Conditional probability by lazy partial evaluation

Chung-chieh Shan
Indiana University
ccshan@indiana.edu

Abstract

This is the text of the abstract.

Categories and Subject Descriptors G.3 [*Probability and Statistics*]: distribution functions

Keywords keyword1, keyword2

1. Introduction

Probability distributions are a popular way to model and handle uncertainty. In particular, the typical *Bayesian* reasoner begins with a *prior* probability distribution on all possible worlds, *observes* the actual world, then *conditions* the prior distribution to obtain a *posterior* distribution on those possible worlds that match the observations. It is common and convenient to specify a distribution by composing a *generative story*, which is a procedure that picks a world randomly, usually by modeling the relevant aspects of how the world comes to be.

TODO our contributions

1.1 Background example

We illustrate the starting point of this paper with an example. Suppose we observed the players of a one-on-one game, perhaps to match them up to make future games more fun. For simplicity, suppose we saw just one game, in which the player Alice beat the player Bob. This outcome may be due to Alice's skill level (whatever it means) being higher than Bob's, or due to Alice being lucky in this particular game (whatever it means). Regardless, we can ask how Alice's skill level compares to Bob's, given that Alice beat Bob. To model the situation, we can write the following generative story:

$$\begin{array}{l}\textbf{do}\ \{a \leftsquigarrow \textbf{normal}\ 10\ 3;\\ \quad b \leftsquigarrow \textbf{normal}\ 10\ 3;\\ \quad l \leftsquigarrow \textbf{normal}\ \ 0\ 2;\\ \quad \textbf{let true} = (l < a - b);\\ \quad \textbf{return}\ (a,b)\}\end{array} \tag{1}$$

The first line of this program means to pick a real number a from the normal distribution with mean 10 and standard deviation 3. The second line picks b from the same distribution. These numbers model the skill levels of Alice and Bob. The third line picks l, which models how much luck Bob has over Alice in this game. These three lines together define a *prior* distribution on $\mathbb{R}^3$, in which each point (a,b,l) is a possible world. Figure 1 depicts this distribution approximately, as a cloud of points (both red triangles and blue circles). The fourth line performs a non-exhaustive pattern-match to restrict the distribution to the part where $l < a - b$. This step incorporates our observation that Alice beat Bob. The first four lines together define the *posterior* distribution on $\mathbb{R}^3$, shown as blue circles in Figure 1. The last line projects this distribution from $\mathbb{R}^3$ to $\mathbb{R}^2$.

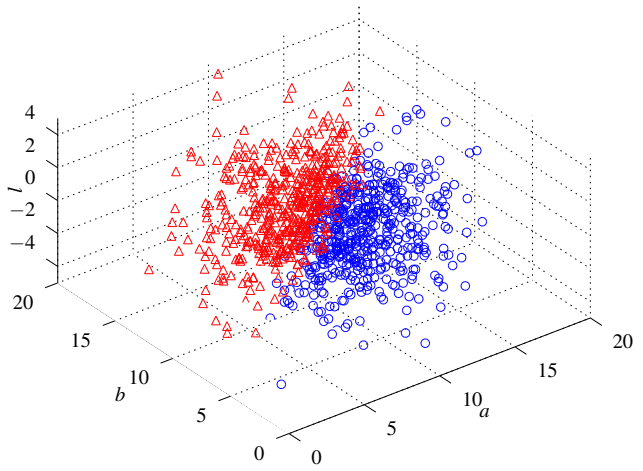

Figure 1. An approximate prior distribution: 1000 random samples generated by the first three lines of (1) (one line for each dimension). The red triangles are samples rejected by the fourth line of (1). The blue circles are samples accepted by that line.

A simple way to answer a question about the distribution is to interpret this program as a *sampler*—that is, a probabilistic algorithm that generates a random point. For instance, it is well-studied how to generate a pseudorandom number from a normal distribution. We run this sampler 1000 times, say (as shown in Figure 1). These resulting points approximate our uncertainty about Alice and Bob's skill levels. For example, as shown in Figure 2, to estimate the probability that Alice is more skilled than Bob, we can compute the proportion of resulting points where $a > b$.

To make this estimation method more accurate, we can change the program (1) to one that denotes the same distribution but makes fewer random choices. For example, instead of picking l then testing $l < a - b$, the following equivalent program generate samples with uneven *importance weights*:

$$\begin{array}{l}\textbf{do}\ \{a \leftsquigarrow \textbf{normal}\ 10\ 3;\\ \quad b \leftsquigarrow \textbf{normal}\ 10\ 3;\\ \quad \textbf{factor}\ \dfrac{1+\textbf{erf}\left((a-b)/(2\times\sqrt{2})\right)}{2};\\ \quad \textbf{return}\ (a,b)\}\end{array} \tag{2}$$

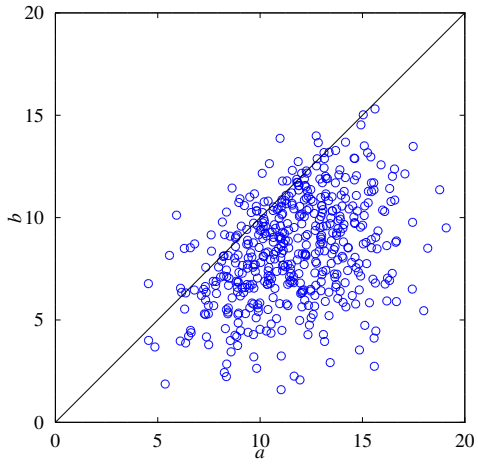

Figure 2. A boolean query on an approximate posterior distribution: fewer than 1000 random samples generated by (1). The samples to the right of the diagonal line are where $a > b$.

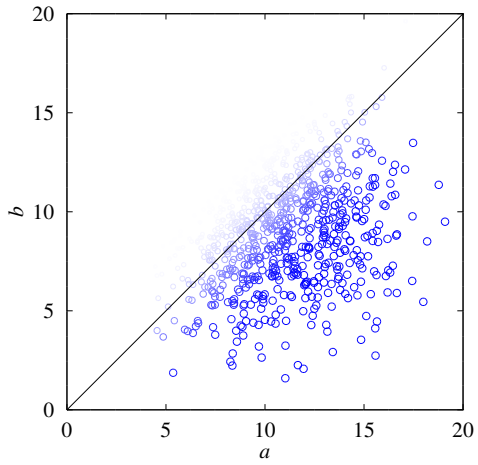

Figure 3. A boolean query on a better approximate posterior distribution: 1000 weighted random samples generated by (2). The size and darkness of each circle reflect its weight. As in Figure 2, the samples to the right of the diagonal line are where $a > b$.

