divide** (**return** 0) B and **divide lebesgue** B, so they had better exist. Thus, these occurrences of **divide** constrain the unknown base measure B such that **return** 0 and **lebesgue** both have densities with respect to it. In general, we can read off from the output of disintegration a set of **divide** constraints on base-measure variables, analogous to the set of constraints on type variables generated in the course of constraint-based type inference. Solving our two constraints yields the *principal base measure* **lebesgue** ⓘ **return** 0. That is, we have inferred that a base measure B works if and only if we can find a density of **lebesgue** ⓘ **return** 0 with respect to it. In particular, setting B = **lebesgue** ⓘ **return** 0 in Figure 13 recovers Figure 12.

## 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

Check base

$$m \;\sqsupseteq\; \mathbf{do}\ \{t \leftsquigarrow b;\ p \leftsquigarrow \mathit{check}\ m\ b\ t;\ \mathbf{return}\ (t, p)\}$$

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

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

Constrain value

$$\mathbf{do}\ \{h;\ \mathbf{let}\ t = e;\ M\} \;\sqsupseteq\; \mathbf{do}\ \{t \leftsquigarrow b;\ \lhd\ e\ b\ t\ \overline{M}\ h\}$$

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

$$\begin{aligned}
\lhd\ e\ \ (\mathbf{return}\ ())\ v\ c\ h &= c\ h\\
\lhd\ e\ \ (b_1 \otimes b_2)\ v\ c\ h &= \lhd\ (\mathit{fst}\ e)\ b_1\ (\mathit{fst}\ v)\ \big(\lhd\ (\mathit{snd}\ e)\ b_2\ (\mathit{snd}\ v)\ c\big)\ h\\
\lhd\ e\ \ (b_1 \oplus b_2)\ v\ c\ h &= \rhd\ e\ \big(\lambda v_0 h'.\ \mathit{outl}\ v_0\ (\lambda e_1 h''.\ \mathbf{do}\ \{\mathbf{let\ inl}\ t_1 = v;\ \lhd\ e_1\ b_1\ t_1\ c\ h''\})\ h'\\
&\qquad \mathbin{⦶}\ \mathit{outr}\ v_0\ (\lambda e_2 h''.\ \mathbf{do}\ \{\mathbf{let\ inr}\ t_2 = v;\ \lhd\ e_2\ b_2\ t_2\ c\ h''\})\ h'\big)\ h\\
\lhd\ u\ \ b\ \ v\ c\ h &= \emptyset \qquad \text{where } u \text{ is atomic}\\
\lhd\ r\ \ b\ \ v\ c\ h &= \emptyset \qquad \text{where } r \text{ is a literal real number}\\
\lhd\ (\mathbf{sqrt}\ e)\ b\ \ v\ c\ h &= \lessdot\ \mathbf{sqrt}^{+}\ e\ b\ v\ c\ h\\
\lhd\ e^2\ \ b\ \ v\ c\ h &= \lessdot\ \mathbf{sqr}^{+}\ e\ b\ v\ c\ h \mathbin{⦶} \lessdot\ \mathbf{sqr}^{-}\ e\ b\ v\ c\ h\\
\lhd\ (e_1 \cdot e_2)\ b\ \ v\ c\ h &= \rhd\ e_1\ \big(\lambda v_1.\ \lessdot\ (\mathbf{mul}\ v_1)\ e_2\ b\ v\ c\big)\ h \cup \rhd\ e_2\ \big(\lambda v_2.\ \lessdot\ (\mathbf{mul}\ v_2)\ e_1\ b\ v\ c\big)\ h\\
\lhd\ (\mathbf{fst}\ e)\ b\ \ v\ c\ h &= \rhd\ e\ \big(\lambda v_0.\ \lhd\ (\mathit{fst}\ v_0)\ b\ v\ c\big)\ h \qquad \text{unless } e \text{ is atomic}\\
\lhd\ (\mathbf{snd}\ e)\ b\ \ v\ c\ h &= \rhd\ e\ \big(\lambda v_0.\ \lhd\ (\mathit{snd}\ v_0)\ b\ v\ c\big)\ h \qquad \text{unless } e \text{ is atomic}\\
\lhd\ x\ b\ v\ c\ [h_1;\ x \leftsquigarrow m;\ h_2] &= \lll\ m\ b\ v\ (\overline{[\mathbf{let}\ x = v;\ h_2]}\ \fatsemi\ c) \quad h_1\\
\lhd\ x\ b\ v\ c\ [h_1;\ \mathbf{let\ inl}\ x = e;\ h_2] &= \rhd\ e\ \big(\lambda v_0.\ \mathit{outl}\ v_0\ (\lambda e_1.\ \lhd\ e_1\ b\ v\ (\overline{[\mathbf{let}\ x = v;\ h_2]}\ \fatsemi\ c))\big)\ h_1\\
\lhd\ x\ b\ v\ c\ [h_1;\ \mathbf{let\ inr}\ x = e;\ h_2] &= \rhd\ e\ \big(\lambda v_0.\ \mathit{outr}\ v_0\ (\lambda e_2.\ \lhd\ e_2\ b\ v\ (\overline{[\mathbf{let}\ x = v;\ h_2]}\ \fatsemi\ c))\big)\ h_1
\end{aligned}$$

Constrain op

$$\mathbf{do}\ \{h;\ \mathbf{observe}\ (e \in \mathit{dom}\ f);\ \mathbf{let}\ t = f\ @\ e;\ M\} \;\sqsupseteq\; \mathbf{do}\ \{t \leftsquigarrow b;\ \lessdot\ f\ e\ b\ t\ \overline{M}\ h\}$$

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

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

Constrain outcome

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

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

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

Fig. 14. The restricted base-checking disintegrator. The meta type constructors $\lceil\cdot\rceil$, $\lfloor\cdot\rfloor$, and $\{\cdot\}$ are explained in this section, as are the continuation builders $\overline{\cdot}$ and $\fatsemi$. The specification relation $\sqsupseteq$ is defined in Definition 4.1. The supporting functions **mul**, **div**, $\mathbf{sqr}^{+}$, $\mathbf{sqr}^{-}$, $\mathbf{sqrt}^{+}$, $\in\!\mathit{dom}$, $\in\!\mathit{rng}$, $@$, *inv*, *jacobian*, *reparam*, and $\div$ are explained in Section 4.2. The supporting functions $\rhd$, $\gg\!\!=$, *fst*, *snd*, *outl*, and *outr* are explained in Section 4.3.

$$
\begin{array}{lll}
\text{Head normal forms} & v ::= u \mid r \mid () \mid (e, e) \mid \mathbf{inl}\ e \mid \mathbf{inr}\ e & \\
& \quad \mid \mathbf{lebesgue} \mid \mathbf{return}\ e \mid \mathbf{fail} \mid m \oplus m \mid \mathbf{do}\ \{g; M\} & \\
\text{Atomic terms} & u ::= z \text{ (not bound in heap)} \mid \mathbf{fst}\ u \mid \mathbf{snd}\ u & \\
& \quad \mid \mathbf{sqrt}\ u \mid u^{\mathbf{2}} \mid u \cdot u \mid u \cdot r \mid r \cdot u \mid u < u \mid u < r \mid r < u \mid \cdots & \\
\text{Bases} & b ::= \mathbf{lebesgue} \mid \mathbf{return}\ () \mid b \otimes b \mid b \oplus b &
\end{array}
$$

Fig. 15. Additional syntactic categories used to describe the disintegrator

handle base measures that are disjoint sums. 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 14 shows the top four functions that make up our restricted base-checking disintegrator: *check*, $\lhd$, $\lessdot$, $\lll$. 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 **mul**, **div**, $\mathbf{sqr}^+$, $\mathbf{sqr}^-$, $\mathbf{sqrt}^+$, $\in dom$, $\in rng$, @, *inv*, *jacobian*, *reparam*, and $\div$.
- Section 4.3 explains $\rhd$, $\ggg$, *fst*, *snd*, *outl*, and *outr*.

In general, we write object syntax constructors in **bold**, and meta-functions in *italics*.

At the top of Figure 14 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 and syntactic categories.* On the second line of Figure 14, the (meta) type $\lceil \mathbb{M}\,(\alpha \times \beta) \rceil$ consists of the core Hakaru terms of type $\mathbb{M}\,(\alpha \times \beta)$, and the (meta) type $\lfloor \alpha \rfloor$ consists of *head normal forms* of type $\alpha$. (Other meta types heap and invertible are explained further below.)

Head normal forms are a subset of core Hakaru terms that we define in order to describe the disintegrator. Their grammar is shown in Figure 15. Intuitively, whereas a term is a part of a program yet to be run, a head normal form is a symbolic representation of what the disintegrator knows about the *result* of running a program. For example, the term 1 + 2 is not a head normal form; *partial evaluation* (Section 4.3) turns it into the head normal form 3. To take another example, the term $\mathbf{fst}\ (\mathbf{inl}\ x, \mathbf{inr}\ y)$ is not a head normal form; partial evaluation turns it into $\mathbf{inl}\ x$, which is a head normal form even though $x$ is a variable whose binding the disintegrator could try to look up. Certain head normal forms are *atomic*, which means roughly that nothing is known about them. For example, suppose that nothing is known about the variable $z$. Then, nothing is known about $1 + z$ and $\mathbf{fst}\ z$ either, so these terms are all atomic and hence head normal forms.

Technically, 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. (Strict functions are functions that inspect their inputs, including **sqrt**, $^{\mathbf{2}}$, $\cdot$, and most numeric operations one might add to core Hakaru such as addition, subtraction, division, exp, log, absolute value, min, max.) A head normal form $v$ is either an atomic term $u$ 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 real literals $r$ and measure terms.

*Base-measure language.* The bottom of Figure 15 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 *genBase* defined below maps each non-measure type $\alpha$ to its unique base

measure $genBase(\alpha) : \mathbb{B}\,\alpha$:

$$\begin{aligned} genBase(\mathbb{R}) &= \mathbf{lebesgue}, & genBase(\alpha \times \beta) &= genBase(\alpha) \otimes genBase(\beta), \\ genBase(\mathbb{1}) &= \mathbf{return}\ (), & genBase(\alpha + \beta) &= genBase(\alpha) \oplus genBase(\beta). \end{aligned} \tag{42}$$

This base language is new. The goal of this paper is to relax the restriction it represents: we extend it in Figure 22 and again in Figure 25.

*Nondeterminism.* The return type of *check* is $\{\lfloor \mathbb{M}\,\beta \rfloor\}$, which means a set (of head normal forms of type $\mathbb{M}\,\beta$, 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 14, the symbol $\sqsupseteq$ appears to relate two terms but actually relates two sets of terms. The right-hand side is the set of all terms $\mathbf{do}\ \{t \leftsquigarrow b;\ p \leftsquigarrow e;\ \mathbf{return}\ (t, p)\}$ 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 $S \sqsupseteq T$ to mean that the set of denotations of elements of $S$ is a superset of the set of denotations of elements of $T$. In other words, every term in $T$ denotes the same as some term in $S$.

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

In other definitions (such as the next function $\lhd$ in Figure 14), we write $\emptyset$ for the empty set of terms and $\cup$ for the union of two sets of terms. These are the only two ways to incur nondeterminism. Note that $\emptyset$ is different from **fail**, which is (a singleton set of) a term that denotes the zero measure, and $\cup$ is different from $\oplus$, 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 $\mathsf{heap} \to \{\lfloor \mathbb{M}\,\gamma \rfloor\}$ and maintain the heap as a piece of mutable state. We use the metavariable $c$ for these continuations. They typically have the form $\overline{M}$, defined as the function

$$\overline{M} \;=\; \lambda h.\, \mathbf{do}\ \{h;\, M\}, \tag{43}$$

which implicitly builds a singleton set by wrapping the given heap $h$ around some measure term $M$. Thus, the third line of Figure 14 defines *check* as the following steps in continuation-passing style: given a joint distribution $m$, a base measure $b$, and an index $t$ (usually a variable representing an observation), the function *check* starts with the empty heap [], invokes $\gg\!\!=$ on $m$ to get $v$, invokes $\lhd$ on $\mathbf{fst}\ v$, and wraps the final heap around $\mathbf{return}\ (\mathbf{snd}\ v)$ to form the result.

In other definitions (such as $\lhd$ in Figure 14), we write $\overline{h_0}$ for the heap-to-heap function

$$\overline{h_0} \;=\; \lambda h.\, [h;\, h_0], \tag{44}$$

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

### 4.1 From specifications to implementations

All the specifications boxed in Figure 14 have the form

$$\boxed{\ldots \;\sqsupseteq\; \mathbf{do}\ \{t \leftsquigarrow b; \ldots\}}, \tag{45}$$

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 $\sqsupseteq$ 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 $t \leftsquigarrow b$. Whereas the external interface *check* rewrites an arbitrary measure term $m$ with respect to a base measure $b$, the other functions $\lhd$ ("constrain value"), $\lessdot$ ("constrain op"), and $\lll$ ("constrain outcome") rewrite measure terms of specific forms, focusing on an expression $e$, operation $f$, or action $m$ 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 ($\emptyset$).

The boxed specifications in Figure 14 formalize the division of labor among the constraining functions $\lhd$, $\lessdot$, and $\lll$. This division of labor is illustrated by the example disintegration problem solved manually in Section 3.2.2 above and automatically in Figure 16 below.

**Constrain value** $\lhd\, e\ b\ t$ constrains the value of $e$ (of type $\alpha$) to be $t$ (of type $\alpha$). In the example, finding a disintegration requires constraining the value of $3 + 2 \cdot x + y$ to be $t$.

**Constrain op** $\lessdot\, f\ e\ b\ t$ constrains the result of the numeric operation $f$ on $e$ (of type $\mathbb{R}$) to be $t$ (of type $\mathbb{R}$). In the example, constraining the value of $3 + 2 \cdot x + y$ to be $t$ reduces to constraining the result of the numeric operation $3 + 2 \cdot - + y$ on $x$ to be $t$. That in turn reduces by reparametrization to constraining the value of $x$ to be $(t - 3 - y)/2$.

**Constrain outcome** $\lll\, m\ b\ t$ constrains the outcome of $m$ (of type $\mathbb{M}\,\alpha$) to be $t$ (of type $\alpha$). Continuing with the example, constraining the value of $x$ reduces to constraining the outcome of its distribution **normal** 0 1. That in turn requires finding a density of the distribution **normal** 0 1 with respect to the given base measure $b =$ **lebesgue**.

All the implementations in Figure 14 are derived from the semantic specifications by equational proofs. By *derive* [Hughes 1995; Hutton and Meijer 1996], we mean that the right-hand side of each implementation equation in Figure 14 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 $\lll$ **fail** is derived and proven from the specification

$$\boxed{\mathbf{do}\ \{h; t \leftsquigarrow m; M\} \;\sqsupseteq\; \mathbf{do}\ \{t \leftsquigarrow b; \lll m\ b\ t\ \overline{M}\ h\}} \tag{46}$$

by substituting **fail** for $m$ and equating both sides to **fail**:

$$\begin{aligned}
& \mathbf{do}\ \{h; t \leftsquigarrow \mathbf{fail}; M\} \\
= \;& \{\mathbf{fail}\ \text{distributivity (17) and (18)}\} \\
& \mathbf{fail} \\
= \;& \{\mathbf{fail}\ \text{distributivity (17)}\} \\
& \mathbf{do}\ \{t \leftsquigarrow b; \mathbf{fail}\}
\end{aligned}$$

$$
\begin{array}{ll}
= & \{\text{definition of } \lll\} \\
& \quad \mathbf{do}\ \{t \leftsquigarrow b;\ \lll\ \mathbf{fail}\ b\ t\ \overline{M}\ h\}.
\end{array} \tag{47}
$$

The first two steps of (47) prove an instance of the specification (46) in which the function call $\lll\ \mathbf{fail}\ b\ t\ \overline{M}\ h$ is replaced by the term **fail**. Matching that function call against that term gives the implementation equation $\lll\ \mathbf{fail}\ b\ v\ c\ h = \mathbf{fail}$ in Figure 14. Filling in that implementation equation allows us to proceed with the final step of (47) and finish this case of the proof that $\lll$ satisfies (46). Intuitively, this implementation disintegrates the zero measure by producing the zero measure.

Similarly, the case $\lll\ (m_1 \oplus m_2)$ is derived and proven from (46) by substituting $m_1 \oplus m_2$ for $m$ and using the induction hypotheses derived also from (46) by substituting $m_1$ and $m_2$ for $m$:

$$
\begin{array}{ll}
& \quad \mathbf{do}\ \{h;\ t \leftsquigarrow m_1 \oplus m_2;\ M\} \\
= & \quad \{\oplus \text{ distributivity (17) and (18)}\} \\
& \quad \mathbf{do}\ \{h;\ t \leftsquigarrow m_1;\ M\} \oplus \mathbf{do}\ \{h;\ t \leftsquigarrow m_2;\ M\} \\
\sqsupseteq & \quad \{\text{induction hypotheses}\} \\
& \quad \mathbf{do}\ \{t \leftsquigarrow b;\ \lll\ m_1\ b\ t\ \overline{M}\ h\} \oplus \mathbf{do}\ \{t \leftsquigarrow b;\ \lll\ m_2\ b\ t\ \overline{M}\ h\} \\
= & \quad \{\oplus \text{ distributivity (17)}\} \\
& \quad \mathbf{do}\ \{t \leftsquigarrow b;\ (\lll\ m_1\ b\ t\ \overline{M}\ h \oplus \lll\ m_2\ b\ t\ \overline{M}\ h)\} \\
= & \quad \{\text{definition of } \lll\} \\
& \quad \mathbf{do}\ \{t \leftsquigarrow b;\ \lll\ (m_1 \oplus m_2)\ b\ t\ \overline{M}\ h\}.
\end{array} \tag{48}
$$

All but the last step of (48) proves an instance of the specification (46) in which the function call $\lll\ (m_1 \oplus m_2)\ b\ t\ \overline{M}\ h$ is replaced by the term $\lll\ m_1\ b\ t\ \overline{M}\ h \oplus \lll\ m_2\ b\ t\ \overline{M}\ h$. Matching that function call against that term gives the implementation equation $\lll\ (m_1 \oplus m_2)\ b\ v\ c\ h = \lll\ m_1\ b\ v\ c\ h \oplus \lll\ m_2\ b\ v\ c\ h$ in Figure 14. Filling in that implementation equation allows us to proceed with the last step of (48) and finish this case of the proof that $\lll$ satisfies (46). Intuitively, this implementation disintegrates the sum of two measures by disintegrating each summand separately. (Duplicating the continuation $c$ to deal with the summands $m_1$ and $m_2$ separately 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 $\lll\ (\mathbf{return}\ e)$, which is derived and proven from (46) by substituting **return** $e$ for $m$ and using the induction hypothesis that is the specification of $\lhd$:

$$
\begin{array}{ll}
& \quad \mathbf{do}\ \{h;\ t \leftsquigarrow \mathbf{return}\ e;\ M\} \\
= & \quad \{\text{abbreviation}\} \\
& \quad \mathbf{do}\ \{h;\ \mathbf{let}\ t = e;\ M\} \\
\sqsupseteq & \quad \{\text{specification of } \lhd\} \\
& \quad \mathbf{do}\ \{t \leftsquigarrow b;\ \lhd\ e\ b\ t\ \overline{M}\ h\} \\
= & \quad \{\text{definition of } \lll\} \\
& \quad \mathbf{do}\ \{t \leftsquigarrow b;\ \lll\ (\mathbf{return}\ e)\ b\ t\ \overline{M}\ h\}.
\end{array} \tag{49}
$$