Interpreted as a sampler, this program does not pick any concrete l, but rather attaches the weight $\frac{1+\mathbf{erf}((a-b)/(2\times\sqrt{2}))}{2}$ (instead of the default weight 1) to each generated point (a,b). This weight is the probability that, had we chosen a concrete l as in (1), we would have chosen an l that passes the test $l < a - b$. We can think of the weight as softening the rejection due to a failed pattern-match. As shown in Figure 3, to estimate the probability that Alice is more skilled than Bob, we should divide the total weight of points where $a > b$ by the total weight of all points. This estimate varies less from run to run than the estimate from (1) does, because this program makes fewer random choices while denoting the same distribution. That is good.

We wait until Section 3 to detail how expressions denote distributions. The gist of the present example is that we turned a prior distribution expressed as a program (namely the first three lines of (1)) into a posterior distribution expressed as a more efficient program (namely (2)), in two steps:

1. We expressed an observation as a non-exhaustive pattern-match (the fourth line of (1)).
2. We justified an optimization by denotation equality (between the third and fourth lines of (1) and the third line of (2)).

1.2 The need to generalize and mechanize conditioning

In general, just as it is useful to express any distribution as a sampler program, it is useful to express a posterior distribution as a sampler program such as (1) or (2). There are a variety of reasons:

1. A human can then read the code and understand the posterior as a distribution in its own right, then reason about it equationally.
2. A larger program can include the posterior. Posteriors are an essential part of many probabilistic inference algorithms (such as *Monte Carlo Markov chain* methods [13]) as well as probabilistic models (for reasoning about reasoning [17]).
3. A program transformation can generate the posterior from the prior. For example, the fourth line of (1) could have been added automatically.
4. A human or a machine such as a compiler pass can simplify and optimize the posterior. For example, we are building a probabilistic programming system Hakaru that can turn the third and fourth lines of (1) into the third line of (2).
5. A robot that receives a stream of observations as the world changes over time can update the posterior to reflect the evolution and uncertainty of its knowledge [10].

Before we can express the posterior distribution as a program, we need to specify what it is, so as to inform the design and justify the correctness of each of the applications listed above. In other words, we need to specify *conditioning*, the mathematical operation that turns the prior into the posterior. Unfortunately, the specification of conditioning in current probabilistic programming languages assumes that the condition has non-zero probability. For example, the probability of $l < a - b$ in (1) and Figure 1 is $1/2$. If this probability were zero, then we would get the empty distribution with no samples, which is unworthy of being called a posterior.

In practice, it is very common for the condition to have zero probability. TODO: Motivating example of zero-probability condition from July 2014 talk.

1.3 Our contributions

We define a probabilistic language whose types are measurable spaces that can be uncountable and whose terms can be interpreted simultaneously as samplers, measures, and functionals.

We generalize the specification of conditioning to when the condition has zero probability, by adapting to programming languages the notion of *disintegration* advertised by Chang and Pollard [5].

We implement this specification as a program transformation that generalizes both Bhat et al.'s density calculator [1, 2] and Fischer et al.'s sharing-preserving lazy partial evaluator [8].

2. Samplers, measures, and functionals

For its motivation, intuition, and correctness, our work relies on a three-way correspondence between *samplers*, *measures*, and *functionals*. Thus, we introduce these concepts and detail the correspondence in this section.

2.1 Samplers

A *sampler* is a program that uses randomness to produce a result. We can think of each run of the program as an experiment, and the result of the run is the *outcome* of the experiment. A typical sampler is composed of building blocks that are primitive samplers. For

example, one primitive sampler might be to choose a real number uniformly at random between 0 and 1. In our language, we can express this sampler by the primitive expression **random**. One way to flip a fair coin (that is, to choose between two outcomes with equal probability) is to perform **random** twice then see which result is bigger. Because our language is monadic, we can express this composed sampler by the expression

$$\mathbf{do}\ \{x \leftsquigarrow \mathbf{random}; y \leftsquigarrow \mathbf{random}; \mathbf{return}\ (x < y)\}. \tag{3}$$

Here **return** is the monad unit operation (`return` in Haskell) and $\mathbf{do}\ \{x \leftsquigarrow \cdots; \cdots\}$ is the monad bind operation (`>>=` in Haskell). The type of **random** is $\mathbb{M}\mathbb{R}$ (where $\mathbb{M}$ is the monad type constructor), because **random** is not a real number but rather a sampler that produces a real number. The type of (3) is $\mathbb{M}(\mathbb{1}+\mathbb{1})$, because our numeric comparison $<$ returns the sum of unit types $\mathbb{1}+\mathbb{1}$ (the value **inl** $()$ for **true** and the value **inr** $()$ for **false**).

To optimize and reason about programs, we want to consider many samplers equivalent that do not describe exactly the same procedure. For example, the following are two other perfectly correct ways to flip a fair coin in our language:

$$\mathbf{do}\ \{y \leftsquigarrow \mathbf{random}; x \leftsquigarrow \mathbf{random}; \mathbf{return}\ (x < y)\} \tag{4}$$

$$\mathbf{do}\ \{x \leftsquigarrow \mathbf{random}; \mathbf{return}\ (x < 1/2)\} \tag{5}$$

If we are running these programs using a pseudo-random number generator, then we may well want to optimize the first two ways to the third way, despite (or due to) the third way using less randomness. Such an optimization is justified by the fact that the three ways denote the same *measure*, even though not the same sampler [15].

2.2 Measures

A *measure* is a mathematical function that maps sets to non-negative real numbers. In the slightly unfortunate standard terminology, the measure is said to *measure* sets and return their *measures*.

1. For example, **random** in our language denotes the *uniform probability measure on* $(0,1)$: given an interval (a,b) of real numbers, it returns the length of the intersection of the intervals $(0,1)$ and (a,b). Thus the measure of the interval $(2/3,2)$ is $1/3$. This matches the fact that, a third of the time, choosing a real number uniformly at random between 0 and 1 produces a number between $2/3$ and 2.

A different measure may well give a different result from measuring the same set.

2. For example, **return** x denotes the *Dirac measure at* x: given the same set S to be measured, whether S is an interval (a,b), it returns 1 if $x \in S$, and 0 otherwise. [TODO: Use x only for a variable and a,b only for atomic terms?]

The general idea is that each sampler corresponds to the measure that, given a set of outcomes, tells the *probability* that running the sampler will produce an outcome in the given set. The deterministic sampler **return** x always produces the same outcome x, so the probability that the outcome is in the measured set is either 1 or 0.

To avoid pathological cases, a measure is not required to measure every set of real numbers. Rather, we introduce the notion of a *measurable space*. A measurable space is a set A, equipped with a notion of what subsets of A can be measured. The complement of a measurable set and the union of a countable collection of measurable sets must be measurable. In particular, the empty set $\{\}$ and the full set A must both be measurable, because the empty set is the nullary union and the full set is the complement of the empty set.

Each type in our language is a measurable space. For example, the type $\mathbb{1}$ in our language is the singleton set $\{()\}$, equipped with the notion that the empty set $\{\}$ and the full set $\{()\}$ are both measurable as required. Another base type $\mathbb{R}$ in our language is the set of real numbers, equipped with the notion that every interval can be measured—so every set of real numbers made from intervals by complement and countable union must also be measurable.

It is easy to add other measurable spaces as base types. For example, we could add the measurable space of integers. Yet another useful measurable space is $[0,\infty]$, the set of non-negative real numbers augmented with positive infinity, again equipped with the notion that every interval can be measured. We don't need this type in our language, but it is an essential measurable space for semantics.

Mathematically speaking, then, a measurable space is a pair

$$\alpha = (\mathrm{set}(\alpha), \mathrm{measurable}(\alpha)), \tag{6}$$

where $\mathrm{set}(\alpha)$ is a set and $\mathrm{measurable}(\alpha)$ is a set of subsets of $\mathrm{set}(\alpha)$ that is closed under complement and countable union. A measure μ on α is a function from $\mathrm{measurable}(\alpha)$ to $[0,\infty]$, such that

$$\mu(S_1 \cup S_2 \cup \cdots) = \mu(S_1) + \mu(S_2) + \cdots \tag{7}$$

for every countable collection $S_1, S_2, \ldots \in \mathrm{measurable}(\alpha)$ of pairwise-disjoint measurable sets. (In particular, the empty collection forces $\mu(\{\})$ to be 0.) We sometimes write a measurable space to mean its underlying set. For example, we write $3 \in [0,\infty]$ to mean $3 \in \mathrm{set}([0,\infty])$.

There is a category of measurable spaces [9]. Its morphisms are the *measurable functions*. A measurable function is a function such that the inverse image of every measurable set is measurable. Formally, given two measurable spaces α and β, a *measurable function* $f \in \alpha \to \beta$ is a function $f \in \mathrm{set}(\alpha) \to \mathrm{set}(\beta)$ such that $f^{-1}(T) \in \mathrm{measurable}(\alpha)$ for every $T \in \mathrm{measurable}(\beta)$. It is a standard exercise to prove that, if $S \in \mathrm{measurable}(\alpha)$, then the indicator function $S_\star \in \alpha \to [0,\infty]$ defined by

$$S_\star(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases} \tag{8}$$

is in fact measurable. Another useful exercise is to prove that, if $c_1, c_2, \ldots \in \alpha \to [0,\infty]$ is a (countable) sequence of measurable functions, and the sequence is *increasing* in the sense that $c_i(x) \le c_j(x)$ for all $x \in \alpha$ whenever $i \le j$, then the pointwise limit function

$$\lambda x.\ \lim_{n\to\infty} c_n(x) \quad \in \quad \alpha \to [0,\infty] \tag{9}$$

is again measurable. [TODO: Don't use the letter c for both integrands and metalanguage continuations.]

Three type constructors $\times$, $+$, and $\mathbb{M}$ in our language build bigger measurable spaces out of smaller ones, as follows.

Given two measurable spaces α and β, we can take their product as well as disjoint union. The product $\alpha \times \beta$ is the Cartesian product of sets $\mathrm{set}(\alpha) \times \mathrm{set}(\beta)$, equipped with the notion that any Cartesian product $S \times T$ of two measurable sets $S \in \mathrm{measurable}(\alpha)$ and $T \in \mathrm{measurable}(\beta)$ can be measured (as is every set made from those Cartesian products by complement and countable union). Thus,

$$(x,y) \in \alpha \times \beta \qquad \text{if} \qquad x \in \alpha \qquad \text{and} \qquad y \in \beta. \tag{10}$$

The disjoint union $\alpha + \beta$ is the disjoint union of sets $\mathrm{set}(\alpha) + \mathrm{set}(\beta)$, equipped with the notion that any disjoint union $S + T$ of two measurable sets $S \in \mathrm{measurable}(\alpha)$ and $T \in \mathrm{measurable}(\beta)$ can be measured. As alluded to above, we write

$$\mathbf{inl}\ x \in \alpha + \beta \qquad \text{if} \qquad x \in \alpha, \tag{11}$$

$$\mathbf{inr}\ y \in \alpha + \beta \qquad \text{if} \qquad y \in \beta; \tag{12}$$

and we abbreviate **inl** $()$ as **true** and **inr** $()$ as **false**.

Given a measurable space α, it turns out we can turn the set of measures on it into a measurable space $\mathbb{M}\alpha$. To do so, we

equip the set with the notion that, for any two measurable sets $S \in \text{measurable}(\alpha)$ and $T \in \text{measurable}([0,\infty])$, the set of measures

$$\{\, \mu \mid \mu(S) \in T \,\} \tag{13}$$

is measurable. This construction $\mathbb{M}$ is a monad on the category of measurable spaces [9]. A measurable function from α to $\mathbb{M}\beta$ (in other words, a morphism from α to β in the Kleisli category) is called a *kernel* from α to β.

2.3 Functionals

We have just described the measure built by the monad unit operation **return**, but not the monad bind operation. We need the monad bind operation not only to define the denotational semantics of our language, but also to define conditioning. To describe the monad bind operation, we turn to a certain class of *functionals* that are in one-to-one correspondence with measures.

A functional is just a higher-order function. Given a measure μ on a measurable space α, let's consider the functional $\mu^\star$ that takes any function c from α and integrates it. As long as the integrand c is a *measurable* function from α to $[0,\infty]$ (that is, $c \in \alpha \to [0,\infty]$), we can define this integral $\mu^\star(c) \in [0,\infty]$. This definition of integration is called *Lebesgue integration*. Its basic idea is to slice the integrand horizontally into sets measurable in α, then take the limit as the height of each slice approaches zero and the total height of all slices approaches infinity. [TODO: Slicing picture, from Wikipedia?]

Thus, to each measure $\mu \in \mathbb{M}\alpha$ corresponds an integration functional $\mu^\star$, which is a function from $\alpha \to [0,\infty]$ to $[0,\infty]$. This functional $\mu^\star$ enjoys three important properties:

1. It extends μ, in that $\mu^\star(S_\star) = \mu(S)$ for all $S \in \text{measurable}(\alpha)$.
2. It is *linear*, in that
$$\mu^\star(\lambda x.\, r \times c(x)) = r \times \mu^\star(c) \tag{14}$$
$$\mu^\star(\lambda x.\, c(x) + c'(x)) = \mu^\star(c) + \mu^\star(c') \tag{15}$$
for all $c, c' \in \alpha \to [0,\infty]$ and $r \in [0,\infty]$.
3. It satisfies *monotone convergence*, in that
$$\mu^\star(\lambda x.\, \lim_{n\to\infty} c_n(x)) = \lim_{n\to\infty} \mu^\star(c_n) \tag{16}$$
for all increasing $c_1, c_2, \ldots \in \alpha \to [0,\infty]$.