(Intuitively, this case constrains the outcome of a deterministic distribution at $e$ to be $t$ by constraining the value of $e$ to be $t$.) Similarly, the definition of top-level *check* is derived and proven from the specifications of $\rhd\!\!>$ and $\lhd$:

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

$$\boxed{m \;\sqsupseteq\; \mathbf{do}\,\{t \leftsquigarrow b;\, p \leftsquigarrow \mathit{check}\; m\; b\; t;\, \mathbf{return}\;(t,p)\}}. \tag{50}$$

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

Proof. We reason equationally from the right-hand side of (50) to the left-hand side, using many separately proven lemmas stated in figures:

$$\begin{aligned}
&\mathbf{do}\,\big\{t \leftsquigarrow b;\, p \leftsquigarrow \mathit{check}\; m\; b\; t;\, \mathbf{return}\;(t,p)\big\}\\
=\;& \{\text{definition of } \mathit{check} \text{ (Figure 14)}\}\\
&\mathbf{do}\,\big\{t \leftsquigarrow b;\, p \leftsquigarrow \gg\!= m\,\big(\lambda v.\, \lhd\,(\mathbf{fst}\; v)\; b\; t\; \overline{(\mathbf{return}\;(\mathbf{snd}\; v))}\big)\;[];\, \mathbf{return}\;(t,p)\big\}\\
=\;& \{\text{associativity of } \gg\!= \text{ (Figure 21)}\}\\
&\mathbf{do}\,\big\{t \leftsquigarrow b;\, \gg\!= m\,\big(\lambda v h.\, \mathbf{do}\,\{p \leftsquigarrow \lhd\,(\mathbf{fst}\; v)\; b\; t\; \overline{(\mathbf{return}\;(\mathbf{snd}\; v))}\; h;\, \mathbf{return}\;(t,p)\}\big)\;[]\big\}\\
=\;& \{\text{associativity of } \lhd \text{ (Figure 21)}\}\\
&\mathbf{do}\,\big\{t \leftsquigarrow b;\, \gg\!= m\,\big(\lambda v h.\, \lhd\,(\mathbf{fst}\; v)\; b\; t\; (\lambda h'.\, \mathbf{do}\,\{p \leftsquigarrow \mathbf{do}\,\{h';\, \mathbf{return}\;(\mathbf{snd}\; v)\};\, \mathbf{return}\;(t,p)\})\; h\big)\;[]\big\}\\
=\;& \{\text{monad laws}\}\\
&\mathbf{do}\,\big\{t \leftsquigarrow b;\, \gg\!= m\,\big(\lambda v h.\, \lhd\,(\mathbf{fst}\; v)\; b\; t\; \overline{(\mathbf{return}\;(t, \mathbf{snd}\; v))}\; h\big)\;[]\big\}\\
=\;& \{\text{commutativity of } \gg\!= \text{ (Figure 21)}\}\\
&\gg\!= m\,\big(\lambda v h.\, \mathbf{do}\,\{t \leftsquigarrow b;\, \lhd\,(\mathbf{fst}\; v)\; b\; t\; \overline{(\mathbf{return}\;(t, \mathbf{snd}\; v))}\; h\}\big)\;[]\\
\sqsubseteq\;& \{\text{specification of } \lhd \text{ (Figure 14)}\}\\
&\gg\!= m\,\big(\lambda v h.\, \mathbf{do}\,\{h;\, \mathbf{let}\; t = \mathbf{fst}\; v;\, \mathbf{return}\;(t, \mathbf{snd}\; v)\}\big)\;[]\\
=\;& \{\text{monad laws}\}\\
&\gg\!= m\,(\lambda v.\, \overline{\mathbf{return}\; v})\;[]\\
=\;& \{\text{specification of } \gg\!= \text{ (Figure 19)}\}\\
&\mathbf{do}\,\{x \leftsquigarrow m;\, \mathbf{return}\; x\} = m.
\end{aligned}$$

□

*Remark* 4.3. Theorem 4.2 assures soundness, not completeness, as with all our specifications stated using $\sqsupseteq$ (Definition 4.1). Our disintegrator is not complete, and we have not characterized when it succeeds (Section 9). And when our disintegrator does succeed, we have not characterized which of the infinite number of solutions (Examples 2.4 and 2.20) it chooses. After all, like most program transformations, our disintegrator is sensitive to the syntax of the input program (Section 3.1) and not just its semantics (Section 3.2.1).

*A sample run.* For concreteness, Figure 16 shows the automatic disintegrator performing essentially the same reasoning as Figure 11 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 $\gg\!=$ ("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 $\lessdot$ emits **factor** $1/2$ to carry out the reparametrization step in Figure 11.
- On line 10, the meta-function $\ll\!\!\lhd$ emits **dnorm** to carry out the density step in Figure 11.

1 $\mathit{check}\ (\mathbf{do}\ \{y \leftsquigarrow \mathbf{normal}\ 0\ 1;\ x \leftsquigarrow \mathbf{normal}\ 0\ 1;\ \mathbf{return}\ (3 + 2 \cdot x + y, 3 + 2 \cdot x)\})\ \mathbf{lebesgue}\ t$

2 $= \bowtie (\mathbf{do}\ \{y \leftsquigarrow \mathbf{normal}\ 0\ 1;\ x \leftsquigarrow \mathbf{normal}\ 0\ 1;\ \mathbf{return}\ (3 + 2 \cdot x + y, 3 + 2 \cdot x)\})$
$(\lambda v.\ \triangleleft\ (\mathbf{fst}\ v)\ \mathbf{lebesgue}\ t\ \overline{(\mathbf{return}\ (\mathbf{snd}\ v))})$
$[]$

3 $= \triangleright\ (3+2\cdot x+y, 3+2\cdot x)\ (\lambda v.\ \triangleleft\ (\mathbf{fst} v)\ \mathbf{lebesgue}\ t\ \overline{(\mathbf{return}\ (\mathbf{snd}\ v))})\ [y \leftsquigarrow \mathbf{normal}\ 0\ 1;\ x \leftsquigarrow \mathbf{normal}\ 0\ 1]$

4 $= \triangleleft\ (3 + 2 \cdot x + y)\ \mathbf{lebesgue}\ t\ \overline{(\mathbf{return}\ (3 + 2 \cdot x))}\ [y \leftsquigarrow \mathbf{normal}\ 0\ 1;\ x \leftsquigarrow \mathbf{normal}\ 0\ 1]$

5 $= \triangleright\ y\ (\lambda v.\ \lessdot\ (3 + 2 \cdot - + v)\ x\ \mathbf{lebesgue}\ t\ \overline{(\mathbf{return}\ (3 + 2 \cdot x))})\ [y \leftsquigarrow \mathbf{normal}\ 0\ 1;\ x \leftsquigarrow \mathbf{normal}\ 0\ 1]$

6 $= \bowtie (\mathbf{normal}\ 0\ 1)$
$(\lambda v h.\ \lessdot\ (3 + 2 \cdot - + v)\ x\ \mathbf{lebesgue}\ t\ \overline{(\mathbf{return}\ (3 + 2 \cdot x))}\ [h;\ \mathbf{let}\ y = v;\ x \leftsquigarrow \mathbf{normal}\ 0\ 1])$
$[]$

7 $= \mathbf{do}\ \{z \leftsquigarrow \mathbf{normal}\ 0\ 1;$
$\lessdot\ (3 + 2 \cdot - + z)\ x\ \mathbf{lebesgue}\ t\ \overline{(\mathbf{return}\ (3 + 2 \cdot x))}\ [\mathbf{let}\ y = z;\ x \leftsquigarrow \mathbf{normal}\ 0\ 1]\}$

8 $= \mathbf{do}\ \{z \leftsquigarrow \mathbf{normal}\ 0\ 1;\ \mathbf{factor}\ 1/2;$
$\triangleleft\ x\ \mathbf{lebesgue}\ \frac{t-3-z}{2}\ \overline{(\mathbf{return}\ (3 + 2 \cdot x))}\ [\mathbf{let}\ y = z;\ x \leftsquigarrow \mathbf{normal}\ 0\ 1]\}$

9 $= \mathbf{do}\ \{z \leftsquigarrow \mathbf{normal}\ 0\ 1;\ \mathbf{factor}\ 1/2;$
$\lll\ (\mathbf{normal}\ 0\ 1)\ \mathbf{lebesgue}\ \frac{t-3-z}{2}\ \overline{\mathbf{do}\ \{\mathbf{let}\ x = \frac{t-3-z}{2};\ \mathbf{return}\ (3 + 2 \cdot x)\}}\ [\mathbf{let}\ y = z]\}$

10 $= \mathbf{do}\ \{z \leftsquigarrow \mathbf{normal}\ 0\ 1;\ \mathbf{factor}\ 1/2;$
$\mathbf{factor}\ (\mathbf{dnorm}\ 0\ 1)(\frac{t-3-z}{2});\ \mathbf{let}\ y = z;\ \mathbf{let}\ x = \frac{t-3-z}{2};\ \mathbf{return}\ (3 + 2 \cdot x)\}$

Fig. 16. A transcript of the automatic disintegrator in action. The disintegration found here is same as in Figure 11, except we make several semantics-preserving simplifications: On line 1, we inline $t$. On line 4, we treat $3 + 2 \cdot - + -$ as a binary operator akin to multiplication, and so to call $\lessdot$, we treat $3 + 2 \cdot - + v$ as an invertible akin to $\mathbf{mul}\ v$. On lines 6 and 9, we treat $\mathbf{normal}\ 0\ 1$ as a primitive, not defined in terms of **lebesgue**.

### 4.2 Constraining with respect to a base measure

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

First, we add a base-measure argument $b : \mathbb{B}\ \alpha$ to every function in Figure 14. This new argument is inspected in the first three cases of $\triangleleft$ in Figure 14.

- The case $\triangleleft\ e\ (\mathbf{return}\ ())$ fleshes out Shan and Ramsey's remark that "on countable spaces, life is easy". Intuitively, because the term $e$ has the unit type, this case constrains the value of $e$ by doing nothing. The equational derivation of this case uses the fact that $e$ is equal to $()$.
- The case $\triangleleft\ e\ (b_1 \otimes b_2)$ fleshes out Shan and Ramsey's remark that "other types $\alpha$ such as $\alpha = \mathbb{R} \times \mathbb{R}$ can be handled by successive disintegration". Intuitively, because the term $e$ has the type of a pair, this case constrains the value of $e$ by constraining each component of the pair in turn. The equational derivation of this case uses the induction hypotheses for the two recursive calls to $\triangleleft$ and uses the specification of *fst* in Figure 19.
- Finally, the case $\triangleleft\ e\ (b_1 \oplus b_2)$ adds handling for disjoint sums (B3), exactly as defined and motivated in equation (28). Intuitively, because the term $e$ has a sum type, this case constrains the value of $e$ by first enforcing that the tag matches (**inl** vs **inr**) and then constraining the value under the tag. The equational derivation of this case depends on the semantics of sum types, whose values are tagged to ensure that the sum stays disjoint:

$\mathbf{do}\ \{h;\ \mathbf{let}\ t = e;\ M\}$
$=$ {specification of $\triangleright$ (Figure 19)}
$\triangleright\ e\ (\lambda v_0 h'.\ \mathbf{do}\ \{h';\ M\{t \mapsto v_0\}\})\ h$

$$
\begin{aligned}
&= \quad \{\text{semantics of } \textbf{let inl} \text{ and } \textbf{let inr}\} \\
&\quad \triangleright\, e\, \big(\lambda v_0 h'.\, \textbf{do}\, \{\textbf{let inl}\, x_1 = v_0;\, h';\, \textbf{let}\, t_1 = x_1;\, M\{t \mapsto \textbf{inl}\, t_1\}\} \\
&\qquad\qquad \oplus\, \textbf{do}\, \{\textbf{let inr}\, x_2 = v_0;\, h';\, \textbf{let}\, t_2 = x_2;\, M\{t \mapsto \textbf{inr}\, t_2\}\}\big)\, h \\
&\sqsupseteq \quad \{\text{induction hypotheses}\} \\
&\quad \triangleright\, e\, \big(\lambda v_0 h'.\, \textbf{do}\, \{\textbf{let inl}\, x_1 = v_0;\, t_1 \leftsquigarrow b_1;\, \lhd\, x_1\, b_1\, t_1\, \overline{M\{t \mapsto \textbf{inl}\, t_1\}}\, h'\} \\
&\qquad\qquad \oplus\, \textbf{do}\, \{\textbf{let inr}\, x_2 = v_0;\, t_2 \leftsquigarrow b_2;\, \lhd\, x_2\, b_2\, t_2\, \overline{M\{t \mapsto \textbf{inr}\, t_2\}}\, h'\}\big)\, h \\
&= \quad \{\text{semantics of } \oplus, \textbf{let inl}, \text{ and } \textbf{let inr}\} \\
&\quad \triangleright\, e\, \big(\lambda v_0 h'.\, \textbf{do}\, \{\textbf{let inl}\, x_1 = v_0;\, t \leftsquigarrow b_1 \oplus b_2;\, \textbf{let inl}\, t_1 = t;\, \lhd\, x_1\, b_1\, t_1\, \overline{M}\, h'\} \\
&\qquad\qquad \oplus\, \textbf{do}\, \{\textbf{let inr}\, x_2 = v_0;\, t \leftsquigarrow b_1 \oplus b_2;\, \textbf{let inr}\, t_2 = t;\, \lhd\, x_2\, b_2\, t_2\, \overline{M}\, h'\}\big)\, h \\
&= \quad \{\text{commutativity of } \triangleright \text{ (Figure 21) and } \oplus \text{ distributivity (17)}\} \\
&\quad \textbf{do}\, \big\{t \leftsquigarrow b_1 \oplus b_2;\, \triangleright\, e\, \big(\lambda v_0 h'.\, \textbf{do}\, \{\textbf{let inl}\, x_1 = v_0;\, \textbf{let inl}\, t_1 = t;\, \lhd\, x_1\, b_1\, t_1\, \overline{M}\, h'\} \\
&\qquad\qquad\qquad\qquad \oplus\, \textbf{do}\, \{\textbf{let inr}\, x_2 = v_0;\, \textbf{let inr}\, t_2 = t;\, \lhd\, x_2\, b_2\, t_2\, \overline{M}\, h'\}\big)\, h\big\} \\
&= \quad \{\text{specification of } \mathit{outl} \text{ and } \mathit{outr} \text{ (Figure 19)}\} \\
&\quad \textbf{do}\, \big\{t \leftsquigarrow b_1 \oplus b_2;\, \triangleright\, e\, \big(\lambda v_0 h'.\, \mathit{outl}\, v_0\, (\lambda e_1 h''.\, \textbf{do}\, \{\textbf{let inl}\, t_1 = t;\, \lhd\, e_1\, b_1\, t_1\, \overline{M}\, h''\})\, h' \\
&\qquad\qquad\qquad\qquad \oplus\, \mathit{outr}\, v_0\, (\lambda e_2 h''.\, \textbf{do}\, \{\textbf{let inr}\, t_2 = t;\, \lhd\, e_2\, b_2\, t_2\, \overline{M}\, h''\})\, h'\big)\, h\big\} \\
&= \quad \{\text{definition of } \lhd \text{ (Figure 14)}\} \\
&\quad \textbf{do}\, \{t \leftsquigarrow b_1 \oplus b_2;\, \lhd\, e\, (b_1 \oplus b_2)\, t\, \overline{M}\, h\}. \qquad (51)
\end{aligned}
$$

The fact that the new base-measure argument $b : \mathbb{B}\,\alpha$ is only inspected by $\lhd$, 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 $\lessdot$ ("constrain op") to encapsulate a repeated pattern in how *invertible* operations are handled. Figure 17 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. For example, the invertible $\textbf{sqrt}^+$ means to take the square root of a non-negative number, because Figure 17 defines $e \in\!dom\ \textbf{sqrt}^+ = 0 \leq e$ and $\textbf{sqrt}^+ \mathbin{@} e = \textbf{sqrt}\ e$. The other functions in Figure 17 are semantically specified and equationally derived as usual.

When $\lhd$ in Figure 14 encounters an operation that is invertible, it hands it to $\lessdot$; for example, the case $\lhd\ (\textbf{sqrt}\ e)$ is derived as follows:

$$
\begin{aligned}
&\quad \textbf{do}\, \{h;\, \textbf{let}\, t = \textbf{sqrt}\ e;\, M\} \\
&= \quad \{\text{assuming that the value of } e \text{ is in the domain of} \textbf{sqrt}\} \\
&\quad \textbf{do}\, \{h;\, \textbf{observe}\, (0 \leq e);\, \textbf{let}\, t = \textbf{sqrt}\ e;\, M\} \\
&= \quad \{\text{definitions of } \in\!dom \text{ and @ (Figure 17)}\} \\
&\quad \textbf{do}\, \{h;\, \textbf{observe}\, (e \in\!dom\, f);\, \textbf{let}\, t = f \mathbin{@} e;\, M\} \\
&\sqsupseteq \quad \{\text{specification of } \lessdot \text{ (Figure 14)}\} \\
&\quad \textbf{do}\, \{t \leftsquigarrow b;\, \lessdot\, \textbf{sqrt}^+\ e\ b\ t\ \overline{M}\ h\} \\
&= \quad \{\text{definition of } \lhd \text{ (Figure 14)}\} \\
&\quad \textbf{do}\, \{t \leftsquigarrow b;\, \lhd\, (\textbf{sqrt}\ e)\ b\ t\ \overline{M}\ h\}. \qquad (52)
\end{aligned}
$$

And when $\lhd$ in Figure 14 encounters an operation that is piecewise-invertible, it hands each invertible piece to $\lessdot$; for example, the case $\lhd\, e^2$ decomposes $e^2$ into two invertible pieces, namely

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

Domain

$(\in dom) : \lceil\mathbb{R}\rceil \to \mathsf{invertible} \to \lceil bool \rceil$

$e \in dom\ (\mathbf{mul}\ v) = \mathbf{true}$

$e \in dom\ (\mathbf{div}\ v) = \mathbf{true}$

$e \in dom\ \mathbf{sqr}^+ = 0 \le e$

$e \in dom\ \mathbf{sqr}^- = e < 0$

$e \in dom\ \mathbf{sqrt}^+ = 0 \le e$

$e \in dom\ \mathbf{sqrt}^- = 0 < e$

Apply invertible

$(@) : \mathsf{invertible} \to \lceil\mathbb{R}\rceil \to \lceil\mathbb{R}\rceil$

$\mathbf{mul}\ v\ @\ e = v \cdot e$

$\mathbf{div}\ v\ @\ e = v^{-1} \cdot e$

$\mathbf{sqr}^+\ @\ e = e^2$

$\mathbf{sqr}^-\ @\ e = e^2$

$\mathbf{sqrt}^+\ @\ e = \mathbf{sqrt}\ e$

$\mathbf{sqrt}^-\ @\ e = -\mathbf{sqrt}\ e$

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

$inv : \mathsf{invertible} \to \mathsf{invertible}$

$inv\ (\mathbf{mul}\ v) = \mathbf{div}\ v$

$inv\ (\mathbf{div}\ v) = \mathbf{mul}\ v$

$inv\ \mathbf{sqr}^+ = \mathbf{sqrt}^+$

$inv\ \mathbf{sqr}^- = \mathbf{sqrt}^-$

$inv\ \mathbf{sqrt}^+ = \mathbf{sqr}^+$

$inv\ \mathbf{sqrt}^- = \mathbf{sqr}^-$

Range

$(\in rng) : \lceil\mathbb{R}\rceil \to \mathsf{invertible} \to \lceil bool \rceil$