Moreover, it turns out that $\mu^\star$ is the *unique* functional with these properties. In other words, there is a one-to-one correspondence between measures μ and linear functionals satisfying monotone convergence $\mu^\star$. In fact, many useful measures are most easily defined by specifying their corresponding functionals—for example,

1. The uniform probability measure on $(0,1)$ is the measure $\mu \in \mathbb{M}\mathbb{R}$ such that $\mu^\star(c) = \int_0^1 c(x)\,dx$.
2. Given an element $x \in \alpha$, the Dirac measure at x is the measure $\mu \in \mathbb{M}\alpha$ such that $\mu^\star(c) = c(x)$.
3. Given a measure $\mu \in \mathbb{M}\alpha$ and a kernel $\nu \in \alpha \to \mathbb{M}\beta$, the monad bind operation produces the measure $\xi \in \mathbb{M}\beta$ such that $\xi^\star(c) = \mu^\star(\lambda x.\, (\nu x)^\star(c))$.

Therefore, de Finetti [7] and Pollard [16] advocate omitting the stars altogether. In other words, they advocate identifying each set $S \in \text{measurable}(\alpha)$ with its indicator function $S_\star \in \alpha \to [0,\infty]$, and each measure $\mu \in \mathbb{M}\alpha$ with its integration functional $\mu^\star \in (\alpha \to [0,\infty]) \to [0,\infty]$. Under this proposal, the monad operations are exactly those of the continuation monad.

Without going that far, we introduce two pieces of notation that make applying the correspondence more concise. First, just as we already abbreviate "the function f such that $f(x) = \cdots$" to "$\lambda x.$", we abbreviate "the measure μ such that $\mu^\star(c) = \cdots$" to "$\lambda^\star c.$". Second [TODO: drop this notation?], we write $\mu^\star(c)$ as $\mu \star c$, so that we can write $\mu^\star(\lambda x. \ldots)$ as $\mu \star \lambda x. \ldots$ without parentheses. This way, we can write that

1. the uniform probability measure on $(0,1)$ is $\lambda^\star c.\, \int_0^1 c(x)\,dx$,
2. monad unit produces the measure $\lambda^\star c.\, c(x)$, and
3. monad bind produces the measure $\lambda^\star c.\, \mu \star \lambda x.\, \nu x \star c$.

2.4

For conditioning and density, it's useful to generalize from probability measures to all measures (actually, to sigma-finite measures/kernels), and from samplers to importance samplers.

Return to motivating examples.

State precisely the three sets, notions of equivalence, and correspondences preserving said notions of equivalence.

2.5 What is disintegration?

Given $\mu \in \mathbb{M}\alpha$ and $\xi \in \mathbb{M}(\alpha \times \beta)$, we say that a kernel $\nu \in \alpha \to \mathbb{M}\beta$ is a *disintegration* of ξ with respect to μ iff

$$\xi = \lambda^\star c.\, \mu \star \lambda t.\, \nu t \star \lambda y.\, c(t,y). \tag{17}$$

Often, as in the few examples below, μ is the Lebesgue measure

$$\Lambda = \lambda^\star c.\, \int_{-\infty}^{\infty} c(t)\,dt. \tag{18}$$

For example [as discussed on 2015-04-15], suppose α is $\mathbb{R}$, μ is the Lebesgue measure Λ, and ξ is the measure

$$\xi = \lambda^\star c.\, \int_4^7 \frac{c(\sin y, y)}{3}\,dy, \tag{19}$$

which by the way is the denotation of the program

$$\textbf{do}\ \{y \leftsquigarrow \textbf{uniform}\ 4\ 7;\ \textbf{return}\ (\textbf{sin}\ y, y)\}. \tag{20}$$

We let $t = \sin y$ and change the integration variable from y to t. We have $|dt/dy| = |\cos y| = \sqrt{1-t^2}$. Solving for y in terms of t gives the countably infinite number of solutions

$$y = 2\times\pi\times n + \arcsin t \quad \text{or} \quad y = 2\times\pi\times n + \pi - \arcsin t \tag{21}$$

where $n \in \mathbb{Z}$. Hence

$$\xi = \lambda^\star c.\, \sum_{n\in\mathbb{Z}} \int_{-1}^{1} \sum_{\substack{y\in\{2\times\pi\times n+\arcsin t,\\ 2\times\pi\times n+\pi-\arcsin t\},\\ 4<y<7}} \frac{c(t,y)}{3\times\sqrt{1-t^2}}\,dt \tag{22}$$

so by Tonelli's theorem

$$\xi = \lambda^\star c.\, \int_{-1}^{1} \sum_{n\in\mathbb{Z}} \sum_{\substack{y\in\{2\times\pi\times n+\arcsin t,\\ 2\times\pi\times n+\pi-\arcsin t\},\\ 4<y<7}} \frac{c(t,y)}{3\times\sqrt{1-t^2}}\,dt. \tag{23}$$

Comparing this against (17) shows that the kernel

$$\nu = \lambda t.\, \lambda^\star c'.\, \text{if } -1 < t < 1 \text{ then } \sum_{n\in\mathbb{Z}} \sum_{\substack{y\in\{2\times\pi\times n+\arcsin t,\\ 2\times\pi\times n+\pi-\arcsin t\},\\ 4<y<7}} \frac{c'(y)}{3\times\sqrt{1-t^2}} \text{ else } 0 \tag{24}$$

is a disintegration of ξ (with respect to Λ).

To take another example [as discussed on 2015-04-14], suppose α is $\mathbb{R}$, μ is the Lebesgue measure Λ, and ξ is some measure of the form

$$\xi = \lambda^\star c.\, \iint w(x,y) \times c(h_1(x,y) + h_2(x), x, y)\,dy\,dx \in \mathbb{M}(\mathbb{R}\times\beta_1\times\beta_2) \tag{25}$$

in which the integrals over x and y are some measures on β_1 and β_2, so $\beta = \beta_1 \times \beta_2$. In the expression $h_1(x,y) + h_2(x)$, note that the second term does not depend on y. Suppose now that we have found a family of kernels $\nu_x \in \mathbb{R} \to \mathbb{M}\beta_2$ ranging over $x \in \beta_1$, such that for each x, the kernel ν_x is a disintegration of

$$\xi_x = \lambda^\star c_x. \int w(x,y) \times c_x(h_1(x,y),y)\, dy \in \mathbb{M}(\mathbb{R} \times \beta_2) \tag{26}$$

with respect to Λ. In other words, suppose now we have

$$\int w(x,y) \times c_x(h_1(x,y),y)\, dy = \int_{-\infty}^{\infty} \nu_x t_1 \star \lambda y.\, c_x(t_1,y)\, dt_1 \tag{27}$$

for each $x \in \beta_1$ and $c_x \in (\mathbb{R} \times \beta_2) \to [0,\infty]$. Then for any $x \in \beta_1$ and $c \in (\mathbb{R} \times \beta_1 \times \beta_2) \to [0,\infty]$, we can let

$$c_x = \lambda(t_1,y).\, c(t_1 + h_2(x), x, y), \tag{28}$$

so by (27) we have

$$\int w(x,y) \times c(h_1(x,y) + h_2(x), x, y)\, dy = \int_{-\infty}^{\infty} \nu_x t_1 \star \lambda y.\, c(t_1 + h_2(x), x, y)\, dt_1. \tag{29}$$

Applying this equation in (25) gives

$$\xi \star c = \int \int_{-\infty}^{\infty} \nu_x t_1 \star \lambda y.\, c(t_1 + h_2(x), x, y)\, dt_1\, dx. \tag{30}$$

We let $t = t_1 + h_2(x)$ and change the inner integration variable from t_1 to t:

$$\xi \star c = \int \int_{-\infty}^{\infty} \nu_x(t - h_2(x)) \star \lambda y.\, c(t,x,y)\, dt\, dx. \tag{31}$$

If we could apply Tonelli's theorem (in particular, if we knew the measure on β_1 to be σ-finite), then we would have

$$\xi \star c = \int_{-\infty}^{\infty} \int \nu_x(t - h_2(x)) \star \lambda y.\, c(t,x,y)\, dx\, dt \tag{32}$$

and so would be able to set

$$\nu = \lambda t.\, \lambda^\star c'. \int \nu_x(t - h_2(x)) \star \lambda y.\, c'(x,y)\, dx. \tag{33}$$

3. Syntax and semantics

Figure 4 defines the syntax and type system of our language.

If $\Gamma \vdash e : \beta$, where Γ is the type environment $x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n$, then the denotation $[\![e]\!]$ of e is a measurable function [TODO: Emphasize measurability. What about σ-finiteness?] from the measurable space $\prod\Gamma = \alpha_1 \times \cdots \times \alpha_n$ to the measurable space β. Figure 5 specifies this denotation $[\![e]\!]$ by induction on e. We treat each element of $\prod\Gamma$ as a function that maps each variable name x_i to an element of the corresponding space α_i. [TODO: Explain syntax and semantics in tandem?]

(If $\Gamma \vdash h : \Delta$, where Δ is the sequence of types of variables bound by the heap h, then the denotation $[\![h]\!]$ of h is a measurable function from the measurable space $\prod\Gamma$ to the measurable space $\prod\Delta$. Again we treat each element of these product spaces as a function from variable names.) Extend do-notation to a binary operation that turns a heap and an expression into an expression:

$$\mathbf{do}\ \{[]; e\} = e \tag{34}$$

$$\mathbf{do}\ \{h; g; e\} = \mathbf{do}\ \{h; \mathbf{do}\ \{g; e\}\} \tag{35}$$

4. Partial evaluation

Our binding-time analysis is online: the metalanguage is static/earlier-stage evaluation (*Lazy s repr* in the Haskell code); the object language is dynamic/later-stage evaluation (*repr* in the Haskell code).

By convention, we write the names of term constructors in bold. For example, **fst** and **inl** are term constructors, so **fst** (x,y) and x are two different terms in our language. In contrast, we write the names of metalanguage functions in italic. For example, the following equations define the metalanguage function *fst*, which projects a head normal form of type $\alpha \times \beta$ to a term of type α:

$$\begin{aligned} \mathit{fst}\ (e_1, e_2) &= e_1 \\ \mathit{fst}\ a &= \mathbf{fst}\ a \end{aligned}$$

(The metavariable a stands for an atomic term.) Informally, we write the signature

$$\mathit{fst}\colon \lfloor \Gamma; \Delta \vdash \alpha \times \beta \rfloor \to \lceil \Gamma, \Delta \vdash \alpha \rceil$$

in which $\lfloor \Gamma; \Delta \vdash \alpha \times \beta \rfloor$ means head normal forms of type $\alpha \times \beta$ in type environment Γ, Δ (but only variables in Γ are considered atomic because the variables in Δ are bound in the heap), and $\lceil \Gamma, \Delta \vdash \alpha \rceil$ means terms of type α in type environment Γ, Δ. Hence *fst* (x,y) equals the term x. Analogously, we define the metalanguage function *snd*:

$$\begin{aligned} &\mathit{snd}\colon \lfloor \Gamma; \Delta \vdash \alpha \times \beta \rfloor \to \lceil \Gamma, \Delta \vdash \beta \rceil \\ &\mathit{snd}\ (e_1, e_2) = e_2 \\ &\mathit{snd}\ a \quad = \mathbf{snd}\ a \end{aligned}$$

The metalanguage functions *fst* and *snd* are thus partially evaluating counterparts to the term constructors **fst** and **snd**.

Following this convention, the term constructors that we write in bold include arithmetic operations. For example, **exp** and **+** and **−** are term constructors, so **exp** 0 and 3 **+** (**−**2) and 1 are two different terms in our language. In contrast, when we want to exponentiate the number 0 or add the number 3 to the negation of the number 2 in the metalanguage, we write non-bold exp 0 or $3 + (-2)$, which equals the number 1. Informally, we write the signatures

$$\begin{aligned} + &: \mathbb{R} \to \mathbb{R} \to \mathbb{R}, \\ - &: \mathbb{R} \to \mathbb{R}, \\ \exp &: \mathbb{R} \to \mathbb{R}. \end{aligned}$$

Moreover, we extend these metalanguage functions from operating on concrete numbers to operating on head normal forms (which include concrete numbers). That is, we define

$$\begin{aligned} + &: \lfloor \Gamma; \Delta \vdash \mathbb{R} \rfloor \to \lfloor \Gamma; \Delta \vdash \mathbb{R} \rfloor \to \lfloor \Gamma; \Delta \vdash \mathbb{R} \rfloor, \\ - &: \lfloor \Gamma; \Delta \vdash \mathbb{R} \rfloor \to \lfloor \Gamma; \Delta \vdash \mathbb{R} \rfloor, \\ \exp &: \lfloor \Gamma; \Delta \vdash \mathbb{R} \rfloor \to \lfloor \Gamma; \Delta \vdash \mathbb{R} \rfloor \end{aligned}$$

by extending the usual operations with the fallback cases

$$\begin{aligned} n_1 + n_2 &= n_1 \mathbf{+} n_2 && \text{if } n_1 \text{ or } n_2 \text{ is atomic,} \\ -a &= \mathbf{-}a && \text{given } a \text{ is atomic,} \\ \exp a &= \mathbf{exp}\ a && \text{given } a \text{ is atomic.} \end{aligned}$$