$e \in rng\ f = e \in dom\ (inv\ f)$

Apply derivative

$diff : \mathsf{invertible} \to \lfloor\mathbb{R}\rfloor \to \lfloor\mathbb{R}\rfloor$

$\boxed{diff\ f\ v = \left.\dfrac{d(f\ @\ x)}{dx}\right|_{x=v}}$

$diff\ (\mathbf{mul}\ v)\ \_ = v$

$diff\ (\mathbf{div}\ v)\ \_ = v^{-1}$

$diff\ \mathbf{sqr}^+\ v = 2 \cdot v \qquad diff\ \mathbf{sqrt}^+\ v = (2 \cdot \mathbf{sqrt}\ v)^{-1}$

$diff\ \mathbf{sqr}^-\ v = 2 \cdot v \qquad diff\ \mathbf{sqrt}^-\ v = (-2 \cdot \mathbf{sqrt}\ v)^{-1}$

Fig. 17. Invertible functions and operations on them

squaring a non-negative number and squaring a negative number:

$$
\begin{aligned}
& \mathbf{do}\ \{h; \mathbf{let}\ t = e^2; M\} \\
= \ & \{\oplus \text{ distributivity (17) and (18); exactly one of } 0 \le e \text{ and } e < 0 \text{ is } \mathbf{true}\} \\
& \mathbf{do}\ \{h; \mathbf{observe}\ (0 \le e); \mathbf{let}\ t = e^2; M\} \oplus \mathbf{do}\ \{h; \mathbf{observe}\ (e < 0); \mathbf{let}\ t = e^2; M\} \\
\sqsupseteq \ & \{\text{definitions of } \in dom \text{ and } @ \text{ (Figure 17) and specification of } \lessdot \text{ (Figure 14)}\} \\
& \mathbf{do}\ \{t \leftsquigarrow b; \lessdot \mathbf{sqr}^+\ e\ b\ t\ \overline{M}\ h\} \oplus \mathbf{do}\ \{t \leftsquigarrow b; \lessdot \mathbf{sqr}^-\ e\ b\ t\ \overline{M}\ h\} \\
= \ & \{\oplus \text{ distributivity (17)}\} \\
& \mathbf{do}\ \{t \leftsquigarrow b; \lessdot \mathbf{sqr}^+\ e\ b\ t\ \overline{M}\ h \oplus \lessdot \mathbf{sqr}^-\ e\ b\ t\ \overline{M}\ h\} \\
= \ & \{\text{definition of } \lhd \text{ (Figure 14)}\} \\
& \mathbf{do}\ \{t \leftsquigarrow b; \lhd e^2\ b\ t\ \overline{M}\ h\}.
\end{aligned} \tag{53}
$$

Third, even though the Lebesgue measure is the only base measure over $\mathbb{R}$ handled so far, we localize our reasoning about it to three new auxiliary functions, to help us add more base measures over $\mathbb{R}$ in Section 5. The functions *jacobian* and *reparam* reparametrize a base measure over $\mathbb{R}$ by an invertible and produce a Jacobian factor and a new base measure; these functions are used by $\lessdot$. The function $\div$ ("divide") finds a density of one base measure over $\mathbb{R}$ with respect to another; it is used by $\lll$. We now turn to these three functions. Figure 18 shows their informal types, semantic specifications (boxed), and implementations.

*4.2.1 Reparametrizing base measures.* Many realistic models invoke deterministic mathematical operations over $\mathbb{R}$, such as curving a random student grade by taking its square root, combining random particle momenta by summing them [Afshar et al. 2016], or just converting a random

Reparametrize a base measure with a Jacobian

$$\begin{aligned}
&\mathit{jacobian} : \mathsf{invertible} \to \mathbb{B}\,\mathbb{R} \to \lfloor \mathbb{R} \rfloor \to \lfloor \mathbb{R} \rfloor \\
&\mathit{reparam} : \mathsf{invertible} \to \mathbb{B}\,\mathbb{R} \to \mathbb{B}\,\mathbb{R} \\
&\mathit{jacobian}\ f\ \mathbf{lebesgue} = \lambda v.\, |\mathit{diff}\ (\mathit{inv}\ f)\ v| \\
&\mathit{reparam}\ f\ \mathbf{lebesgue} = \mathbf{lebesgue}
\end{aligned}$$

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

Divide base measures

$$\begin{aligned}
&(\div) : \mathbb{B}\,\mathbb{R} \to \mathbb{B}\,\mathbb{R} \to \{\lfloor \mathbb{R} \rfloor \to \mathbb{M}\,\mathbb{1}\} \\
&\mathbf{lebesgue} \div \mathbf{lebesgue} = \lambda v.\, \mathbf{return}\ ()
\end{aligned}$$

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

Fig. 18. Operations on the Lebesgue base measure

measurement from Celsius to Fahrenheit. To constrain an expression that invokes such operations, $\lhd$ calls $\lessdot$, which in turn calls *jacobian* and *reparam*, defined in Figure 18.

To see how handling deterministic mathematical operations amounts to reparametrizing a base measure and changing an integration variable, suppose we want the density of the distribution that is like **normal** 0 1 but with its outcome multiplied by 3. According to equation (9), we seek $\kappa : \mathbb{R} \to \mathbb{R}_+$ such that

$$(\mathbf{normal}\ 0\ 1)(\lambda x.\, f(3 \cdot x)) = \mathbf{lebesgue}(\lambda y.\, \kappa(y) \cdot f(y)) \tag{54}$$

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

$$\int_{\mathbb{R}} (\mathbf{dnorm}\ 0\ 1)(x) \cdot f(3 \cdot x)\, dx = \int_{\mathbb{R}} \kappa(y) \cdot f(y)\, dy. \tag{55}$$

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

$$\int_{\mathbb{R}} (\mathbf{dnorm}\ 0\ 1)(x) \cdot f(3 \cdot x)\, dx = \int_{\mathbb{R}} (\mathbf{dnorm}\ 0\ 1)(y/3) \cdot f(y) \cdot (dx/dy)\, dy, \tag{56}$$

in which $dx/dy = 1/3$. Hence, we find the density $\kappa = \lambda y.\, (\mathbf{dnorm}\ 0\ 1)(y/3) \cdot (1/3)$.

We can use the disintegrator to find the same density, starting with a top-level call to *check*:

$$\begin{aligned}
&\mathit{check}\ (\mathbf{do}\ \{x \leftsquigarrow \mathbf{normal}\ 0\ 1;\ \mathbf{return}\ (3 \cdot x, ())\})\ \mathbf{lebesgue}\ y \\
=\ &\{\text{definitions of } \mathit{check} \text{ (Figure 14) and } \gg\!\!> \text{ and } \rhd \text{ (Figure 19)}\} \\
&\lhd (3 \cdot x)\ \mathbf{lebesgue}\ y\ \overline{(\mathbf{return}\ ())}\ [x \leftsquigarrow \mathbf{normal}\ 0\ 1] \\
=\ &\{\text{definitions of } \lhd \text{ (Figure 14) and } \rhd \text{ (Figure 19)}\} \\
&\lessdot (\mathbf{mul}\ 3)\ x\ \mathbf{lebesgue}\ y\ \overline{(\mathbf{return}\ ())}\ [x \leftsquigarrow \mathbf{normal}\ 0\ 1].
\end{aligned} \tag{57}$$

Following the specifications of *check*, $\lhd$, and $\lessdot$ in Figure 14, all three terms in (57) seek to rewrite the heap $[x \leftsquigarrow \mathbf{normal}\ 0\ 1;\ \mathbf{let}\ y = 3 \cdot x]$ to an equivalent heap of the form $[y \leftsquigarrow \mathbf{lebesgue}; \ldots]$. To invert $y = 3 \cdot x$ is to $\mathbf{let}\ x = y/3$. However, to preserve semantics, the result cannot just be $[y \leftsquigarrow \mathbf{lebesgue};\ \mathbf{let}\ x = y/3]$, because the distribution over $x$ would not be **normal** 0 1. Instead, a correct result is

$$[y \leftsquigarrow \mathbf{lebesgue};\ \mathbf{factor}\ 1/3;\ \mathbf{let}\ x = y/3;\ \mathbf{factor}\ (\mathbf{dnorm}\ 0\ 1)(x)]. \tag{58}$$

To this end, $\lessdot$ calls *jacobian* (**mul** 3). The *jacobian* function differentiates $\mathit{inv}\ (\mathbf{mul}\ 3) = \mathbf{div}\ 3$ and returns $\lambda y.\, 1/3$, hence computing the Jacobian factor necessary to change the integration variable.

This **factor** $1/3$ is emitted by $\lessdot$ before constraining the value of $x$ to be $y/3$:

$$\begin{aligned}
&\lessdot\ (\mathbf{mul}\ 3)\ x\ \mathbf{lebesgue}\ y\ \overline{(\mathbf{return}\ ())}\ [x \leftsquigarrow \mathbf{normal}\ 0\ 1] \\
=\ & \{\text{definition of } \lessdot \text{ (Figure 14)}\} \\
& \mathbf{do}\ \{\mathbf{factor}\ 1/3;\ \lhd\ x\ \mathbf{lebesgue}\ (y/3)\ \overline{(\mathbf{return}\ ())}\ [x \leftsquigarrow \mathbf{normal}\ 0\ 1]\}.
\end{aligned} \tag{59}$$

More generally, the implementation of $\lessdot$ in Figure 14 is derived from the specifications of $\lessdot$ and *jacobian* and *reparam*:

$$\begin{aligned}
& \mathbf{do}\ \{h;\ \mathbf{observe}\ (e \in dom\ f);\ \mathbf{let}\ t = f\ @\ e;\ M\} \\
=\ & \{\text{monad left-identity law}\} \\
& \mathbf{do}\ \{h;\ \mathbf{let}\ x = e;\ \mathbf{observe}\ (x \in dom\ f);\ \mathbf{let}\ t = f\ @\ x;\ M\} \\
\sqsupseteq\ & \{\text{specification of } \lhd \text{ (Figure 14)}\} \\
& \mathbf{do}\ \{x \leftsquigarrow reparam\ f\ b;\ \lhd\ e\ (reparam\ f\ b)\ x\ \overline{(\mathbf{do}\ \{\mathbf{observe}\ (x \in dom\ f);\ \mathbf{let}\ t = f\ @\ x;\ M\})}\ h\} \\
=\ & \{\text{commutativity of } \lhd \text{ (Figure 21)}\} \\
& \mathbf{do}\ \{x \leftsquigarrow reparam\ f\ b;\ \mathbf{observe}\ (x \in dom\ f);\ \mathbf{let}\ t = f\ @\ x;\ \lhd\ e\ (reparam\ f\ b)\ x\ \overline{M}\ h\} \\
=\ & \{\text{specification of } jacobian \text{ and } reparam \text{ (Figure 18); monad laws}\} \\
& \mathbf{do}\ \{x \leftsquigarrow \mathbf{do}\ \{t \leftsquigarrow b;\ \mathbf{observe}\ (t \in rng\ f);\ \mathbf{factor}\ (jacobian\ f\ b\ t);\ \mathbf{return}\ (inv\ f\ @\ t)\}; \\
& \qquad \mathbf{let}\ t = f\ @\ x;\ \lhd\ e\ (reparam\ f\ b)\ x\ \overline{M}\ h\} \\
=\ & \{f\ @\ (inv\ f\ @\ t) = t \text{ because } t \in rng\ f;\ \text{monad laws}\} \\
& \mathbf{do}\ \{t \leftsquigarrow b;\ \mathbf{observe}\ (t \in rng\ f);\ \mathbf{factor}\ (jacobian\ f\ b\ t);\ \lhd\ e\ (reparam\ f\ b)\ (inv\ f\ @\ t)\ \overline{M}\ h\} \\
=\ & \{\text{definition of } \lessdot \text{ (Figure 14)}\} \\
& \mathbf{do}\ \{t \leftsquigarrow b;\ \lessdot\ f\ e\ b\ t\ \overline{M}\ h\}.
\end{aligned} \tag{60}$$

Because $\lessdot\ f$ emits a *jacobian* **factor** outside the scope of the heap, the invertible $f$ must not use any variable bound in the heap. The grammar of invertibles in Figure 17 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 (Figure 15) shows that no head normal form of type $\mathbb{R}$ uses any variable bound in the heap. This requirement is established by partial evaluation (Section 4.3).

*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 **fst** $v$ in *check* in Figure 14, 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 $\nu$. This reduction takes place in Figure 14, as *check* uses $\lhd$, and $\lhd$ in turn uses $\lll$:

- In Figure 14, the case $\lll$ **lebesgue** $b$ faces the problem of finding a density of **lebesgue** with respect to $b$.
- In the simplified transcript in Figure 16, the call to $\lll$ (**normal** 0 1) **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 $\lll\ \xi\ \nu$ using Proposition 2.10(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 $\lll$ 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 $\ll$ **lebesgue**). It then remains to find a density of the base measure $\mu$ with respect to the base measure $\nu$, and that is the job of the auxiliary function $\div$, defined in Figure 18. Although the job of $\div$ is to find a density, its return type is not $\mathbb{R}_+$ but the isomorphic type $\mathbb{M}\,\mathbb{1}$.

Because the only base measure over $\mathbb{R}$ so far is **lebesgue**, the implementation of $\div$ so far is extremely simple. The constant-1 function is a density of **lebesgue** with respect to **lebesgue**; accordingly, $\div$ returns **return** $() : \mathbb{M}\,\mathbb{1}$, which is isomorphic to $1 : \mathbb{R}_+$.

The case $\ll$ **lebesgue** $b$ in Figure 14 is derived from the specification of $\div$ in Figure 18:

$$
\begin{aligned}
& \textbf{do}\ \{h;\ t \leftsquigarrow \textbf{lebesgue};\ M\} \\
= \quad & \{\text{commutativity (40)}\} \\
& \textbf{do}\ \{t \leftsquigarrow \textbf{lebesgue};\ h;\ M\} \\
\sqsupseteq \quad & \{\text{specification of } \div \text{ (Figure 18); monad laws}\} \\
& \textbf{do}\ \{t \leftsquigarrow b;\ () \leftsquigarrow (\textbf{lebesgue} \div b)\ t;\ h;\ M\} \\
= \quad & \{\text{definition of } \ll \text{ (Figure 14)}\} \\
& \textbf{do}\ \{t \leftsquigarrow b;\ \ll \textbf{lebesgue}\ b\ t\ \overline{M}\ h\}. \qquad (61)
\end{aligned}
$$

Unlike the implementation of $\div$, the implementation of $\ll$ does not inspect the base measure $b$ or assume that it is **lebesgue**. This indifference helps us add more base measures over $\mathbb{R}$ in Section 5.

### 4.3 Partial evaluation

The functions $\triangleright$ ("evaluate") and $\gg$ ("perform"), used in Figure 14, perform *lazy partial evaluation* [Jørgensen 1992; Fischer et al. 2008]. Figure 19 shows their informal types, semantic specifications (boxed), and implementations. These functions are standard and unchanged from Shan and Ramsey's disintegrator—they do not even take a new base-measure argument. We describe them first intuitively then technically.

Intuitively, the job of partial evaluation [Jones et al. 1993] is to run a program symbolically and represent its result as a head normal form, without taking any concrete inputs or making any random choices. Because running a program is part of the job, the partial evaluator is much like a definitional interpreter [Reynolds 1972]. Concretely, to understand Figure 19, one can start with an interpreter that *evaluates* arithmetic expressions, shown in Figure 20(a). The following steps turn Figure 20(a) into Figure 19:

(1) Handle random choices in the input term by making random choices. Because random choices are expressed monadically, this step is to add an interpreter that *performs* monadic expressions, shown in Figure 20(b).
(2) Handle bound variables in the input term *lazily*, by maintaining a heap that binds them to monadic expressions. For example, *perform* (**do** $\{x \leftsquigarrow \textbf{normal}\ 0\ 1;\ M\}$) does not pick a random number $x$ right away. Instead, it adds the binding $x \leftsquigarrow$ **normal** 0 1 to the heap, then proceeds to *perform* $M$. Performing $M$ may *evaluate* the variable $x$—for example, if $M =$ **return** $x^2$ (but not if $M =$ **return** (**fst** $(0, x)$)). Then and only then, the partial evaluator picks a random number and replaces the heap binding $x \leftsquigarrow$ **normal** 0 1 by **let** $x =$ the number picked.
(3) Handle unbound variables in the input term by producing symbolic results. For example, if the variable $z$ is not bound, then *evaluate* $z$ and *evaluate* (**sqrt** $z$) just return the terms $z$ and **sqrt** $z$ as results. To this end, overload the *sqrt* function called in Figure 20(a), to construct a term as a last resort. This overloading turns *sqrt* into a *smart constructor*.
(4) Turn the functions *evaluate* and *perform* into continuation-passing style, so as to generate code without making random choices. In particular, the standard *call-by-value continuation-passing-style transformation* [Plotkin 1975] turns Figure 20(a) into Figure 20(c), which is equivalent

Evaluate $\boxed{\textbf{do}\ \{h; \textbf{let}\ x = e; M\} \quad = \quad \triangleright\, e\ (\lambda v.\ \overline{M\{x \mapsto v\}})\ h}$

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

$\triangleright\ v \qquad k\ h \qquad = k\ v\ h$ where $v$ is in head normal form

$\triangleright\ (\textbf{sqrt}\ e) \quad k\ h \qquad = \triangleright\, e\ \big(\lambda v_0.\ k\ (\mathit{sqrt}\ v_0)\big)\ h$

$\triangleright\ e^{\mathbf{2}} \qquad k\ h \qquad = \triangleright\, e\ \big(\lambda v_0.\ k\ v_0^2\big)\ h$

$\triangleright\ (e_1 \cdot e_2) \quad k\ h \qquad = \triangleright\, e_1\ \big(\lambda v_1.\ \triangleright\, e_2\ \big(\lambda v_2.\ k\ (v_1 \cdot v_2)\big)\big)\ h$

$\triangleright\ (e_1 < e_2)\ k\ h \qquad = \triangleright\, e_1\ \big(\lambda v_1.\ \triangleright\, e_2\ \big(\lambda v_2.\ k\ (v_1 < v_2)\big)\big)\ h$

$\triangleright\ (\textbf{fst}\ e) \quad k\ h \qquad = \triangleright\, e\ \big(\lambda v_0.\ \triangleright\ (\mathit{fst}\ v_0)\ k\big)\ h$ unless $e$ is atomic

$\triangleright\ (\textbf{snd}\ e) \quad k\ h \qquad = \triangleright\, e\ \big(\lambda v_0.\ \triangleright\ (\mathit{snd}\ v_0)\ k\big)\ h$ unless $e$ is atomic

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

$\triangleright\ x\ k\ [h_1; \textbf{let inl}\ x = e; h_2] = \triangleright\, e\ \big(\lambda v_0.\ \mathit{outl}\ v_0\ (\lambda e_0.\ \triangleright\, e_0\ (\lambda v.\ \overline{[\textbf{let}\ x = v; h_2]} \fatsemi k\ v))\big)\ h_1$

$\triangleright\ x\ k\ [h_1; \textbf{let inr}\ x = e; h_2] = \triangleright\, e\ \big(\lambda v_0.\ \mathit{outr}\ v_0\ (\lambda e_0.\ \triangleright\, e_0\ (\lambda v.\ \overline{[\textbf{let}\ x = v; h_2]} \fatsemi k\ v))\big)\ h_1$

Perform $\boxed{\textbf{do}\ \{h; x \leftsquigarrow m; M\} \quad = \quad \gg\!= m\ (\lambda v.\ \overline{M\{x \mapsto v\}})\ h}$

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

$\gg\!= u \qquad k\ h = \textbf{do}\ \{z \leftsquigarrow u; \qquad k\ z\ h\}$ where $u$ is atomic, $z$ is fresh

$\gg\!= \textbf{lebesgue} \quad k\ h = \textbf{do}\ \{z \leftsquigarrow \textbf{lebesgue}; k\ z\ h\}$ where $z$ is fresh

$\gg\!= (\textbf{return}\ e) \quad k\ h = \triangleright\ e\ k\ h$

$\gg\!= \textbf{fail} \qquad k\ h = \textbf{fail}$

$\gg\!= (m_1 \oplus m_2) \quad k\ h = \gg\!= m_1\ k\ h \oplus \gg\!= m_2\ k\ h$

$\gg\!= (\textbf{do}\ \{g; m\})\ k\ h = \gg\!= m\ k\ [h; g]$

$\gg\!= e \qquad k\ h = \triangleright\, e\ (\lambda m.\ \gg\!= m\ k)\ h$ where $e$ is not in head normal form

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

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

$\mathit{fst}\ (e_1, e_2) = e_1 \qquad \mathit{outr}\ (\textbf{inl}\ e)\ k\ h = \textbf{fail}$

$\mathit{fst}\ e \quad = \textbf{fst}\ e \qquad \mathit{outr}\ (\textbf{inr}\ e)\ k\ h = k\ e\ h$

$\mathit{outr}\ u \qquad k\ h = \textbf{do}\ \{\textbf{let inr}\ x = u; k\ x\ h\}$

Fig. 19. Shan and Ramsey's operations for partial evaluation

to the corresponding cases of $\triangleright$ in Figure 19. Continuation passing is necessary because, despite steps 1 and 2 above, the partial evaluator should not actually make random choices. Rather, the partial evaluator should emit code that makes random choices. For example, in the case $\gg\!=$ **lebesgue** in Figure 19, the right-hand side does not draw any actual number $z$ from **lebesgue**. Rather, it generates a fresh variable $z$ and feeds it to the continuation $k$ under the scope of an emitted binding $z \leftsquigarrow$ **lebesgue** [Bondorf 1992; Lawall and Danvy 1994].