(It is straightforward to add algebraic simplifications such as $n + 0 = n$.) Similarly, we define the metalanguage function

$$<\colon \lfloor \Gamma; \Delta \vdash \mathbb{R} \rfloor \to \lfloor \Gamma; \Delta \vdash \mathbb{R} \rfloor \to \lfloor \Gamma; \Delta \vdash \mathbb{1} + \mathbb{1} \rfloor$$

by extending the usual comparison on concrete numbers

$$\begin{aligned} r_1 < r_2 &= \mathbf{inl}\ () && \text{if } r_1 \text{ is less than } r_2, \\ r_1 < r_2 &= \mathbf{inr}\ () && \text{if } r_1 \text{ is greater than or equal to } r_2 \end{aligned}$$

(where $r_1, r_2 \in \mathbb{R}$) with the fallback case

$$n_1 < n_2 = n_1 \mathbf{<} n_2 \qquad \text{if } n_1 \text{ or } n_2 \text{ is atomic.}$$

For pattern matching on sum types, our language includes the projection constructs **do** $\{$**let inl** $x = e; e'\}$ and **do** $\{$**let inr** $x = e; e'\}$.

Types $\alpha,\beta,\gamma ::= \mathbb{R} \mid \mathbb{1} \mid \alpha\times\beta \mid \alpha+\beta \mid \mathbb{M}\,\alpha$ Terms e

Type environments $\Gamma,\Delta ::= [] \mid \Gamma, x{:}\alpha$ Variables x, y

Type judgments $\Gamma \vdash e : \alpha$ Real numbers $r \in \mathbb{R}$

$$\frac{}{\Gamma, x{:}\alpha, \Delta \vdash x : \alpha} \quad \frac{r \in \mathbb{R}}{\Gamma \vdash r : \mathbb{R}} \quad \frac{\Gamma \vdash e : \mathbb{R}}{\Gamma \vdash -e : \mathbb{R}} \quad \frac{\Gamma \vdash e : \mathbb{R}}{\Gamma \vdash e^{-1} : \mathbb{R}} \quad \frac{\Gamma \vdash e : \mathbb{R}}{\Gamma \vdash \mathbf{exp}\ e : \mathbb{R}} \quad \frac{\Gamma \vdash e : \mathbb{R}}{\Gamma \vdash \mathbf{log}\ e : \mathbb{R}} \quad \frac{\Gamma \vdash e_1 : \mathbb{R} \quad \Gamma \vdash e_2 : \mathbb{R}}{\Gamma \vdash e_1 + e_2 : \mathbb{R}} \quad \frac{\Gamma \vdash e_1 : \mathbb{R} \quad \Gamma \vdash e_2 : \mathbb{R}}{\Gamma \vdash e_1 \times e_2 : \mathbb{R}}$$

$$\frac{\Gamma \vdash e_1 : \mathbb{R} \quad \Gamma \vdash e_2 : \mathbb{R}}{\Gamma \vdash e_1 < e_2 : \mathbb{1}+\mathbb{1}} \quad \frac{}{\Gamma \vdash () : \mathbb{1}} \quad \frac{\Gamma \vdash e_1 : \alpha \quad \Gamma \vdash e_2 : \beta}{\Gamma \vdash (e_1, e_2) : \alpha\times\beta} \quad \frac{\Gamma \vdash e : \alpha\times\beta}{\Gamma \vdash \mathbf{fst}\ e : \alpha} \quad \frac{\Gamma \vdash e : \alpha\times\beta}{\Gamma \vdash \mathbf{snd}\ e : \beta} \quad \frac{\Gamma \vdash e : \alpha}{\Gamma \vdash \mathbf{inl}\ e : \alpha+\beta} \quad \frac{\Gamma \vdash e : \beta}{\Gamma \vdash \mathbf{inr}\ e : \alpha+\beta}$$

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

$$\frac{\Gamma \vdash e : \mathbb{M}\,\alpha \quad \Gamma, x{:}\alpha \vdash e' : \mathbb{M}\,\beta}{\Gamma \vdash \mathbf{do}\ \{x \leftsquigarrow e; e'\} : \mathbb{M}\,\beta} \quad \frac{\Gamma \vdash e : \mathbb{R} \quad \Gamma \vdash e' : \mathbb{M}\,\alpha}{\Gamma \vdash \mathbf{do}\ \{\mathbf{factor}\ e; e'\} : \mathbb{M}\,\alpha} \quad \frac{}{\Gamma \vdash \mathbf{mzero} : \mathbb{M}\,\alpha} \quad \frac{\Gamma \vdash e_1 : \mathbb{M}\,\alpha \quad \Gamma \vdash e_2 : \mathbb{M}\,\alpha}{\Gamma \vdash \mathbf{mplus}\ e_1\ e_2 : \mathbb{M}\,\alpha}$$

Bindings (guards) $g ::= \mathbf{let\ inl}\ x = e \mid \mathbf{let\ inr}\ x = e \mid x \leftsquigarrow e \mid \mathbf{factor}\ e$

Heaps $h ::= [] \mid h; g$

Head normal forms $n, m ::= a \mid r \mid () \mid (e_1, e_2) \mid \mathbf{inl}\ e \mid \mathbf{inr}\ e \mid \mathbf{lebesgue} \mid \mathbf{return}\ e \mid \mathbf{do}\ \{g; e\} \mid \mathbf{mzero} \mid \mathbf{mplus}\ e_1\ e_2$

Atomic terms $a ::= x$ (not bound in the heap) $\mid -a \mid a^{-1} \mid \mathbf{exp}\ a \mid \mathbf{log}\ a \mid a+n \mid n+a \mid a\times n \mid n\times a \mid a<n \mid n<a \mid \mathbf{fst}\ a \mid \mathbf{snd}\ a$

Figure 4. The syntax and type system of our language

In these constructs, the body expression e' must be of measure type. Given $e : \alpha+\beta$, these constructs test whether the result of e is an **inl** value or an **inr** value. If e is an **inl** value, say, then **do** $\{\mathbf{let\ inl}\ x = e; e'\}$ binds $x : \alpha$ in e', whereas **do** $\{\mathbf{let\ inr}\ x = e; e'\}$ simply fails. In measure-theoretic terms, this construct converts a measure on α into a measure on $\alpha+\beta$, via the inclusion map **inl**.

To project from sum types in our partial evaluator, we define a pair of metalanguage functions *outl* and *outr*. Conceptually, *outl* projects a head normal form of type $\alpha+\beta$ to a term of type α. But because it is not known whether an atomic term is **inl**, we need to define *outl* in continuation-passing style [4, 6, 12]:

$$\begin{aligned}
&\mathit{outl}\colon \lfloor\Gamma;\Delta\vdash\alpha+\beta\rfloor \to (\lceil\Gamma,\Delta\vdash\alpha\rceil \to \langle\Gamma\vdash\Delta\rangle \to \lfloor\Gamma;\vdash\mathbb{M}\,\gamma\rfloor) \\
&\qquad\qquad \to \langle\Gamma\vdash\Delta\rangle \to \lfloor\Gamma;\vdash\mathbb{M}\,\gamma\rfloor \\
&\mathit{outl}\ a \qquad c\ h = \mathbf{do}\ \{\mathbf{let\ inl}\ x = a; c\ x\ h\} \\
&\mathit{outl}\ (\mathbf{inl}\ e)\ \ c\ h = c\ e\ h \\
&\mathit{outl}\ (\mathbf{inr}\ e)\ c\ h = \mathbf{mzero}
\end{aligned}$$

In the type above, $\langle\Gamma\vdash\Delta\rangle$ means heaps that bind the variables in Δ using the variables in Γ. The definition of

$$\begin{aligned}
&\mathit{outr}\colon \lfloor\Gamma;\Delta\vdash\alpha+\beta\rfloor \to (\lceil\Gamma,\Delta\vdash\beta\rceil \to \langle\Gamma\vdash\Delta\rangle \to \lfloor\Gamma;\vdash\mathbb{M}\,\gamma\rfloor) \\
&\qquad\qquad \to \langle\Gamma\vdash\Delta\rangle \to \lfloor\Gamma;\vdash\mathbb{M}\,\gamma\rfloor
\end{aligned}$$

is analogous.

Prove by mutual induction: a head normal form of type $\mathbb{R}$ does not use any variable bound in the heap; an atomic term of any type does not use any variable bound in the heap. So those terms can be strengthened (i.e., have variables bound in the heap removed from their type environments: $\lfloor\Gamma;\Delta\vdash\mathbb{R}\rfloor \to \lfloor\Gamma;\vdash\mathbb{R}\rfloor$).

Weakening is admissible (i.e., any term can have variables added to its type environment: $\forall\Gamma'\geq\Gamma.\,\forall\Delta'\geq\Delta.\,(\lfloor\Gamma;\Delta\vdash\alpha\rfloor \to \lfloor\Gamma';\Delta'\vdash\alpha\rfloor, \lceil\Gamma,\Delta\vdash\alpha\rceil \to \lceil\Gamma',\Delta'\vdash\alpha\rceil)$.

$$\begin{aligned}
&\mathit{abs}\colon \lfloor\Gamma;\Delta\vdash\mathbb{R}\rfloor \to (\forall\Gamma'\geq\Gamma.\,\lfloor\Gamma';\Delta\vdash\mathbb{R}\rfloor \to \langle\Gamma'\vdash\Delta\rangle \to \lfloor\Gamma';\vdash\mathbb{M}\,\gamma\rfloor) \\
&\qquad\qquad \to \langle\Gamma\vdash\Delta\rangle \to \lfloor\Gamma;\vdash\mathbb{M}\,\gamma\rfloor \\
&\mathit{abs}\ r\ c\ h = c\ |r|\ h \qquad \text{given } r\in\mathbb{R} \\
&\mathit{abs}\ a\ c\ h = \mathbf{do}\ \{x \leftsquigarrow \mathbf{mplus}\ (\mathbf{do}\ \{\mathbf{let\ inl}\ _ = a<0; \mathbf{return}\ (-a)\}) \\
&\qquad\qquad (\mathbf{do}\ \{\mathbf{let\ inr}\ _ = a<0; \mathbf{return}\ a\}); \\
&\qquad\qquad c\ x\ h\} \qquad \text{given } a \text{ is atomic}
\end{aligned}$$

Call-by-need PE/supercompilation [3, 11, 14] though our side effect is commutative yet non-idempotent.

To preserve sharing in lazy evaluation, use a heap; for partial evaluation, the heap leaves some locations unbound [8].

Example—uniform distribution on $[0,1]$:

$$\begin{aligned}
\mathbf{random} = \mathbf{do}\ \{&x \leftsquigarrow \mathbf{lebesgue}; \\
&\mathbf{let\ inl}\ _ = 0<x; \\
&\mathbf{let\ inl}\ _ = x<1; \\
&\mathbf{return}\ x\}
\end{aligned} \tag{36}$$

For disintegration to succeed on this simple example, it is essential that our constructs for discriminating **inl** from **inr** do not force evaluation of the scrutinee! Follow-up example: sum of uniform distributions on $[0,1]$ and $[2,3]$, to show how we handle **mplus**, which is where one invocation of partial evaluation turns into two.

Acknowledgments

Thanks to Jacques Carette, Ryan Culpepper, Praveen Narayanan, Norman Ramsey, Mitchell Wand, and Robert Zinkov for helpful comments and discussions.

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

$$
\begin{aligned}
[\![x]\!]\rho &= \rho(x) \\
[\![r]\!]\rho &= r \\
[\![-e]\!]\rho &= -[\![e]\!]\rho \\
[\![e^{-1}]\!]\rho &= ([\![e]\!]\rho)^{-1} \\
[\![\mathbf{exp}\ e]\!]\rho &= \exp([\![e]\!]\rho) \\
[\![\mathbf{log}\ e]\!]\rho &= \log([\![e]\!]\rho) \\
[\![e_1 + e_2]\!]\rho &= [\![e_1]\!]\rho + [\![e_2]\!]\rho \\
[\![e_1 \times e_2]\!]\rho &= [\![e_1]\!]\rho \times [\![e_2]\!]\rho \\
[\![e_1 < e_2]\!]\rho &= \mathbf{true} && \text{if } [\![e_1]\!]\rho < [\![e_2]\!]\rho \\
[\![e_1 < e_2]\!]\rho &= \mathbf{false} && \text{if } [\![e_1]\!]\rho \geq [\![e_2]\!]\rho \\
[\![()]\!]\rho &= () \\
[\![(e_1, e_2)]\!]\rho &= ([\![e_1]\!]\rho, [\![e_2]\!]\rho) \\
[\![\mathbf{fst}\ e]\!]\rho &= x && \text{if } [\![e]\!]\rho = (x, y) \\
[\![\mathbf{snd}\ e]\!]\rho &= y && \text{if } [\![e]\!]\rho = (x, y) \\
[\![\mathbf{inl}\ e]\!]\rho &= \mathbf{inl}\ ([\![e]\!]\rho) \\
[\![\mathbf{inr}\ e]\!]\rho &= \mathbf{inr}\ ([\![e]\!]\rho) \\
[\![\mathbf{lebesgue}]\!]\rho &= \lambda^\star c.\, \textstyle\int_{-\infty}^{\infty} c(x)\, dx \\
[\![\mathbf{return}\ e]\!]\rho &= \lambda^\star c.\, c([\![e]\!]\rho) \\
[\![\mathbf{do}\ \{\mathbf{let\ inl}\ x = e; e'\}]\!]\rho &= [\![e']\!](\rho\{x \mapsto a\}) && \text{if } [\![e]\!]\rho = \mathbf{inl}\ a \\
[\![\mathbf{do}\ \{\mathbf{let\ inl}\ x = e; e'\}]\!]\rho &= \lambda^\star c.\, 0 && \text{if } [\![e]\!]\rho = \mathbf{inr}\ b \\
[\![\mathbf{do}\ \{\mathbf{let\ inr}\ x = e; e'\}]\!]\rho &= [\![e']\!](\rho\{x \mapsto b\}) && \text{if } [\![e]\!]\rho = \mathbf{inr}\ b \\
[\![\mathbf{do}\ \{\mathbf{let\ inr}\ x = e; e'\}]\!]\rho &= \lambda^\star c.\, 0 && \text{if } [\![e]\!]\rho = \mathbf{inl}\ a \\
[\![\mathbf{do}\ \{x \leftsquigarrow e; e'\}]\!]\rho &= \lambda^\star c.\, [\![e]\!]\rho \star \lambda a.\, [\![e']\!](\rho\{x \mapsto a\}) \star c \\
[\![\mathbf{do}\ \{\mathbf{factor}\ e; e'\}]\!]\rho &= \lambda^\star c.\, ([\![e]\!]\rho) \times ([\![e']\!]\rho \star c) \\
[\![\mathbf{mzero}]\!]\rho &= \lambda^\star c.\, 0 \\
[\![\mathbf{mplus}\ e_1\ e_2]\!]\rho &= \lambda^\star c.\, ([\![e_1]\!]\rho \star c) + ([\![e_2]\!]\rho \star c)
\end{aligned}
$$