Technically, the job of partial evaluation is to bring a given term into head normal form (indicated by the meta-type $\lfloor \alpha \rfloor$ and the metavariable $v$) while preserving its meaning. This job serves two purposes in automatic disintegration, each illustrated in the sample run in Figure 16.

$$
\begin{aligned}
&\mathit{evaluate}\ r &&= r\\
&\mathit{evaluate}\ (\mathbf{sqrt}\ e) &&= \mathit{sqrt}\ (\mathit{evaluate}\ e)\\
&\mathit{evaluate}\ e^{\mathbf{2}} &&= (\mathit{evaluate}\ e)^2\\
&\mathit{evaluate}\ (e_1 \centerdot e_2) &&= \mathit{evaluate}\ e_1 \cdot \mathit{evaluate}\ e_2
\end{aligned}
$$

(a) A metacircular interpreter that evaluates arithmetic expressions

$$
\begin{aligned}
&\mathit{perform}\ \mathbf{lebesgue} &&= \mathbf{lebesgue}\\
&\mathit{perform}\ (\mathbf{return}\ e) &&= \mathbf{return}\ (\mathit{evaluate}\ e)\\
&\mathit{perform}\ \mathbf{fail} &&= \mathbf{fail}\\
&\mathit{perform}\ (m_1 \mathbin{⏀} m_2) &&= \mathit{perform}\ m_1 \mathbin{⏀} \mathit{perform}\ m_2
\end{aligned}
$$

(b) A metacircular interpreter that performs monadic expressions

$$
\begin{aligned}
&\mathit{evaluate}\ r &&k = k\ r\\
&\mathit{evaluate}\ (\mathbf{sqrt}\ e) &&k = \mathit{evaluate}\ e\ \left(\lambda v_0.\ k\ (\mathit{sqrt}\ v_0)\right)\\
&\mathit{evaluate}\ e^{\mathbf{2}} &&k = \mathit{evaluate}\ e\ \left(\lambda v_0.\ k\ v_0^2\right)\\
&\mathit{evaluate}\ (e_1 \centerdot e_2) &&k = \mathit{evaluate}\ e_1\ \left(\lambda v_1.\ \mathit{evaluate}\ e_2\ \left(\lambda v_2.\ k\ (v_1 \cdot v_2)\right)\right)
\end{aligned}
$$

(c) A metacircular interpreter that evaluates arithmetic expressions in continuation-passing style

Fig. 20. Evaluating arithmetic expressions and performing random choices

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 $\bowtie$ in Figure 14. For example, on lines 2 and 3 of Figure 16, 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 $\lhd$ and the second component $3 + 2 \cdot x$ can be inferred by **return**. This use of partial evaluation is justified by the last step in the proof of Theorem 4.2, which invokes the specification of $\bowtie$ in Figure 19.

Second, given a binary operation such as $e_1 \centerdot e_2$ to constrain, $\lhd$ needs to build an invertible operation like **mul** $e_1$ or **mul** $e_2$, but to support reparametrization, a term used to build an invertible must be a head normal form (Section 4.2.1). Partial evaluation establishes this requirement; for example, on lines 5 and 6 of Figure 16, partial evaluation turns the heap-bound variable $y$ into the non-heap-bound variable $z$, so that an invertible can be built using the head normal form $z$ and passed to $\lessdot$. In Figure 14, to satisfy this requirement, the case $\lhd\ (e_1 \centerdot e_2)$ does not build an invertible by **mul** $e_1$ or **mul** $e_2$. Rather, it calls $\rhd\ e_1$; this case is derived as follows.

$$
\begin{aligned}
&\quad \mathbf{do}\ \{h;\ \mathbf{let}\ t = e_1 \centerdot e_2;\ M\}\\
&= \quad \{\text{monad left-identity law; } \mathbf{observe\ true} \text{ does nothing}\}\\
&\quad \mathbf{do}\ \{h;\ \mathbf{let}\ x = e_1;\ \mathbf{observe\ true};\ \mathbf{let}\ t = x \centerdot e_2;\ M\}\\
&= \quad \{\text{specification of } \rhd \text{ (Figure 19)}\}\\
&\quad \rhd\ e_1\ (\lambda v_1 h'.\ \mathbf{do}\ \{h';\ \mathbf{observe\ true};\ \mathbf{let}\ t = v_1 \centerdot e_2;\ M\})\ h\\
&\sqsupseteq \quad \{\text{definitions of } \in\mathit{dom} \text{ and } @ \text{ (Figure 17) and specification of } \lessdot \text{ (Figure 14)}\}\\
&\quad \rhd\ e_1\ (\lambda v_1 h'.\ \mathbf{do}\ \{t \leftsquigarrow b;\ \lessdot\ (\mathbf{mul}\ v_1)\ e_2\ b\ t\ \overline{M}\ h'\})\ h\\
&= \quad \{\text{commutativity of } \rhd \text{ (Figure 21)}\}\\
&\quad \mathbf{do}\ \{t \leftsquigarrow b;\ \rhd\ e_1\ (\lambda v_1.\ \lessdot\ (\mathbf{mul}\ v_1)\ e_2\ b\ t\ \overline{M})\ h\}\\
&= \quad \{\text{definition of } \lhd \text{ (Figure 14)}\}\\
&\quad \mathbf{do}\ \{t \leftsquigarrow b;\ \lhd\ (e_1 \centerdot e_2)\ b\ t\ \overline{M}\ h\}.
\end{aligned}
\tag{62}
$$

(The other nondeterministic choice, calling $\triangleright\, e_2$, is derived similarly.)

To preserve semantics, partial evaluation may emit bindings into the output as well as accumulate bindings onto the heap. Both possibilities are illustrated in the sample run in Figure 16. Between lines 6 and 7 of Figure 16, the binding $z \leftsquigarrow \mathbf{normal}\ 0\ 1$ is emitted into the output; this emission is the work of the case $\gg\!=$ **lebesgue** in Figure 19, derived as follows.

$$\begin{aligned}
&\mathbf{do}\ \{h;\ x \leftsquigarrow \mathbf{lebesgue};\ M\}\\
=\ &\{\text{commutativity (40) and variable renaming}\}\\
&\mathbf{do}\ \{z \leftsquigarrow \mathbf{lebesgue};\ h;\ M\{x \mapsto z\}\}\\
=\ &\{\text{definition of } \gg\!= \text{ (Figure 19)}\}\\
&\gg\!= \mathbf{lebesgue}\ (\lambda v.\ \overline{M\{x \mapsto v\}})\ h.
\end{aligned} \tag{63}$$

Between lines 2 and 3, the bindings $y \leftsquigarrow \mathbf{normal}\ 0\ 1$; $x \leftsquigarrow \mathbf{normal}\ 0\ 1$ are accumulated onto the heap; this accumulation is the work of the case $\gg\!= (\mathbf{do}\ \{g;\ m\})$ in Figure 19, derived as follows.

$$\begin{aligned}
&\mathbf{do}\ \{h;\ x \leftsquigarrow \mathbf{do}\ \{g;\ m\};\ M\}\\
=\ &\{\text{monad associativity law}\}\\
&\mathbf{do}\ \{h;\ g;\ x \leftsquigarrow m;\ M\}\\
=\ &\{\text{induction hypothesis}\}\\
&\gg\!= m\ (\lambda v.\ \overline{M\{x \mapsto v\}})\ [h;\ g]\\
=\ &\{\text{definition of } \gg\!= \text{ (Figure 19)}\}\\
&\gg\!= (\mathbf{do}\ \{g;\ m\})\ (\lambda v.\ \overline{M\{x \mapsto v\}})\ h.
\end{aligned} \tag{64}$$

(The case $\ll\!= (\mathbf{do}\ \{g;\ m\})$ in Figure 14 is derived similarly.) Accumulated heap bindings are used lazily to evaluate variables in the three cases $\triangleright\, x$ in Figure 19; the first case is derived as follows.

$$\begin{aligned}
&\mathbf{do}\ \{h_1;\ x \leftsquigarrow m;\ h_2;\ \mathbf{let}\ y = x;\ M\}\\
=\ &\{\text{specification of } \gg\!= \text{ (Figure 19)}\}\\
&\gg\!= m\ (\lambda v.\ \overline{(\mathbf{do}\ \{h_2;\ \mathbf{let}\ y = x;\ M\})\{x \mapsto v\}})\ h_1\\
=\ &\{\text{semantics of } \mathbf{let} \text{ and substitution}\}\\
&\gg\!= m\ (\lambda v.\ \overline{\mathbf{do}\ \{\mathbf{let}\ x = v;\ h_2;\ M\{y \mapsto v\}\}})\ h_1\\
=\ &\{\text{definition of } \triangleright \text{ (Figure 19)}\}\\
&\triangleright\, x\ (\lambda v.\ \overline{M\{y \mapsto v\}})\ [h_1;\ x \leftsquigarrow m;\ h_2].
\end{aligned} \tag{65}$$

(The two other cases of $\triangleright\, x$, and the three cases $\triangleleft\, x$ in Figure 14, are derived similarly.)

Partial evaluation is assisted by smart constructors such as *sqrt*, $^2$, $\cdot$, $<$, *fst*, and *outr*. As the boxed specifications near the bottom of Figure 19 show, these meta-functions are semantically equivalent to **sqrt**, $^{\mathbf{2}}$, $\boldsymbol{\cdot}$, $\mathbf{<}$, **fst**, and **let inr** in core Hakaru syntax. The soundness of the disintegrator depends on this equivalence; for example, the derivation (51) for disjoint sums relies on the specification of *outl* and *outr*. However, unlike ordinary core Hakaru syntax, smart constructors reduce away constructor applications if possible. For example, $sqrt\,9 = 3$, $fst\,(e_1, e_2) = e_1$, and $outr\,(\mathbf{inr}\,e)\,k\,h = k\,e\,h$. In case of pattern mismatch, the call *outr* (**inl** *e*) *k h* is smart to return the zero measure **fail** without invoking *k* at all; because invoking *k* might produce $\emptyset$, this smart helps the disintegrator succeed more often.

Commutativity

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

Associativity

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

Parametricity

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

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

Fig. 21. Inductively proven properties of the four workhorse functions $\lhd, \lll, \rhd, \ggg$

### 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 four workhorse functions $\lhd, \lll, \rhd, \ggg$. These properties are specified in Figure 21.

(1) *Commutativity* says that emitting a binding $g$ and then constraining or evaluating an expression $e$ or $m$ is the 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 it does not matter which way we disambiguate the following phrase using square brackets: [[constraining or evaluating $e$ or $m$ with the continuation $c$ or $k$] followed by the final action $M$] is the same as [constraining or evaluating $e$ or $m$ with the continuation [$c$ 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 constraining functions $\lhd$ and $\lll$ treat their arguments $v$ and $c$ parametrically. That is, these functions never inspect those arguments, so they commute with substitution on those arguments.

The definition of the four workhorse functions $\lhd, \lll, \rhd, \ggg$ uses recursion and nondeterminism, so assembling all the equational derivations into an overall soundness proof is not entirely trivial. Technically, Figures 14 and 19 define a function $F$ from 4-tuples of functions $(\lhd_n, \lll_n, \rhd_n, \ggg_n)$ to 4-tuples of functions $(\lhd_{n+1}, \lll_{n+1}, \rhd_{n+1}, \ggg_{n+1})$, but the step-indexing subscripts $n$ in the right-hand sides and $n+1$ in the left-hand sides are elided. Moreover, as the types show, each of these step-indexed functions returns a set of terms and takes a continuation that also returns a set of terms. These sets of terms are partially ordered by the subset relation, and these functions (including continuations) returning a set of terms are then partially ordered pointwise.

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

words, as the pointwise union of all the step-indexed 4-tuples. Each property of the four functions (commutativity, associativity, parametricity, and then the semantic specifications) is then proven by induction on the step-index $n$.

## 5 A RESTRICTED BASE-CHECKING DISINTEGRATOR

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

$$\begin{aligned}
&\mathbf{lebesgue} \oplus \mathbf{return}\ 0 \oplus \mathbf{return}\ 1 && : \mathbb{M}\,\mathbb{R} && \text{in (14),}\\
&\mathbf{lebesgue} \otimes\!\!= \lambda x.\ \mathbf{return}\ x && : \mathbb{M}\,\mathbb{R}^2 && \text{(a diagonal), or}\\
&\mathbf{lebesgue}^2 \otimes\!\!= \lambda x.\ \big((\mathbf{lebesgue} \oplus \mathbf{return}\ (\mathbf{fst}\ x)) \otimes && && \\
&\qquad (\mathbf{lebesgue} \oplus \mathbf{return}\ (\mathbf{snd}\ x))\big) && : \mathbb{M}\,(\mathbb{R}^2)^2 && \text{in (27).}
\end{aligned}$$

In contrast, the base-measure language in Figure 15 only allows independent products and disjoint sums of Lebesgue and unit measures. Accordingly, we generalize the base-measure language in two ways.

(B1) Base measures over $\mathbb{R}$ have the form $\mathbf{mix}\ l\ \wr e_1, \ldots \int$, distinct from any measure term in the term language. Here $\wr e_1, \ldots \int$ is a bag of real terms whose Dirac measures are mixed together, and $l$ is a meta-level Boolean indicating whether the Lebesgue measure is mixed in as well. In other words, the base measure $\mathbf{mix}\ \mathrm{ff}\ \wr e_1, \ldots \int$ means $\mathbf{return}\ e_1 \oplus \cdots$, and the base measure $\mathbf{mix}\,\mathrm{tt}\,\wr e_1, \ldots \int$ means $\mathbf{lebesgue} \oplus \mathbf{return}\, e_1 \oplus \cdots$. Hence **lebesgue** can be expressed as $\mathbf{mix}\,\mathrm{tt}\,\wr\int$, and $\mathbf{return}\ x$ can be expressed as $\mathbf{mix}\ \mathrm{ff}\ \wr x \int$.

(B2) Independent products $b_1 \otimes b_2$ become dependent products $b_1 \otimes\!\!= \lambda x.\ b_2$, in which $x$ can appear in $b_2$, namely in a bag of real terms.

These changes are summarized at the top of Figure 22. The rest of the figure updates the functions $\lhd$, $\lll$, *jacobian*, *reparam*, and $\div$ to handle the new base measures. Thanks to the groundwork laid in Section 4, the semantic specifications for these functions remain the same, and only cases corresponding to the new base measures need to be added. As in Section 4, these cases are derived and proven from those semantic specifications by equational reasoning.

The new cases of $\lhd$ take advantage of the extended base-measure language. Constraining the second element of a pair (*snd e*) now uses a base measure $b_2\{x \mapsto \mathit{fst}\ v\}$ that can depend on *fst v*, which is what the first element (*fst e*) was constrained to be. And constraining a Dirac measure (at $u$ or $r$) now passes the job to $\div$ instead of returning $\emptyset$ right away. In fact, now the only function that can return $\emptyset$ 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 $\mathbf{mix}\ \mathrm{tt}\ \wr\int \div \mathbf{mix}\ \mathrm{tt}\ \wr 4\int$ and $\mathbf{mix}\ \mathrm{ff}\ \wr 4\int \div \mathbf{mix}\ \mathrm{tt}\ \wr 4\int$. We show the former case, which means dividing **lebesgue** by $\mathbf{lebesgue} \oplus \mathbf{return}\ 4$:

$$\begin{aligned}
&\mathbf{lebesgue}\\
=\ &\{\mathbf{fail}\text{ is identity of }\oplus\}\\
&\mathbf{lebesgue} \oplus \mathbf{fail}\\
=\ &\{\text{the set }\{4\}\text{ is }\mathbf{lebesgue}\text{-negligible; and the observation }4 \neq 4\text{ is }\mathbf{false}\}\\
&\mathbf{do}\ \{t \leftsquigarrow \mathbf{lebesgue};\ \mathbf{observe}\ (t \neq 4);\ \mathbf{return}\ t\} \oplus \mathbf{do}\ \{t \leftsquigarrow \mathbf{return}\ 4;\ \mathbf{observe}\ (t \neq 4);\ \mathbf{return}\ t\}\\
=\ &\{\oplus\text{ distributivity (18)}\}\\
&\mathbf{do}\ \{t \leftsquigarrow \mathbf{lebesgue} \oplus \mathbf{return}\ 4;\ \mathbf{observe}\ (t \neq 4);\ \mathbf{return}\ t\}. \qquad (66)
\end{aligned}$$

Bases $\quad b ::= \mathbf{mix}\ l\ \wr e, \ldots \int \mid \mathbf{return}\ () \mid b \otimes\!\!= \lambda x.\, b \mid b \oplus b$

Continuity $\quad l ::= \mathrm{ff} \mid \mathrm{tt}$

Base typing rules

$$\frac{e : \mathbb{R} \quad \cdots}{\mathbf{mix}\ l\ \wr e, \ldots \int : \mathbb{B}\,\mathbb{R}} \qquad \frac{}{\mathbf{return}\ () : \mathbb{B}\,\mathbb{1}} \qquad \frac{\begin{array}{c}[x : \alpha] \\ \vdots \\ b_1 : \mathbb{B}\,\alpha \quad b_2 : \mathbb{B}\,\beta\end{array}}{b_1 \otimes\!\!= \lambda x.\, b_2 : \mathbb{B}\,(\alpha \times \beta)} \qquad \frac{b_1 : \mathbb{B}\,\alpha \quad b_2 : \mathbb{B}\,\beta}{b_1 \oplus b_2 : \mathbb{B}\,(\alpha + \beta)}$$

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

$$\lhd\ e \qquad (b_1 \otimes\!\!= \lambda x.\, b_2)\ v\ c\ h = \lhd\ (fst\ e)\ b_1\ (fst\ v)\ \big(\lhd\ (snd\ e)\ (b_2\{x \mapsto fst\ v\})\ (snd\ v)\ c\big)\ h$$

$$\lhd\ u \qquad b \qquad v\ c\ h = \mathbf{do}\ \{() \leftsquigarrow (\mathbf{mix}\ \mathrm{ff}\ \wr u \int \div b)\ v;\ c\ h\} \quad \text{where } u : \mathbb{R} \text{ is atomic}$$

$$\lhd\ r \qquad b \qquad v\ c\ h = \mathbf{do}\ \{() \leftsquigarrow (\mathbf{mix}\ \mathrm{ff}\ \wr r \int \div b)\ v;\ c\ h\} \quad \text{where } r : \mathbb{R} \text{ is a literal}$$

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

$$\lll\ \mathbf{lebesgue}\ b \qquad v\ c\ h = \mathbf{do}\ \{() \leftsquigarrow (\mathbf{mix}\ \mathrm{tt}\ \wr \int \div b)\ v;\ c\ h\}$$

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

$$jacobian\ f\ (\mathbf{mix}\ \mathrm{ff}\ \_) = \lambda v.\, 1 \qquad reparam\ f\ (\mathbf{mix}\ l\ \wr e_1, \ldots \int) = \mathbf{mix}\ l\ \wr inv\ f\ @\ e_1, \ldots \int$$

$$jacobian\ f\ (\mathbf{mix}\ \mathrm{tt}\ \wr e_1, \ldots \int) = \lambda v.\, \mathbf{if}\ v = e_1 \vee \cdots\ \mathbf{then}\ 1\ \mathbf{else}\ |diff\ (inv\ f)\ v|$$

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

$$\mathbf{mix}\ \mathrm{tt}\ \_ \quad \div\ \mathbf{mix}\ \mathrm{ff}\ \_ = \emptyset$$

$$\mathbf{mix}\ \mathrm{tt}\ \wr \int \quad \div\ \mathbf{mix}\ \mathrm{tt}\ \wr e_1, \ldots \int = \lambda v.\, \mathbf{do}\ \{\mathbf{observe}\ (v \neq e_1 \wedge \cdots);\ \mathbf{return}\ ()\}$$

$$\mathbf{mix}\ \mathrm{ff}\ \wr \int \quad \div\ \_ = \mathbf{fail}$$

$$\mathbf{mix}\ \mathrm{ff}\ \wr e \int \quad \div\ \mathbf{mix}\ \_\ \wr e_1, \ldots \int = \lambda v.\, \mathbf{do}\ \{\mathbf{observe}\ (v = e);\ \mathbf{factor}\ 1/\#\wr i \mid v = e_i \int;\ \mathbf{return}\ ()\}$$

if $e$ is known to be equal to at least one of $\wr e_1, \ldots \int$

$$\mathbf{mix}\ \mathrm{ff}\ \wr e \int \quad \div\ \mathbf{mix}\ \_\ \wr e_1, \ldots \int = \emptyset \quad \text{otherwise}$$

$$\mathbf{mix}\ l\ \wr e, e_1, \ldots \int \div b = \lambda v.\, (\mathbf{mix}\ \mathrm{ff}\ \wr e \int \div b)\ v \oplus (\mathbf{mix}\ l\ \wr e_1, \ldots \int \div b)\ v$$

Fig. 22. 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 $b : \mathbb{B}\,\alpha$, which says that $b$ is a base measure over $\alpha$. In the new definition of $\div$, the syntactic sugar **observe** $\cdots$ for **let inl** $\_ = \cdots$ 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.

The **observe** $(t \neq 4)$ emitted in this case is how, in the GPA problem in Figure 2, our disintegrator knows to revise the weight at 4 to 0.

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 a *congruence*, meaning an equivalence relation that is preserved by substitution and @:

(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 $e_3$.
(4) Whenever it declares $e_1$ equal to $e_2$, it must also declare $e_1\{x \mapsto e\}$ equal to $e_2\{x \mapsto e\}$ 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 ÷ 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 2.23). Sections 4 and 5 have implemented this relation as a program transformation that takes the joint and base measures and produces the kernel. Building on that work, in this section we implement the same ternary relation as a different program transformation that takes the joint measure and produces the base measure. In other words, 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 3.2.3 and 3.2.4. That measure has a disintegration not with respect to the Lebesgue base $\mathbf{mix}\ \mathrm{tt}\ \wr\int$ but with respect to the mixture base $\mathbf{mix}\ \mathrm{tt}\ \wr 0\int$. The *check* disintegrator can find that density, as well as another density with respect to $\mathbf{mix}\ \mathrm{tt}\ \wr 0, 1\int$. However, it may fail with respect to $\mathbf{mix}\ \mathrm{tt}\ \wr 0^2\int$, depending on whether the term-equality checker used in ÷ is complete enough to affirm that $0^2 = 0$. In general, *check* finds a kernel with respect to precisely those bases $\mathbf{mix}\ \mathrm{tt}\ \wr e_0, \ldots\int$ 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 ÷ is the only place where base checking can fail—by returning $\emptyset$, the empty set of solutions, in Figure 22. 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 22. We define when $b$ is *divisible* by $b'$ (notated $b <: b'$) inductively:

(1) If $b = \mathbf{mix}\ l\ \wr e_1, \ldots\int$ and $b' = \mathbf{mix}\ l'\ \wr e'_1, \ldots\int$ over $\mathbb{R}$, then $b <: b'$ if and only if $b \div b' \neq \emptyset$. In other words, $b <: b'$ if and only if $l$ implies $l'$ and every element of $\wr e_1, \ldots\int$ is known to equal some element of $\wr e'_1, \ldots\int$.
(2) If $b = b' = \mathbf{return}\ ()$ over $\mathbb{1}$, then $b <: b'$ always.
(3) If $b = b_1 \mathbin{\otimes\!\!=} \lambda x.\, b_2$ and $b' = b'_1 \mathbin{\otimes\!\!=} \lambda x.\, b'_2$ 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 $b = b_1 \oplus b_2$ and $b' = b'_1 \oplus b'_2$ 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 ÷ in Figure 22 computes a preorder that is preserved by substitution and @. □

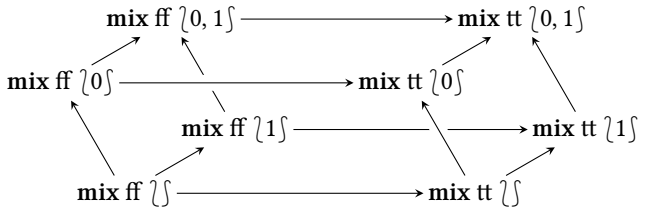


Fig. 23. The preorder $<:$ ("is divisible by"), where an edge from $b$ to $b'$ means $b <: b'$

We illustrate a small slice of this preorder as a graph in Figure 23, 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 $\mathbf{mix}\ \mathrm{tt}\ ⟅⟆ <: \mathbf{mix}\ \mathrm{tt}\ ⟅0⟆ <: \mathbf{mix}\ \mathrm{tt}\ ⟅0, 1⟆$.

Given a joint measure $m$, the base measures $b$ for which *check* $m$ $b$ succeeds (if any) are typically not unique. One reason is the nondeterminism $\cup$ in Figure 14. But even when we restrict that nondeterminism to a certain execution path (as if $\cup$ makes a deterministic choice dictated by an external oracle), there are still infinitely many bases to choose from:

Proposition 6.4. *If (a certain execution path of) one of the functions check, $\lhd$, $\lessdot$, $\lll$ 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. □

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* (for a certain execution path of the *check* disintegrator) if for every base measure $b'$, (that execution path of) *check* can find a kernel with respect to $b'$ (that is, *check* $m$ $b'$ $t \neq \emptyset$, where $t$ is a fresh variable) if and only if $b <: b'$. (In particular, *check* succeeds on $b$ itself: *check* $m$ $b$ $t \neq \emptyset$ 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 (27). 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 disintegration and density 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 22. That *unrestricted* disintegrator is described in Section 7 below.

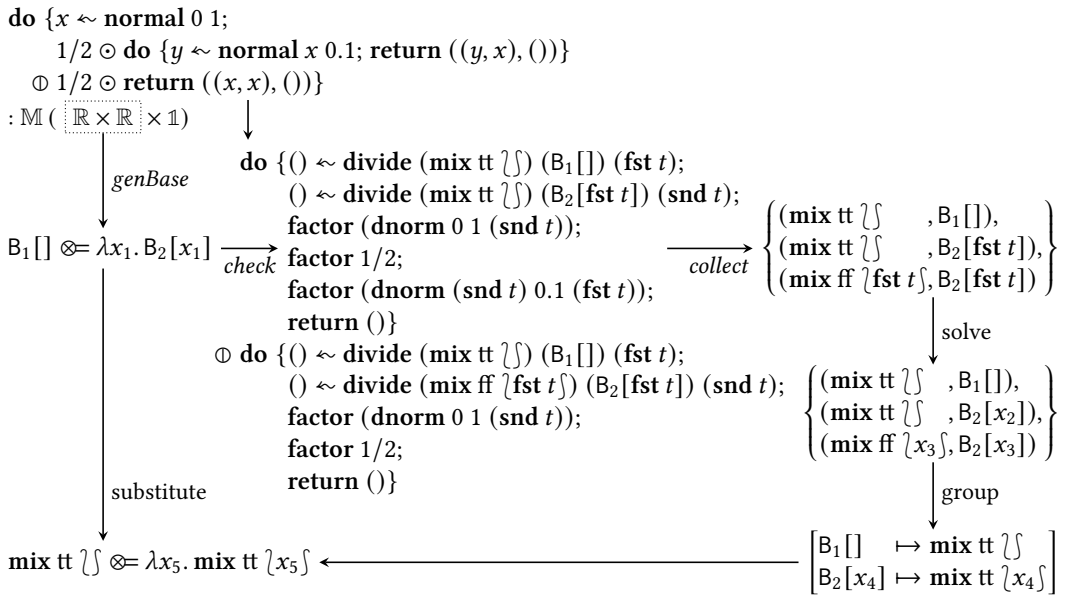


Fig. 24. Inferring a principal base measure (lower left) for an input program (upper left)

### 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 first sketch these steps using an example, then detail each step in the subsections below.

Figure 24 shows our example input program and how it undergoes each step of base inference. This example is motivated by Metropolis-Hastings sampling (Section 2.3): the input program, in the upper-left corner of Figure 24, can alternatively be written as $(\kappa \mathrel{=\!\!\otimes} \xi) \otimes \mathbf{return}\ ()$, composed of the target distribution

$$\xi \;=\; \mathbf{normal}\ 0\ 1 \qquad : \quad \mathbb{M}\,\mathbb{R} \tag{67}$$

and the proposal kernel

$$\kappa \;=\; \lambda x.\, \big((1/2) \odot \mathbf{normal}\ x\ 0.1\big) \oplus \big((1/2) \odot \mathbf{return}\ x\big) \quad : \quad \mathbb{R} \to \mathbb{M}\,\mathbb{R} \tag{68}$$

(similar to the composite kernel in (26) 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 22 mandates that all base measures over $\mathbb{R} \times \mathbb{R}$ must have the form $\mathbf{mix} \ldots \mathrel{\otimes\!\!=} \lambda x.\, \mathbf{mix} \ldots$, in which the term variable $x : \mathbb{R}$ can appear in the second $\mathbf{mix} \ldots$. We represent this form by $\mathsf{B}_1[] \mathrel{\otimes\!\!=} \lambda x_1.\, \mathsf{B}_2[x_1]$, returned by *genBase* in Figure 22. Here the base variables $\mathsf{B}_1$ and $\mathsf{B}_2$ each range over a base measure $\mathbf{mix} \ldots$ over $\mathbb{R}$. The notation $\mathsf{B}_2[x_1]$ means that $\mathsf{B}_2$ can contain zero or more occurrences of a *hole*, plugged by a fresh term variable $x_1 : \mathbb{R}$.

Our second step of base inference is to invoke the *check* disintegrator with respect to the unknown base. Whenever *check* matches a base variable against a pattern $\mathbf{mix} \ldots : \mathbb{B}\,\mathbb{R}$, it suspends that part of the program transformation and produces a *residual* term [Jones et al. 1993]. This suspension

only happens in the functions *jacobian*, *reparam*, and ÷. For example, the call **mix** tt ⟅∫ ÷ $\mathsf{B}_1[]$ produces the term **divide** (**mix** tt ⟅∫) ($\mathsf{B}_1[]$) in Figure 24, which represents an unknown density of the Lebesgue measure with respect to the unknown base $\mathsf{B}_1[]$. Each occurrence of **divide** in the output of *check* expresses a constraint on a base variable, such as **mix** tt ⟅∫ <: $\mathsf{B}_1[]$. Collecting these constraints yields a set of pairs such as (**mix** tt ⟅∫, $\mathsf{B}_1[]$). The *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 24, the *solve* step turns each constraint on a plugged base variable (such as the constraint (**mix** ff ⟅**fst** $t$∫, $\mathsf{B}_2$[**fst** $t$]) on $\mathsf{B}_2$[**fst** $t$]) into a constraint on the base variable unplugged (such as the constraint (**mix** ff ⟅$x_3$∫, $\mathsf{B}_2[x_3]$) on $\mathsf{B}_2$, in which the fresh term variable $x_3 : \mathbb{R}$ is local). Then, the *group* step conjoins all the constraints on each base variable (such as the constraints (**mix** tt ⟅∫, $\mathsf{B}_2[x_2]$) and (**mix** ff ⟅$x_3$∫, $\mathsf{B}_2[x_3]$) on $\mathsf{B}_2$) into one constraint (such as the constraint (**mix** tt ⟅$x_4$∫, $\mathsf{B}_2[x_4]$) on $\mathsf{B}_2$, expressed as a substitution using ↦).

Substituting these conjoined constraints for the base variables in the unknown base $\mathsf{B}_1[]$ ⊗= $\lambda x_1.\, \mathsf{B}_2[x_1]$ yields the principal base **mix** tt ⟅∫ ⊗= $\lambda x_5.$ **mix** tt ⟅$x_5$∫. Principality means that the *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 $\mathsf{B}[]$ to our language of bases in Figure 22. The $\mathsf{B}$ part of the notation expresses that a base variable names a stand-in that must be solved to produce a base measure, while the [] part expresses that the solution is a base measure *with holes*. A base with holes is like a function from terms to bases but cannot, say, distinguish the form of a term that plugs a hole.

Base variables represent bases with holes because they might occur under a binder $\lambda x$ to the right of ⊗=. 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 $\mathsf{B}[]$ with a sequence of core Hakaru variables $\vec{x}$. Moreover, the pair case ◁ $e$ ($b_1$ ⊗= $\lambda x.\, b_2$) $v$ in Figure 22 requires a sequence of terms $\vec{e}$, not just variables $\vec{x}$. In this case, ◁ deconstructs the ⊗= base measure and substitutes *fst* $v$ for the bound variable $x$ in $b_2$. Hence, a *plugged* base variable must store terms that may be in scope after such a substitution.

We thus add a new construct, $\mathsf{B}[\vec{e}]$, which represents a base measure with holes that are plugged by terms $\vec{e}$ in scope. This construct stores a name $\mathsf{B}$ 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 25 shows the new base language, which combines unknown bases with *ground bases*.

*Definition 6.6.* A *ground base* $\underline{b}$ is a base measure with no base variables. The (meta) type of a ground base is notated $\underline{\mathbb{B}}\,\alpha$. 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 24, the ground base in the lower-left corner is the result of applying the ground substitution in the lower-right corner to the base $\mathsf{B}_1[]$ ⊗= $\lambda x_1.\, \mathsf{B}_2[x_1]$ produced by *genBase*.

The relation <: (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 22, base measures have a unique structure for each of the other non-measure type constructors: **return** () is the only base over $\mathbb{1}$, bases over pairs are always of the form $b_1$ ⊗= $\lambda x.\, b_2$, and bases over disjoint sums are always of the form $b_1 \oplus b_2$.

$$
\begin{array}{ll}
\text{Bases} & b ::= \mathbf{mix}\ l\ \wr e, \ldots \int \mid \mathbf{return}\ () \mid b \otimes\!= \lambda x.\, b \mid b \oplus b \mid B \\
\text{Unknown bases} & B ::= \mathsf{B}[\vec{e}] \mid \mathbf{reparam}\ f\ B \\
\text{Base variables} & \mathsf{B}[] \\
\text{Terms} & e ::= \cdots \mid \mathbf{jacobian}\ f\ B\ e \mid \mathbf{divide}\ \underline{b}\ \mathsf{B}[\vec{e}]\ e \\
\text{New base typing rule} & \dfrac{}{B : \mathbb{B}\,\mathbb{R}}
\end{array}
$$

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

$$
\begin{array}{ll}
jacobian : \mathsf{invertible} \to \mathbb{B}\,\mathbb{R} \to \lfloor \mathbb{R} \rfloor \to \lfloor \mathbb{R} \rfloor & reparam : \mathsf{invertible} \to \mathbb{B}\,\mathbb{R} \to \mathbb{B}\,\mathbb{R} \\
jacobian\ f\ B\ e = \mathbf{jacobian}\ f\ B\ e & reparam\ f\ B = \mathbf{reparam}\ f\ B
\end{array}
$$

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

Fig. 25. Introducing base-measure variables and updating the base-checking disintegrator. The first line adds unknown bases $B$ to the grammar of bases $b$. The fifth line adds a typing rule to say that every unknown base $B$ is over $\mathbb{R}$.

### 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 $(\underline{b}, \mathsf{B}[\vec{e}])$. Given an input program, we obtain a set of constraints by

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

First, we need to construct a base $b^*$ that may contain base variables. As done above in (42), we construct the base using $genBase(\alpha)$. We modify the definition of $genBase(\alpha)$—as shown in the middle of Figure 25—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 24, we have $\alpha = \mathbb{R} \times \mathbb{R}$ and we construct $b^* = \mathsf{B}_1[] \otimes\!= \lambda x_1.\, \mathsf{B}_2[x_1]$. The constructed $b^*$ represents an unknown base measure because any ground base $\underline{b}$ of type $\underline{\mathbb{B}}\,\alpha$ can be obtained by applying to $b^*$ a suitable ground substitution $\sigma$ over the base variables. The function $genBase(\alpha)$ now takes a set of core Hakaru variables $\vec{x}$ 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 $b^* = genBase(\alpha)\ ()$.

Second, we need to run base-checking disintegration with respect to $b^*$. In order for this to work we need to update *jacobian*, *reparam*, and $\div$ to handle unknown bases. The necessary updates are shown at the bottom of Figure 25.

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

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

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

Thus, the new constructs **jacobian**, **reparam**, and **divide** represent suspended calls to the existing functions *jacobian*, *reparam*, and $\div$. These suspended calls are resumed when a ground substitution (Definition 6.6) is applied.

*Definition 6.7.* We define applying a ground substitution $\sigma$ to an unknown base:

$$\sigma\ (\mathsf{B}[\vec{e}]) \qquad = \underline{b}\{\vec{x} \mapsto \vec{e}\} \qquad \text{where } \sigma \text{ maps } \mathsf{B}[\vec{x}] \text{ to } \underline{b}, \tag{69}$$

$$\sigma\ (\mathbf{reparam}\ f\ B) = \mathit{reparam}\ (\sigma\ f)\ (\sigma\ B). \tag{70}$$

We define applying a ground substitution $\sigma$ to a term $e$:

$$\sigma\ (\mathbf{jacobian}\ f\ B\ e)\ = \mathit{jacobian}\ (\sigma\ f)\ (\sigma\ B)\ (\sigma\ e), \tag{71}$$

$$\sigma\ (\mathbf{divide}\ \underline{b}\ \mathsf{B}[\vec{e}]\ e) = \big(\underline{b} \div \sigma\ (\mathsf{B}[\vec{e}])\big)(\sigma\ e). \tag{72}$$

(We omit many cases of $\sigma\ e$ that just recursively apply $\sigma$ to the immediate subterms of $e$.) We define applying a ground substitution $\sigma$ to an invertible $f$; all the cases just recursively apply $\sigma$ to the immediate subterms of $f$: for example, $\sigma\ (\mathbf{mul}\ v) = \mathbf{mul}\ (\sigma\ v)$.

With the updates in Figure 25, running base-checking disintegration with respect to a non-ground base produces a program with embedded residual $\div$ calls of the form **divide** $\underline{b}\ \mathsf{B}[\vec{e}]\ e$, where $\underline{b}$ is a ground base, $\mathsf{B}[\vec{e}]$ 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 **divide** expression into a set of constraints $(\underline{b}, \mathsf{B}[\vec{e}])$. This simple *collect* function is defined in Figure 26. The composition of *check* followed by *collect* is illustrated in Figure 24.

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

$$\boxed{\mathit{check}\ (\sigma\ m)\ (\sigma\ b^*)\ t\ \ =\ \ \sigma(\mathit{check}\ m\ b^*\ t)}. \tag{73}$$

That is, applying a ground substitution $\sigma$ to produce a ground base $\sigma\ b^*$ then running *check* is equivalent to running *check* then applying $\sigma$ to resume the disintegration. For a simple example, let

$$m = \mathbf{normal}\ 0\ 1 \otimes \mathbf{return}\ () : \mathbb{M}\,(\mathbb{R} \times \mathbb{1}), \quad b^* = \mathsf{B}_1[] : \mathbb{B}\,\mathbb{R}, \quad \sigma = \big\{\mathsf{B}_1[] \mapsto \mathbf{mix}\ \mathsf{tt}\ \langle 4 \rangle\big\}. \tag{74}$$

$$
\begin{array}{lll}
\mathit{collect}\ x = & \mathit{collect}\ (\mathbf{fst}\ e) = & \mathit{collect}\ (e_1, e_2) = \\
\mathit{collect}\ r = & \mathit{collect}\ (\mathbf{snd}\ e) = & \mathit{collect}\ (e_1 \oplus e_2) = \\
\mathit{collect}\ () = & \mathit{collect}\ (\mathbf{inl}\ e) = & \mathit{collect}\ (\mathbf{do}\ \{x \leftsquigarrow e_1;\ e_2\}) = \\
\mathit{collect}\ \mathbf{lebesgue} = & \mathit{collect}\ (\mathbf{inr}\ e) = & \mathit{collect}\ (\mathbf{do}\ \{\mathbf{factor}\ e_1;\ e_2\}) = \\
\mathit{collect}\ \mathbf{fail} = \{\} & \mathit{collect}\ (\mathbf{return}\ e) = & \mathit{collect}\ (\mathbf{do}\ \{\mathbf{let\ inl}\ x = e_1;\ e_2\}) = \\
 & \mathit{collect}\ (\mathbf{sqrt}\ e) = & \mathit{collect}\ (\mathbf{do}\ \{\mathbf{let\ inr}\ x = e_1;\ e_2\}) = \\
 & \mathit{collect}\ e^2 = \mathit{collect}\ e & \mathit{collect}\ (e_1 \cdot e_2) = \\
 & & \mathit{collect}\ (e_1 < e_2) = \mathit{collect}\ e_1 \cup \mathit{collect}\ e_2
\end{array}
$$

$$\mathit{collect}\ (\mathbf{divide}\ \underline{b}\ \mathsf{B}[\vec{e}]\ e) = \{(\underline{b}, \mathsf{B}[\vec{e}])\}$$

Fig. 26. Collecting **divide** expressions into a set of constraints

On the right-hand side of (73), running *check* with the non-ground base $b^*$ produces the program

$$
\begin{aligned}
\mathit{check}\ m\ b^*\ t = \mathbf{do}\ \{() \leftsquigarrow\ & \mathbf{divide}\ (\mathbf{mix}\ \mathsf{tt}\ \wr\int)\ (\mathsf{B}_1[])\ t; \\
& \mathbf{factor}\ (\mathbf{dnorm}\ 0\ 1)(t); \\
& \mathbf{return}\ ()\},
\end{aligned}
\tag{75}
$$

which contains a residual **divide**. On the left-hand side of (73), we have $\sigma\ m = m$, and running *check* with the ground base $\sigma\ b^* = \mathbf{mix}\ \mathsf{tt}\ \wr 4\int$ produces the program

$$
\begin{aligned}
\mathit{check}\ m\ (\sigma\ b^*)\ t = \mathbf{do}\ \{() \leftsquigarrow\ & \mathbf{do}\ \{\mathbf{observe}\ (t \neq 4);\ \mathbf{return}\ ()\}; \\
& \mathbf{factor}\ (\mathbf{dnorm}\ 0\ 1)(t); \\
& \mathbf{return}\ ()\},
\end{aligned}
\tag{76}
$$

which does not contain any residual **divide**. Equation (73) relates these two programs because applying the ground substitution $\sigma$ to (a program containing) **divide** resumes a suspended call to $\div$:

$$
\begin{aligned}
& \sigma\big(\mathbf{divide}\ (\mathbf{mix}\ \mathsf{tt}\ \wr\int)\ (\mathsf{B}_1[])\ t\big) \\
= \quad & \{\text{applying } \sigma \text{ to a term (72)}\} \\
& \big(\mathbf{mix}\ \mathsf{tt}\ \wr\int \div \mathbf{mix}\ \mathsf{tt}\ \wr 4\int\big)(t) \\
= \quad & \{\text{definition of } \div \text{ (Figure 22)}\} \\
& \mathbf{do}\ \{\mathbf{observe}\ (t \neq 4);\ \mathbf{return}\ ()\}.
\end{aligned}
\tag{77}
$$

### 6.5 Solving constraints to produce the principal base measure

The final step of base inference produces a ground base $\underline{b}^*$ that is principal. Like all ground bases of the correct type, $\underline{b}^*$ shares the structure of the base measure $b^*$ constructed initially by *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 $(\underline{b}, \mathsf{B}[\vec{e}])$. This set may contain multiple constraints for the same base variable $\mathsf{B}[]$ (plugged with possibly different term sequences $\vec{e}$). From such a set we obtain a principal base in three steps.

(1) *Solve* each individual constraint in the set. This is the only step that can fail. 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 $(\underline{b}_1, \mathsf{B}[\vec{e}])$ is a ground base replacement $\underline{b}_2[]$ for the base variable $\mathsf{B}[]$ such that $\underline{b}_1 <: \underline{b}_2[\vec{e}]$. Because the relation $<:$ is transitive, there may be infinitely many solutions. We need a *principal* solution $\underline{b}_1^*[]$ such that

$$\forall \underline{b}_2[], \quad \underline{b}_1 <: \underline{b}_2[\vec{e}] \quad \Leftrightarrow \quad \underline{b}_1^*[\vec{x}] <: \underline{b}_2[\vec{x}], \tag{78}$$

so in particular $\underline{b}_1 <: \underline{b}_1^*[\vec{e}]$. Here $\vec{x}$ are as many fresh variables as there are terms in $\vec{e}$. If $\underline{b}_1$ contained no open core Hakaru terms, then we could just let $\underline{b}_1^*[] = \underline{b}_1$ and not use any hole. For example in Figure 24, the first two collected constraints ($\mathbf{mix}\ \mathrm{tt}\ \wr\int, \mathsf{B}_1[]$) and ($\mathbf{mix}\ \mathrm{tt}\ \wr\int, \mathsf{B}_2[\mathbf{fst}\ t]$) both have the hole-less principal solution $\mathbf{mix}\ \mathrm{tt}\ \wr\int$. Using such a hole-less solution is *almost* what we do.

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

For example in Figure 24, to solve the third collected constraint ($\mathbf{mix}\ \mathrm{ff}\ \wr\mathbf{fst}\ t\int, \mathsf{B}_2[\mathbf{fst}\ t]$), we solve the matching problem $?[\mathbf{fst}\ t] = \mathbf{mix}\ \mathrm{ff}\ \wr\mathbf{fst}\ t\int$, by replacing $\mathbf{fst}\ t$ in $\mathbf{mix}\ \mathrm{ff}\ \wr\mathbf{fst}\ t\int$ with a hole. The result of this replacement is $\mathbf{mix}\ \mathrm{ff}\ \wr[]\int = (\mathbf{mix}\ \mathrm{ff}\ \wr\mathbf{fst}\ t\int)\{[] \leftarrowtail \mathbf{fst}\ t\}$, which does not contain any free variable, so it is the principal solution for $\mathsf{B}_2[]$. This principal solution is represented in Figure 24 as ($\mathbf{mix}\ \mathrm{ff}\ \wr x_3\int, \mathsf{B}_2[x_3]$), 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 $\mathbf{fst}\ t$ and $\mathbf{fst}\ (\mathbf{snd}\ t)$, produced by the first case of $\triangleleft$ in Figure 22—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 no longer need to worry about failure once we have successfully solved each constraint. The next step is to group the solved constraints $(\underline{b}_1^*[\vec{x}], \mathsf{B}[\vec{x}])$ by their base variables $\mathsf{B}[]$. For this we define a binary operation *bplus* that acts as a join in the preorder $<:$ by summing together ground base measures of type $\mathbb{B}\,\mathbb{R}$. Since $\mathbf{mix}$ is the only way to construct such bases, *bplus* has a one-line definition:

$$\begin{aligned} &\mathit{bplus} : \underline{\mathbb{B}}\,\mathbb{R} \to \underline{\mathbb{B}}\,\mathbb{R} \to \underline{\mathbb{B}}\,\mathbb{R} \\ &\mathit{bplus}\ (\mathbf{mix}\ \ l\ \wr e_1, \ldots\int)\ (\mathbf{mix}\ \ l'\ \wr e_1', \ldots\int) = \mathbf{mix}\ \ (l \vee l')\ \wr e_1, \ldots, e_1', \ldots\int. \end{aligned} \tag{79}$$

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

For example in Figure 24, the two solved constraints ($\mathbf{mix}\ \mathrm{tt}\ \wr\int, \mathsf{B}_2[x_2]$) and ($\mathbf{mix}\ \mathrm{ff}\ \wr x_3\int, \mathsf{B}_2[x_3]$) get grouped into $\mathsf{B}_2[x_4] \mapsto \mathbf{mix}\ \mathrm{tt}\ \wr x_4\int$, because *bplus* joins $\mathbf{mix}\ \mathrm{tt}\ \wr\int$ and $\mathbf{mix}\ \mathrm{ff}\ \wr x_4\int$ into $\mathbf{mix}\ \mathrm{tt}\ \wr x_4\int$. 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, for each base variable $\mathsf{B}$ we have a solution $\mathsf{B}[\vec{x}] \mapsto \underline{b}$. These solutions together constitute a ground substitution $\sigma^*$, such as the lower-right corner in Figure 24. We obtain our principal base measure $\underline{b}^*$ by substituting these ground bases into $b^*$:

$$\underline{b}^* = \sigma^*\ b^*. \tag{80}$$

In Figure 24 we produce the base $\underline{b}^* = \mathbf{mix}\ \mathrm{tt}\ \wr\int \otimes\!\!= \lambda x_5.\ \mathbf{mix}\ \mathrm{tt}\ \wr x_5\int$.

## 7 AN UNRESTRICTED BASE-CHECKING DISINTEGRATOR

When the user of disintegration has a base measure in mind, it is easier to specify it as a core Hakaru measure term rather than in the relatively spartan base language of Figure 22. 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 $\zeta \mathrel{=\!\!\otimes} \xi$ with respect to $\xi \mathrel{\otimes\!\!=} \zeta$ [Tierney 1998], where $\xi$ and $\zeta$ are specified as probabilistic programs.

To disintegrate one core Hakaru term with respect to another, we use Proposition 2.10 and define

$$\mathit{disint} : \lceil \mathbb{M}\,(\alpha \times \beta)\rceil \to \lceil \mathbb{M}\,\alpha\rceil \to \lfloor \alpha \rfloor \to \{\lfloor \mathbb{M}\,\beta \rfloor\} \qquad \boxed{m_1 \;\sqsupseteq\; m_2 \mathrel{\otimes\!\!=} \mathit{disint}\; m_1\; m_2} \quad (81)$$

$$\mathit{disint}\; m_1\; m_2\; t = |\mathit{check}\; m_2'\; b\; t|^{\mathbf{-1}} \odot \mathit{check}\; m_1\; b\; t \quad \text{where} \quad b = \mathit{infer}\; m_2', \quad m_2' = m_2 \otimes \mathbf{return}\; ().$$

That is, we use base inference to find the intermediate base measure $\mu$ in Proposition 2.10(1). One caveat of this approach is that *disint* takes the reciprocal of the density $|\mathit{check}\; m_2'\; b\; t|$ and thus assumes that the density is almost never 0 or $\infty$. We discuss this caveat at the end of Section 8.

## 8 EVALUATION

Because we handle distributions whose disintegration had never been automated before, there is not yet a corpus of programs found 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:

(E1) a clamped normal distribution with respect to a **mix** tt base (Example 2.6),
(E2) clamped normal distributions with respect to each other (Example 2.11),
(E3) mutual information (Example 2.12) in a joint distribution that is a discrete-continuous mixture,
(E4) importance sampling using a custom, non-continuous proposal distribution (Example 2.14),
(E5) Metropolis-Hastings sampling (Example 2.17) using single-site and reversible-jump proposals,
(E6) belief update using a clamped observation (Example 2.21),
(E7) Gibbs sampling (Example 2.22) of a joint distribution whose marginals are non-continuous.

Each of these calls to our disintegrator implementation takes a fraction of a second to complete using an ordinary personal computer.

### 8.1 Clamped distributions

Evaluations (E1) and (E2), clamped densities, demonstrate how our disintegrator handles discrete-continuous mixtures (B1) and supports exact and approximate inference. The two distributions concerned are **normal** 0 1 and **normal** 3 2 (or any other continuous distributions whose densities can be found by previous disintegrators), both clamped to between 0 and 1:

$$\xi \;=\; \mathbf{do}\; \{x \leftsquigarrow \mathbf{normal}\; 0\; 1;\; \mathbf{return}\; \max\{0, \min\{1, x\}\}\}, \quad (82)$$

$$\mu \;=\; \mathbf{do}\; \{x \leftsquigarrow \mathbf{normal}\; 3\; 2;\; \mathbf{return}\; \max\{0, \min\{1, x\}\}\}. \quad (83)$$

In these terms, max and min simply abbreviate **if**:

$$\max\{0, \min\{1, x\}\} \;=\; \mathbf{if}\; 0 < \min\{1, x\}\; \mathbf{then}\; \min\{1, x\}\; \mathbf{else}\; 0, \quad (84)$$

$$\min\{1, x\} \;=\; \mathbf{if}\; 1 < x\; \mathbf{then}\; 1\; \mathbf{else}\; x. \quad (85)$$

In turn, our disintegrator implementation handles **if** by expanding it on the fly to surrounding guards and $\oplus$, as explained by equation (37).

For evaluation (E1), to find a density of $\xi$ (at $t : \mathbb{R}$), we can invoke the restricted base-checking disintegrator (Section 5) by $\mathit{check}\; (\xi \otimes \mathbf{return}\; ())\; b\; t$. If we specify the base measure $b = \mathbf{mix}\; \mathrm{tt}\; \wr\int$ (meaning the Lebesgue measure) or $b = \mathbf{mix}\; \mathrm{ff}\; \wr 0, 1\int$ (meaning the discrete mixture $\mathbf{return}\; 0 \oplus \mathbf{return}\; 1$), then the disintegrator correctly returns no solution. If we let $b = \mathbf{mix}\; \mathrm{tt}\; \wr 0, 1\int$ (meaning

the discrete-continuous mixture **lebesgue** ⊕ **return** 0 ⊕ **return** 1), then the disintegrator correctly returns the following solution (simplified by removing **factor** 1 and inlining **let**-bindings):

$$\begin{aligned}
&\textbf{do}\ \{\textbf{observe}\ t = 1;\ x \leftsquigarrow \textbf{normal}\ 0\ 1;\ \textbf{observe}\ 0 < \min\{1, x\};\ \textbf{observe}\ 1 < x;\ \textbf{return}\ ()\}\\
\oplus\ &\textbf{do}\ \{\textbf{observe}\ t \neq 0 \wedge t \neq 1; \textbf{factor}\ (\textbf{dnorm}\ 0\ 1)(t); \textbf{observe}\ 0 < \min\{1, t\}; \textbf{observe}\ t \leq 1; \textbf{return}\ ()\}\\
\oplus\ &\textbf{do}\ \{\textbf{observe}\ t = 0;\ x \leftsquigarrow \textbf{normal}\ 0\ 1;\ \textbf{observe}\ \min\{1, x\} \leq 0;\ \textbf{return}\ ()\}.
\end{aligned} \tag{86}$$

Applying the totaling transformation then yields the density $\kappa(t)$ in Example 2.6. In particular, the two bindings $x \leftsquigarrow \textbf{normal}\ 0\ 1$ in (86) become the two definite integrals in (15). Thus, to compute further with $\kappa(1)$ and $\kappa(0)$ (whether symbolically or numerically, exactly or approximately) is to compute with those definite integrals, which amounts to computing with the error function. For instance, following (86), we can approximate $\kappa(1)$ and $\kappa(0)$ by rejection-sampling $x$ from $\textbf{normal}\,0\,1$. Alternatively, automatic simplification [Carette and Shan 2016; Gehr et al. 2016] can turn (86) into an exact closed-form expression for the density $\kappa(t)$.

For evaluation (E2), to find a density of $\xi$ with respect to $\mu$, we can invoke the unrestricted disintegrator (Section 7) by $\mathit{disint}\ (\xi \otimes \textbf{return}\ ())\ \mu\ t$. This call succeeds because the base-inferring disintegrator (Section 6) infers the principal base $b = \textbf{mix}\ \mathrm{tt}\ \lbag 0, 1 \rbag$ for $\mu \otimes \textbf{return}\ ()$ (as well as for $\xi \otimes \textbf{return}\ ()$). Applying the totaling transformation to the result of *disint* then produces the ratio expression for a density of $\xi$ with respect to $\mu$, as desired in Example 2.11. This ratio contains four definite integrals (two in the numerator and two in the denominator).

### 8.2 Mutual information

Given a joint probability distribution $\xi : \mathbb{M}\,(\alpha \times \beta)$, let $\kappa : (\alpha \times \beta) \to \mathbb{R}_+$ be a density of $\xi$ with respect to the product of marginals $\mu = (\textbf{fst} \diamond \xi) \otimes (\textbf{snd} \diamond \xi) : \mathbb{M}\,(\alpha \times \beta)$. Then, *mutual information* is the expectation (that is, integral) of $\lambda t.\, \log\big(\kappa(t)\big)$ with respect to $\xi$ [Cover and Thomas 2006]. One way to estimate mutual information is to draw many samples $t$ from $\xi$ and average their values of $\log\big(\kappa(t)\big)$ [Gao et al. 2017]. Evaluation (E3) shows how our disintegrator handles (independent) products of discrete-continuous mixtures (B1) so as to find and evaluate $\kappa$. We let $\xi : \mathbb{M}\,\mathbb{R}^2$ be a discrete-continuous mixture as in Gao et al.'s Experiment I:

$$\begin{aligned}
\xi \;=\; &\big((1/2) \odot (\textbf{normal}\ 0\ 1 \otimes \textbf{normal}\ 0\ 1)\big)\\
&\oplus \big((1/40) \odot \textbf{return}\ (-1, +1)\big) \oplus \big((9/40) \odot \textbf{return}\ (+1, +1)\big)\\
&\oplus \big((9/40) \odot \textbf{return}\ (-1, -1)\big) \oplus \big((1/40) \odot \textbf{return}\ (+1, -1)\big).
\end{aligned} \tag{87}$$

To find a density of $\xi$ with respect to $\mu = (\textbf{fst} \diamond \xi) \otimes (\textbf{snd} \diamond \xi)$, we can invoke the unrestricted disintegrator (Section 7) by $\mathit{disint}\ (\xi \otimes \textbf{return}\ ())\ \mu\ t$. This call succeeds because the base-inferring disintegrator (Section 6) infers the principal base $b = (\textbf{mix}\ \mathrm{tt}\ \lbag -1, +1 \rbag) \otimes (\textbf{mix}\ \mathrm{tt}\ \lbag -1, +1 \rbag)$ for $\mu \otimes \textbf{return}\ ()$ (as well as for $\xi \otimes \textbf{return}\ ()$). As with evaluation (E2) above, applying the totaling transformation to the result of *disint* then produces a ratio expression

$$\lambda(x,y).\ \frac{\begin{array}{l}\big(\textbf{if}\ x = -1 \vee x = +1 \vee y = -1 \vee y = +1\ \textbf{then}\ 0\ \textbf{else}\ (\textbf{dnorm}\ 0\ 1)(x) \cdot (\textbf{dnorm}\ 0\ 1)(y)\big)\\ +\ \big(\textbf{if}\ x = -1 \wedge y = +1\ \textbf{then}\ 1/40\ \textbf{else}\ 0\big) + \big(\textbf{if}\ x = +1 \wedge y = +1\ \textbf{then}\ 9/40\ \textbf{else}\ 0\big)\\ +\ \big(\textbf{if}\ x = -1 \wedge y = -1\ \textbf{then}\ 9/40\ \textbf{else}\ 0\big) + \big(\textbf{if}\ x = +1 \wedge y = -1\ \textbf{then}\ 1/40\ \textbf{else}\ 0\big)\end{array}}{\begin{array}{l}\left(\begin{array}{l}\big(\textbf{if}\ x = -1 \vee x = +1\ \textbf{then}\ 0\ \textbf{else}\ \int_{\mathbb{R}} (\textbf{dnorm}\ 0\ 1)(x) \cdot (\textbf{dnorm}\ 0\ 1)(y)\, dy\big)\\ +\ \big(\textbf{if}\ x = -1\ \textbf{then}\ 1/40\ \textbf{else}\ 0\big) + \big(\textbf{if}\ x = +1\ \textbf{then}\ 9/40\ \textbf{else}\ 0\big)\\ +\ \big(\textbf{if}\ x = -1\ \textbf{then}\ 9/40\ \textbf{else}\ 0\big) + \big(\textbf{if}\ x = +1\ \textbf{then}\ 1/40\ \textbf{else}\ 0\big)\end{array}\right)\\ \cdot \left(\begin{array}{l}\big(\textbf{if}\ y = -1 \vee y = +1\ \textbf{then}\ 0\ \textbf{else}\ \int_{\mathbb{R}} (\textbf{dnorm}\ 0\ 1)(x) \cdot (\textbf{dnorm}\ 0\ 1)(y)\, dx\big)\\ +\ \big(\textbf{if}\ y = +1\ \textbf{then}\ 1/40\ \textbf{else}\ 0\big) + \big(\textbf{if}\ y = +1\ \textbf{then}\ 9/40\ \textbf{else}\ 0\big)\\ +\ \big(\textbf{if}\ y = -1\ \textbf{then}\ 9/40\ \textbf{else}\ 0\big) + \big(\textbf{if}\ y = -1\ \textbf{then}\ 1/40\ \textbf{else}\ 0\big)\end{array}\right)\end{array}} \tag{88}$$

for $\kappa$, a density of $\xi$ with respect to $\mu$. Performing the definite integrals in the denominator is equivalent to simplifying the marginals $\mathbf{fst} \diamond \xi$ and $\mathbf{snd} \diamond \xi$ to eliminate the variable $y$ and $x$, and has been automated [Carette and Shan 2016; Gehr et al. 2016]. Thus, disintegration followed by simplification produces an exact closed-form expression for the density $\kappa$.

### 8.3 Importance sampling

Evaluation (E4) demonstrates how our disintegrator calculates the density needed for importance sampling, even when the proposal distribution is not continuous. Consider the discrete-continuous mixture $\mu$ from (83) as a proposal distribution. This proposal distribution ranges between 0 and 1, so as an example of a target, let us truncate MacKay's (19) to between 0 and 1, and mix in point masses at 0 and 1 for good measure:

$$\begin{aligned}\xi \;=\; & \mathbf{do}\ \{x \leftsquigarrow \mathbf{lebesgue};\ \mathbf{observe}\ 0 < x < 1;\ \exp(0.4(x-0.4)^2 - 0.08x^4) \odot \mathbf{return}\ x\} \\ & \oplus\ (0.37 \odot \mathbf{return}\ 0) \oplus (0.42 \odot \mathbf{return}\ 1).\end{aligned} \tag{89}$$

Importance sampling draws samples from $\mu$ and weights them by a density $\kappa$ of $\xi$ with respect to $\mu$. To find $\kappa$, as in evaluations (E2) and (E3), we can invoke the unrestricted disintegrator (Section 7) by *disint* $(\xi \otimes \mathbf{return}\ ())\ \mu\ t$. This call succeeds because the base-inferring disintegrator (Section 6) infers the principal base $b = \mathbf{mix}\ \mathrm{tt}\ \lfloor 0, 1 \rfloor$ for $\mu \otimes \mathbf{return}\ ()$ (as well as for $\xi \otimes \mathbf{return}\ ()$). Once again, applying the totaling transformation to the result of *disint* then produces the weight function

$$\begin{aligned}&\kappa(0) = 0.37 \Big/ \textstyle\int_{-\infty}^{0} (\mathbf{dnorm}\ 3\ 2)(x)\, dx, \\ &\kappa(x) = \exp(0.4(x-0.4)^2 - 0.08x^4) \Big/ (\mathbf{dnorm}\ 3\ 2)(x) \quad \text{if } 0 < x < 1, \\ &\kappa(1) = 0.42 \Big/ \textstyle\int_{1}^{+\infty} (\mathbf{dnorm}\ 3\ 2)(x)\, dx\end{aligned} \tag{90}$$

for use in the importance sampler.

The definite integrals in (90), like those in (15), can be computed with the error function. But in general, when intractable integrals appear in inference, it may be time to turn the integration variables into auxiliary variables in the distribution and handle them by sampling. After all, Monte Carlo integration is a major application of sampling. Introducing auxiliary variables moves the target distribution to a higher-dimensional product space, where other sampling techniques such as Metropolis-Hastings sampling and Gibbs sampling may be more effective.

### 8.4 Metropolis-Hastings sampling

Evaluation (E5), Metropolis-Hastings sampling, comprises the use of single-site proposals (B2) and reversible-jump proposals (B3). These applications offer a good look at the boundary of what our disintegrator can and cannot handle.

Let $\xi_{\mathbb{R}} : \mathbb{M}\,\mathbb{R}$ and $\xi_{\mathbb{R}^2} : \mathbb{M}\,\mathbb{R}^2$ be some continuous distributions over $\mathbb{R}$ and $\mathbb{R}^2$. (The precise distributions do not matter so long as their densities are expressible; for instance, the densities can be $\lambda x.\, \exp(0.4(x-0.4)^2 - 0.08x^4)$ in (19) and $\kappa$ in (24).) To conduct Metropolis-Hastings sampling of the target $\xi_{\mathbb{R}^2}$, we follow equation (26) and construct the single-site proposal kernel

$$\begin{aligned}\zeta_{\mathbb{R}^2} \;=\; & \lambda x.\, \left(\tfrac{1}{2} \odot \mathbf{do}\ \{x_1' \leftsquigarrow \mathbf{normal}\ (\mathbf{fst}\ x)\ 0.1;\ \mathbf{return}\ (x_1', \mathbf{snd}\ x)\}\right) \\ & \oplus\ \left(\tfrac{1}{2} \odot \mathbf{do}\ \{x_2' \leftsquigarrow \mathbf{normal}\ (\mathbf{snd}\ x)\ 0.1;\ \mathbf{return}\ (\mathbf{fst}\ x, x_2')\}\right).\end{aligned} \tag{91}$$

This proposal kernel flips a fair coin to decide whether to perturb $\mathbf{fst}\ x$ and keep $\mathbf{snd}\ x$ or to perturb $\mathbf{snd}\ x$ and keep $\mathbf{fst}\ x$. And to conduct Metropolis-Hastings sampling of the target $\xi_{\mathbb{R}} \oplus \xi_{\mathbb{R}^2}$ :

$\mathbb{M}\,(\mathbb{R}+\mathbb{R}^2)$, we construct the reversible-jump proposal kernel

$$\begin{aligned}\zeta_{\mathbb{R}+\mathbb{R}^2} \;=\; \lambda z.\,\mathbf{case}\ z\ \mathbf{of}\ \mathbf{inl}\,x_0 &\to \left(\tfrac{1}{2}\odot \mathbf{inl}\diamond \mathbf{normal}\,x_0\ 0.1\right)\\ &\quad \oplus \left(\tfrac{1}{2}\odot \mathbf{inr}\diamond (\mathbf{normal}\,x_0\ 0.1\otimes \mathbf{normal}\,x_0\ 0.1)\right)\\ \mathbf{inr}\,x &\to \left(\tfrac{1}{2}\odot \mathbf{inl}\diamond \mathbf{normal}\,((\mathbf{fst}\,x+\mathbf{snd}\,x)/2)\ 0.1\right)\\ &\quad \oplus \left(\tfrac{1}{2}\odot \mathbf{inr}\diamond (\mathbf{normal}\,(\mathbf{fst}\,x)\ 0.1\otimes \mathbf{normal}\,(\mathbf{snd}\,x)\ 0.1)\right).\end{aligned} \tag{92}$$

This proposal kernel flips a fair coin to decide whether to stay in the same component of the sum type $\mathbb{R}+\mathbb{R}^2$ or to jump to the other component.

For the single-site proposal (91), the Metropolis-Hastings acceptance ratio is the density of $\zeta_{\mathbb{R}^2} \multimap\!\otimes \xi_{\mathbb{R}^2}$ with respect to $\xi_{\mathbb{R}^2} \otimes\!\multimap \zeta_{\mathbb{R}^2}$. To compute the ratio, we invoke the unrestricted base-checking disintegrator (Section 7) by *disint* $\left((\zeta_{\mathbb{R}^2} \multimap\!\otimes \xi_{\mathbb{R}^2}) \otimes \mathbf{return}\ ()\right)\ (\xi_{\mathbb{R}^2} \otimes\!\multimap \zeta_{\mathbb{R}^2})\ t$. This call succeeds because the base-inferring disintegrator (Section 6) infers the principal base

$$(\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \otimes\!\multimap \lambda x_1.\,\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor) \otimes\!\multimap \lambda x.\,(\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \mathbf{fst}\,x \rfloor\!\rfloor \otimes\!\multimap \lambda x_1'.\,\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \mathbf{snd}\,x \rfloor\!\rfloor) : \mathbb{B}\,(\mathbb{R}^2)^2 \tag{93}$$

for $(\xi_{\mathbb{R}^2} \otimes\!\multimap \zeta_{\mathbb{R}^2}) \otimes \mathbf{return}\ ()$ (as well as for $(\zeta_{\mathbb{R}^2} \multimap\!\otimes \xi_{\mathbb{R}^2}) \otimes \mathbf{return}\ ()$). The meaning of this dependent-product principal base is as expected in (27). Applying the totaling transformation to the result of *disint* then produces the desired acceptance ratio.

Similarly, for the reversible-jump proposal (92), the Metropolis-Hastings acceptance ratio is the density of $\zeta_{\mathbb{R}+\mathbb{R}^2} \multimap\!\otimes \xi_{\mathbb{R}+\mathbb{R}^2}$ with respect to $\xi_{\mathbb{R}+\mathbb{R}^2} \otimes\!\multimap \zeta_{\mathbb{R}+\mathbb{R}^2}$. This time, the call *disint* $\left((\zeta_{\mathbb{R}+\mathbb{R}^2} \multimap\!\otimes \xi_{\mathbb{R}+\mathbb{R}^2}) \otimes \mathbf{return}\ ()\right)\ (\xi_{\mathbb{R}+\mathbb{R}^2} \otimes\!\multimap \zeta_{\mathbb{R}+\mathbb{R}^2})\ t$ succeeds by inferring the principal base

$$\begin{aligned}&\left(\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \oplus (\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \otimes\!\multimap \lambda x_1.\,\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor)\right)\\ &\qquad \otimes\!\multimap \lambda z.\,\left(\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \oplus (\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \otimes\!\multimap \lambda x_1'.\,\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor)\right) : \mathbb{B}\,(\mathbb{R}+\mathbb{R}^2)^2,\end{aligned} \tag{94}$$

which just means $(\mathbf{lebesgue} \oplus \mathbf{lebesgue}^2)^2$ as expected in (28). Again, applying the totaling transformation then produces the desired acceptance ratio.

Unfortunately, our disintegrator returns no result if we change the reversible-jump proposal (92) so that it sometimes jumps without adding noise, for instance

$$\begin{aligned}\zeta'_{\mathbb{R}+\mathbb{R}^2} \;=\; \lambda z.\,\mathbf{case}\ z\ \mathbf{of}\ \mathbf{inl}\,x_0 &\to \left(\tfrac{1}{2}\odot \mathbf{inl}\diamond \mathbf{normal}\,x_0\ 0.1\right)\\ &\quad \oplus \left(\tfrac{1}{2}\odot \mathbf{inr}\diamond \mathbf{do}\ \{d \leftsquigarrow \mathbf{normal}\ 0\ 0.1;\ \mathbf{return}\ (x_0+d, x_0-d)\}\right)\\ \mathbf{inr}\,x &\to \left(\tfrac{1}{2}\odot \mathbf{inl}\diamond \mathbf{return}\ ((\mathbf{fst}\,x+\mathbf{snd}\,x)/2)\right)\\ &\quad \oplus \left(\tfrac{1}{2}\odot \mathbf{inr}\diamond (\mathbf{normal}\,(\mathbf{fst}\,x)\ 0.1\otimes \mathbf{normal}\,(\mathbf{snd}\,x)\ 0.1)\right).\end{aligned} \tag{95}$$

The failure occurs in base inference: we want a base like

$$\begin{aligned}&\left(\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \oplus (\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \otimes\!\multimap \lambda x_1.\,\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor)\right)\\ &\quad \otimes\!\multimap \lambda z.\,\mathbf{case}\ z\ \mathbf{of}\ \mathbf{inl}\,x_0 \to \left(\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \oplus (\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \otimes\!\multimap \lambda x_1'.\,\mathbf{mix}\ \mathrm{ff}\ \lfloor\!\lfloor x_0-(x_1'-x_0) \rfloor\!\rfloor)\right)\\ &\qquad\qquad \mathbf{inr}\,x \to \left(\mathbf{mix}\ \mathrm{ff}\ \lfloor\!\lfloor (\mathbf{fst}\,x+\mathbf{snd}\,x)/2 \rfloor\!\rfloor \oplus (\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \otimes\!\multimap \lambda x_1'.\,\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor)\right)\end{aligned} \tag{96}$$

or perhaps

$$\begin{aligned}&\left(\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \oplus (\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \otimes\!\multimap \lambda x_1.\,\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor)\right)\\ &\quad \otimes\!\multimap \lambda z.\,\left(\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \mathbf{case}\ z\ \mathbf{of}\ \mathbf{inr}\,x \to (\mathbf{fst}\,x+\mathbf{snd}\,x)/2 \rfloor\!\rfloor\right.\\ &\qquad\qquad \left.\oplus (\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \otimes\!\multimap \lambda x_1'.\,\mathbf{mix}\ \mathrm{tt}\ \lfloor\!\lfloor \mathbf{case}\ z\ \mathbf{of}\ \mathbf{inl}\,x_0 \to x_0-(x_1'-x_0) \rfloor\!\rfloor)\right) : \mathbb{B}\,(\mathbb{R}+\mathbb{R}^2)^2.\end{aligned} \tag{97}$$

The suggestion (96) is beyond the reach of our disintegrator because case discrimination is not in our base language (Figures 22 and 25). And the suggestion (97) is beyond the reach of our disintegrator

because the constraints

$$\left(\mathbf{mix}\ \mathrm{ff}\ \wr(\mathbf{fst}\ x + \mathbf{snd}\ x)/2\int,\ \mathtt{B}_1[z]\right) \qquad \text{and} \qquad \left(\mathbf{mix}\ \mathrm{ff}\ \wr x_0 - (x_1' - x_0)\int,\ \mathtt{B}_2[z, x_1']\right) \tag{98}$$

cannot be solved (Section 6.5.1). The failure to solve these constraints can in turn be blamed on the fact that the notion of term equality and matching we implemented in Figure 22 and Section 6.5.1 ("known to be equal") is mere $\alpha$-equivalence: it is not modulo $\beta$-equivalence for sum types, and it does not take into account having emitted the guards **let inr** $x = z$ and **let inl** $x_0 = z$.

### 8.5 Belief update

Evaluation (E6) shows that our disintegrator can use a discrete-continuous mixture base (B1) to find not only densities (E1) but also conditional distributions. We start with the joint distribution $\xi : \mathbb{M}\,(\mathbb{R} \times \mathbb{R})$ in equation (36): we want to infer an unknown random number from its clamped observation. To condition on the observation (the **snd** dimension of $\xi$), we can invoke the restricted base-checking disintegrator (Section 5) by *check* $(swap \diamond \xi)\ b\ t$. As in Section 8.1, there is no solution for the base $b = \mathbf{mix}\,\mathrm{tt}\,\wr\int$ or $b = \mathbf{mix}\,\mathrm{ff}\,\wr 0, 1\int$, but if we let $b = \mathbf{mix}\,\mathrm{tt}\,\wr 0, 1\int$, then we get the following solution (similar to (86), and again simplified by removing **factor** 1 and inlining **let**-bindings):

$$\begin{aligned}
\lambda t.\quad & \mathbf{do}\ \{\mathbf{observe}\ t = 1;\ x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ y \leftsquigarrow \mathbf{normal}\ x\ 1;\\
& \qquad \mathbf{observe}\ 0 < \min\{1, y\};\ \mathbf{observe}\ 1 < y;\ \mathbf{return}\ x\}\\
\oplus\ & \mathbf{do}\ \{\mathbf{observe}\ t \neq 0 \wedge t \neq 1;\ x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ \mathbf{factor}\ (\mathbf{dnorm}\ x\ 1)(t);\\
& \qquad \mathbf{observe}\ 0 < \min\{1, t\};\ \mathbf{observe}\ t \leq 1;\ \mathbf{return}\ x\}\\
\oplus\ & \mathbf{do}\ \{\mathbf{observe}\ t = 0;\ x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ y \leftsquigarrow \mathbf{normal}\ x\ 1;\\
& \qquad \mathbf{observe}\ \min\{1, y\} \leq 0;\ \mathbf{return}\ x\} \qquad : \mathbb{R} \to \mathbb{M}\,\mathbb{R}.
\end{aligned} \tag{99}$$

For each possible observation $t : \mathbb{R}$, only one of the three summands above is nonzero: the first summand if $t = 1$; the second if $0 < t < 1$; the third if $t = 0$. To estimate the distribution of $x$ given $t$, the code of that summand can be executed in a standard way as an importance sampler: $\leftsquigarrow$ makes a random choice; **factor** incurs an importance weight, and **observe** decides whether to reject a sample. As in Example 2.20, symbolic equational reasoning about normal distributions can rewrite $x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ \mathbf{factor}\ (\mathbf{dnorm}\ x\ 1)(t)$ in the second summand to $\mathbf{factor}\ (\mathbf{dnorm}\ 3\ \sqrt{5})(t)$; $x \leftsquigarrow \mathbf{normal}\ \frac{3+4t}{5}\ \frac{2}{\sqrt{5}}$. And as in Section 8.1, symbolic equational reasoning about error functions can rewrite the guards about $y$ in the first and third summands into exact closed-form importance weights. These automatic simplifications [Carette and Shan 2016; Gehr et al. 2016] increase the accuracy and reduce the variance of the importance sampler.

Instead of specifying the base $b = \mathbf{mix}\,\mathrm{tt}\,\wr 0, 1\int$, it works just as well to invoke the unrestricted base-checking disintegrator by *disint* $(swap \diamond \xi)\ (\mathbf{snd} \diamond \xi)\ t$, because the base-inferring disintegrator infers that principal base for $\mathbf{snd} \diamond \xi$.

### 8.6 Gibbs sampling

Evaluation (E7) applies our disintegrator to a discrete-continuous mixture base (B1) to generate conditional distributions that constitute a Gibbs sampler. The target distribution

$$\begin{aligned}
\xi \ = \ & \mathbf{do}\ \{x \leftsquigarrow \mathbf{normal}\ 3\ 2;\\
& \qquad y_1 \leftsquigarrow \mathbf{normal}\ x\ 1;\\
& \qquad y_2 \leftsquigarrow \mathbf{normal}\ x\ 1;\\
& \qquad \mathbf{return}\ (\max\{0, \min\{1, y_1\}\}, \max\{0, \min\{1, y_2\}\})\} : \mathbb{M}\,\mathbb{R}^2
\end{aligned} \tag{100}$$

is generated by making two noisy measurements $y_1, y_2$ of the same random variable $x$. Before the measurements are returned, they are clamped to between 0 and 1. This is a very simple instance of a Tobit model.

A Gibbs sampler for $\xi : \mathbb{M}\,\mathbb{R}^2$ alternates repeatedly between sampling the clamped $y_2$ given the clamped $y_1$ and sampling the clamped $y_1$ given the clamped $y_2$. By symmetry, we only discuss the former conditional distribution. To find it, we can invoke the restricted base-checking disintegrator (Section 5) by $\mathit{check}\ \xi\ (\mathbf{mix}\ \mathrm{tt}\ \lceil 0, 1\rfloor)\ t$, where the variable $t : \mathbb{R}$ represents the clamped $y_1$. (Just as well, we can invoke the unrestricted base-checking disintegrator by $\mathit{disint}\ \xi\ (\mathbf{fst} \diamond \xi)\ t$.) We get the following solution (again simplified by removing **factor** 1 and inlining **let**-bindings):

$$\begin{aligned}
\lambda t.\quad & \mathbf{do}\ \{\mathbf{observe}\ t = 1;\ x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ y_1 \leftsquigarrow \mathbf{normal}\ x\ 1;\ y_2 \leftsquigarrow \mathbf{normal}\ x\ 1;\\
& \qquad \mathbf{observe}\ 0 < \min\{1, y_1\};\ \mathbf{observe}\ 1 < y_1;\ \mathbf{return}\ \max\{0, \min\{1, y_2\}\}\}\\
\oplus\ & \mathbf{do}\ \{\mathbf{observe}\ t \neq 0 \wedge t \neq 1;\ x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ \mathbf{factor}\ (\mathbf{dnorm}\ x\ 1)(t);\ y_2 \leftsquigarrow \mathbf{normal}\ x\ 1;\\
& \qquad \mathbf{observe}\ 0 < \min\{1, t\};\ \mathbf{observe}\ t \leq 1;\ \mathbf{return}\ \max\{0, \min\{1, y_2\}\}\}\\
\oplus\ & \mathbf{do}\ \{\mathbf{observe}\ t = 0;\ x \leftsquigarrow \mathbf{normal}\ 3\ 2;\ y_1 \leftsquigarrow \mathbf{normal}\ x\ 1;\ y_2 \leftsquigarrow \mathbf{normal}\ x\ 1;\\
& \qquad \mathbf{observe}\ \min\{1, y_1\} \leq 0;\ \mathbf{return}\ \max\{0, \min\{1, y_2\}\}\} \qquad : \mathbb{R} \to \mathbb{M}\,\mathbb{R}.
\end{aligned} \tag{101}$$

As in Section 8.5, for each $t : \mathbb{R}$, only one of the three summands above is nonzero. Moreover, automatic simplification [Carette and Shan 2016; Gehr et al. 2016] can rewrite the solution to

$$\begin{aligned}
\lambda t.\quad & \mathbf{do}\ \{\mathbf{observe}\ t = 1;\ y_1 \leftsquigarrow \mathbf{normal}\ 3\ \sqrt{5};\ y_2 \leftsquigarrow \mathbf{normal}\ \tfrac{3+4y_1}{5}\ \tfrac{3}{\sqrt{5}};\\
& \qquad \mathbf{observe}\ 0 < \min\{1, y_1\};\ \mathbf{observe}\ 1 < y_1;\ \mathbf{return}\ \max\{0, \min\{1, y_2\}\}\}\\
\oplus\ & \mathbf{do}\ \{\mathbf{observe}\ t \neq 0 \wedge t \neq 1;\ \mathbf{factor}\ (\mathbf{dnorm}\ 3\ \sqrt{5})(t);\ y_2 \leftsquigarrow \mathbf{normal}\ \tfrac{3+4t}{5}\ \tfrac{3}{\sqrt{5}};\\
& \qquad \mathbf{observe}\ 0 < \min\{1, t\};\ \mathbf{observe}\ t \leq 1;\ \mathbf{return}\ \max\{0, \min\{1, y_2\}\}\}\\
\oplus\ & \mathbf{do}\ \{\mathbf{observe}\ t = 0;\ y_1 \leftsquigarrow \mathbf{normal}\ 3\ \sqrt{5};\ y_2 \leftsquigarrow \mathbf{normal}\ \tfrac{3+4y_1}{5}\ \tfrac{3}{\sqrt{5}};\\
& \qquad \mathbf{observe}\ \min\{1, y_1\} \leq 0;\ \mathbf{return}\ \max\{0, \min\{1, y_2\}\}\} \qquad : \mathbb{R} \to \mathbb{M}\,\mathbb{R}.
\end{aligned} \tag{102}$$

By drawing $y_1$ from a truncated normal distribution and drawing $y_2$ from a normal distribution, we can run this simplified code (102) to sample the clamped $y_2$ given $t$, the clamped $y_1$. Thus, disintegration followed by simplification produces the exact code for a Gibbs update step.

If we make the target distribution (100) a more substantial Tobit (Bayesian censored linear regression) model by including more than one coefficient $x$ and more than two measurements $y_1, y_2$, then the disintegrator still succeeds. However, it is difficult to sample Gibbs updates from a distribution conditioned on multiple censored measurements. Instead, it is common to include the coefficients $x$ in the target distribution and update them as part of Gibbs sampling. Our disintegrator can generate the code for those updates as well.

## 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], Gehr et al. [2016], and Walia et al. [2019]: 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 (Section 1.2 and Example 2.11), 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 **lebesgue**, the Lebesgue measure over $\mathbb{R}$.
- Bhat et al.'s density calculator [2012, 2013] handles tuples. That is, it allows base measures $\mathbf{lebesgue}^n$ where $n$ is a concrete natural number. Bhat et al.'s *stock measure* is our initial *genBase* function (42).
- Narayanan and Shan's disintegrator [2017] handles variable-length arrays without unrolling them. That is, it allows base measures $\mathbf{lebesgue}^n$ for symbolic $n$.
- Roberts et al.'s density calculator [2019] handles tuples and variable-length arrays as well as their disjoint sums, for the purpose of automating reversible-jump Metropolis-Hastings sampling. Because it is specialized to disjoint sums of tuples, it is not clear how it might handle tuples that contain disjoint sums. And because it does not reason about discrete-continuous mixtures and dependent products (as in (27)), it is not clear how it might handle proposal kernels that mix single-site and multi-site updating.

Our base-measure language (Figure 22) 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 [2017] and Roberts et al.'s [2019] handling of arrays, and is the first to allow different base measures over the same type.

*Soundness.* 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.

*Semantics.* 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 3.2.1 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.

*Completeness.* 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 force it to fail (such as $\lhd\ (x + x)$), and we demonstrate its utility only empirically (Section 8). This incompleteness is not explained by Ackerman et al.'s result [2011, 2017] 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.

*Continuations.* 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 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.

*Inference.* 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].

## 10 FUTURE WORK

Section 8.4 describes a common kind of reversible-jump proposal kernel for which our disintegrator fails to compute the Metropolis-Hastings acceptance ratio due to its incomplete handling of sum types. We leave it to future work to handle this case.

Section 9 mentions that Narayanan and Shan's disintegrator [2017] and Roberts et al.'s density calculator [2019] handle variable-length arrays whereas our new disintegrator does not. Thus, another direction for future work is to handle array programs without unrolling them. In other words, we would like to add *plates* ($n$-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.

Finally, our unrestricted base-checking disintegrator in Section 7 assumes that, not only does the given base measure $m_2$ have a density with respect to the inferred base measure $b$ (which is guaranteed by inference), but also vice versa. For example, to compute the density of **normal** 0 1 with respect to $m_2 =$ **normal** 3 2 in Figure 5, the definition (81) first infers that $m_2$ has the density **dnorm** 3 2 with respect to the Lebesgue measure $b =$ **mix** tt $\wr\int$, then uses Proposition 2.10(2) to conclude that the reciprocal of **dnorm** 3 2 is a density of $b$ with respect to $m_2$, without checking the assumption that **dnorm** 3 2 is almost everywhere finite and nonzero. It would be nice to check this assumption algorithmically. In fact, we only need to check in (81) that $|\mathit{check}\ m_2'\ b\ t|$ is $b$-almost everywhere finite and nonzero at those places $t$ where $|\mathit{check}\ m_1\ b\ t|$ is nonzero. We leave this checking for future work.

## ACKNOWLEDGMENTS

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

## REFERENCES


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

Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. 2017. On Computability and Disintegration. *Mathematical Structures in Computer Science* 27, 8 (2017), 1287–1314.

Hadi Mohasel Afshar, Scott Sanner, and Christfried Webers. 2016. Closed-Form Gibbs Sampling for Graphical Models with Algebraic Constraints. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*. AAAI Press.

Amal Ahmed and Matthias Blume. 2011. An Equivalence-Preserving CPS Translation via Multi-language Semantics. In *Proceedings of the 2011 ACM SIGPLAN International Conference on Functional Programming*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM Press, 431–444.

Sooraj Bhat, Ashish Agarwal, Richard Vuduc, and Alexander Gray. 2012. A Type Theory for Probability Density Functions. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 545–556.

Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio V. Russo. 2013. Deriving Probability Density Functions from Probabilistic Functional Programs. In *Proceedings of TACAS 2013: 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer, 508–522.

Richard S. Bird and Oege de Moor. 1996. *Algebra of Programming*. Prentice-Hall.

Anders Bondorf. 1992. Improving Binding Times Without Explicit CPS-Conversion. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming (Lisp Pointers)*, William D. Clinger (Ed.), Vol. V(1). ACM Press, 1–10.

Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2016. A Lambda-Calculus Foundation for Universal Probabilistic Programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM Press, 33–46.

Wray L. Buntine. 1994. Operations for Learning with Graphical Models. *Journal of Artificial Intelligence Research* 2 (1994), 159–225.

Jacques Carette and Chung-chieh Shan. 2016. Simplifying Probabilistic Programs Using Computer Algebra. In *Practical Aspects of Declarative Languages: 18th International Symposium, PADL 2016 (Lecture Notes in Computer Science)*, Marco Gavanelli and John H. Reppy (Eds.). Springer, 135–152.

Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* 76, 1 (2017), 1–32.

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

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

Ryan Culpepper and Andrew Cobb. 2017. Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring. In *Programming Languages and Systems: Proceedings of ESOP 2017, 26th European Symposium on Programming (Lecture Notes in Computer Science)*, Yang Hongseok (Ed.). Springer, 368–392.

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: a General-Purpose Probabilistic Programming System with Programmable Inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM Press, 221–236.

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*. ACM Press, 151–160.

Bruno de Finetti. 1970. *Teoria delle Probabilità: Sintesi Introduttiva con Appendice Critica*. Vol. 1. Giulio Einaudi, Torino. Translated as de Finetti 1974.

Bruno de Finetti. 1972. *Probability, Induction, and Statistics*. Wiley.

Bruno de Finetti. 1974. *Theory of Probability: A Critical Introductory Treatment*. Vol. 1. Wiley.

Luc Devroye. 1986. *Non-Uniform Random Variate Generation*. Springer.

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

Sebastian Fischer, Josep Silva, Salvador Tamarit, and Germán Vidal. 2008. Preserving Sharing in the Partial Evaluation of Lazy Functional Programs. In *Revised Selected Papers from LOPSTR 2007: 17th International Symposium on Logic-Based Program Synthesis and Transformation (Lecture Notes in Computer Science)*, Andy King (Ed.). Springer, 74–89.

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

Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Proceedings of the 28th International Conference on Computer Aided Verification, Part I (Lecture Notes in Computer Science)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 62–83.

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

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

Michèle Giry. 1982. A Categorical Approach to Probability Theory. In *Categorical Aspects of Topology and Analysis: Proceedings of an International Conference Held at Carleton University, Ottawa, August 11–15, 1981*, Bernhard Banaschewski (Ed.). Springer, 68–85.

Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, David Allen McAllester and Petri Myllymäki (Eds.). AUAI Press, 220–229.

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

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

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

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A Convenient Category for Higher-Order Probability Theory. In *LICS 2017: Proceedings of the 32nd Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1–12.

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

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

John Hughes. 1995. The Design of a Pretty-Printing Library. In *Advanced Functional Programming: 1st International Spring School on Advanced Functional Programming Techniques*, Johan Jeuring and Erik Meijer (Eds.). Number 925 in Lecture Notes in Computer Science. Springer, 53–96.

Graham Hutton and Erik Meijer. 1996. Back to Basics: Deriving Representation Changers Functionally. *Journal of Functional Programming* 6, 1 (1996), 181–188.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall.

Jesper Jørgensen. 1992. Generating a Compiler for a Lazy Language by Partial Evaluation. In *Conference Record of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Ravi Sethi (Ed.). ACM Press, 258–268.

Herman Kahn and T. E. Harris. 1951. Estimation of Particle Transmission by Random Sampling. *National Bureau of Standards Applied Mathematics Series* 12 (1951), 27–30.

Anders Kock. 2012. Commutative Monads as a Theory of Distributions. *Theory and Applications of Categories* 26, 4 (2012), 97–131.

Julia L. Lawall and Olivier Danvy. 1994. Continuation-Based Partial Evaluation. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*. ACM Press, 227–238.

David J. Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. 2000. WinBUGS—A Bayesian Modelling Framework: Concepts, Structure, and Extensibility. *Statistics and Computing* 10, 4 (2000), 325–337.

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

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

Wazim Mohammed Ismail and Chung-chieh Shan. 2016. Deriving a Probability Density Calculator (Functional Pearl). In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM Press, 47–59.

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *Proceedings of FLOPS 2016: 13th International Symposium on Functional and Logic Programming (Lecture Notes in Computer Science)*, Oleg Kiselyov and Andy King (Eds.). Springer, 62–79.

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

Otton Nikodym. 1930. Sur une Généralisation des Intégrales de M. J. Radon. *Fundamenta Mathematicae* 15 (1930), 131–179.

Avi Pfeffer. 2009. CTPPL: A Continuous Time Probabilistic Programming Language. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, Craig Boutilier (Ed.). 1943–1950.

Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press.

Gordon Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94.

Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the $\lambda$-Calculus. *Theoretical Computer Science* 1, 2 (1975), 125–159.

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

Norman Ramsey and Avi Pfeffer. 2002. Stochastic Lambda Calculus and Monads of Probability Distributions. In *Conference Record of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 154–165.

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM National Conference*, Vol. 2. ACM Press, 717–740. Reprinted with introduction in *Higher-Order and Symbolic Computation* 11, 4 (1998), 363–397.

David A. Roberts, Marcus Gallagher, and Thomas Taimre. 2019. Reversible Jump Probabilistic Programming. In *Proceedings of AISTATS 2019: 22nd International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Masashi Sugiyama (Eds.). 634–643.

Halsey L. Royden. 1988. *Real Analysis* (third ed.). Prentice-Hall.

Tetsuya Sato, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu. 2019. Formal Verification of Higher-Order Probabilistic Programs: Reasoning about Approximation, Convergence, Bayesian Inference, and Optimization. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 38:1–38:30.

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

Chung-chieh Shan and Norman Ramsey. 2017. Exact Bayesian Inference by Symbolic Disintegration. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM Press, 130–144.

Robert J. Shiller. 1999. The ET Interview: Professor James Tobin. *Econometric Theory* 15, 6 (1999), 867–900.

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

Robert D. Tennent. 1973. Mathematical Semantics of SNOBOL 4. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, Patrick C. Fischer and Jeffrey D. Ullman (Eds.). ACM Press, 95–107.

Hayo Thielecke. 2003. From Control Effects to Typed Continuation Passing. In *Conference Record of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 139–149.

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

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

Matthijs Vákár, Ohad Kammar, and Sam Staton. 2019. A Domain Theory for Statistical Probabilistic Programming. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 36:1–36:29.

Matthijs Vákár and Luke Ong. 2018. *On S-Finite Measures and Kernels.* e-Print 1810.01837. arXiv.org. https://arxiv.org/abs/1810.01837

Rajan Walia, Praveen Narayanan, Jacques Carette, Sam Tobin-Hochstadt, and Chung-chieh Shan. 2019. From High-Level Inference Algorithms to Efficient Code. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 98:1–98:30.

Mitchell Wand. 1987a. Complete Type Inference for Simple Objects. In *LICS '87: Proceedings of the Symposium on Logic in Computer Science*. IEEE Computer Society Press, 37–44.

Mitchell Wand. 1987b. A Simple Algorithm and Proof for Type Inference. *Fundamenta Informaticae* 10, 2 (1987), 115–122.

Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. 2018. Contextual Equivalence for a Probabilistic Language with Continuous Random Variables and Recursion. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 87:1–87:30.

David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. 2011. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *Proceedings of AISTATS 2011: 14th International Conference on Artificial Intelligence and Statistics (JMLR Workshop and Conference Proceedings)*, Geoffrey Gordon, David Dunson, and Miroslav Dudík (Eds.). 770–778.

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

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

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