Figure 5. The denotational semantics of our language

References

[1] S. Bhat, A. Agarwal, R. Vuduc, and A. Gray. A type theory for probability density functions. In *POPL '12: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 545–556, New York, Jan. 2012. ACM Press.

[2] S. Bhat, J. Borgström, A. D. Gordon, and C. V. Russo. Deriving probability density functions from probabilistic functional programs. In N. Piterman and S. A. Smolka, editors, *Proceedings of TACAS 2013: 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 7795 in Lecture Notes in Computer Science, pages 508–522, Berlin, 16–24 Mar. 2013. Springer. ISBN 978-3-642-36741-0.

[3] M. Bolingbroke and S. Peyton Jones. Supercompilation by evaluation. In J. Gibbons, editor, *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell*, pages 135–146, New York, 2010. ACM Press. ISBN 978-1-4503-0252-4.

[4] A. Bondorf. Improving binding times without explicit CPS-conversion. In W. D. Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, volume V(1) of *Lisp Pointers*, pages 1–10, New York, 22–24 June 1992. ACM Press.

[5] J. T. Chang and D. Pollard. Conditioning as disintegration. *Statistica Neerlandica*, 51(3):287–317, Nov. 1997. URL `http://www.stat.yale.edu/ jtc5/papers/ConditioningAsDisintegration.pdf`.

[6] O. Danvy, K. Malmkjær, and J. Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 18(6):730–751, 1996.

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

[8] S. Fischer, J. Silva, S. Tamarit, and G. Vidal. Preserving sharing in the partial evaluation of lazy functional programs. In A. King, editor, *Revised Selected Papers from LOPSTR 2007: 17th International Symposium on Logic-Based Program Synthesis and Transformation*, number 4915 in Lecture Notes in Computer Science, pages 74–89, Berlin, 2008. Springer. ISBN 978-3-540-78768-6.

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

[10] A. H. Jazwinski. *Stochastic Processes and Filtering Theory*. Academic Press, San Diego, CA, 1970.

[11] J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *POPL '92: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 258–268, New York, 1992. ACM Press. ISBN 0-89791-453-8.

[12] J. L. Lawall and O. Danvy. Continuation-based partial evaluation. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 227–238, New York, 1994. ACM Press. URL `http://www.diku.dk/users/julia/lawall-danvy-lfp94.ps.gz`.

[13] D. J. C. MacKay. Introduction to Monte Carlo methods. In M. I. Jordan, editor, *Learning and Inference in Graphical Models*. Kluwer, Dordrecht, 1998. URL `http://www.cs.toronto.edu/ mackay/abstracts/erice.html` `http://www.inference.phy.cam.ac.uk/mackay/BayesMC.html`. Paperback: *Learning in Graphical Models*, MIT Press.

[14] N. Mitchell. Rethinking supercompilation. In P. Hudak and S. Weirich, editors, *ICFP '10: Proceedings of the ACM International Conference on Functional Programming*, pages 309–320, New York, 2010. ACM Press. ISBN 978-1-60558-794-3.

[15] S. Park, F. Pfenning, and S. Thrun. A probabilistic language based upon sampling functions. `http://www.postech.ac.kr/ gla/`, Aug. 2006. URL `http://www.postech.ac.kr/ gla/paper/toplas06.ps` `http://www.postech.ac.kr/ gla/paper/toplas06.pdf`. Submitted.

[16] D. Pollard. *A User's Guide to Measure Theoretic Probability*. Cambridge University Press, Cambridge, 2001. URL `http://www.stat.yale.edu/ pollard/Books/UGMTP/`.

[17] A. Stuhlmüller and N. D. Goodman. Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs. *Cognitive Systems Research*, 28:80–99, 2014.

For all $h : \langle \Gamma \vdash \Delta \rangle$, $m : \lfloor \Gamma; \Delta \vdash \mathbb{M}\,\alpha \rfloor$, $e' : \lceil \Gamma, \Delta, x{:}\alpha \vdash \mathbb{M}\,\gamma \rceil$, $[\![\mathit{fwdExec}\ m\ (\lambda n.\, \lambda h'.\, \mathbf{do}\ \{h'; e'\{x \mapsto n\}\})\ h]\!] = [\![\mathbf{do}\ \{h; x \leftsquigarrow m; e'\}]\!]$.

For all $h : \langle \Gamma \vdash \Delta \rangle$, $e : \lceil \Gamma, \Delta \vdash \alpha \rceil$, $e' : \lceil \Gamma, \Delta, x{:}\alpha \vdash \mathbb{M}\,\gamma \rceil$, $[\![\mathit{fwdEval}\ e\ (\lambda n.\, \lambda h'.\, \mathbf{do}\ \{h'; e'\{x \mapsto n\}\})\ h]\!] = [\![\mathbf{do}\ \{h; e'\{x \mapsto e\}\}]\!]$.

For all $h : \langle \Gamma \vdash \Delta \rangle$, $m : \lfloor \Gamma; \Delta \vdash \mathbb{M}\,\mathbb{R} \rfloor$, $e' : \lceil \Gamma, \Delta, x{:}\mathbb{R} \vdash \mathbb{M}\,\gamma \rceil$, $[\![\mathbf{do}\ \{x \leftsquigarrow \mathbf{lebesgue}; \mathit{bwdExec}\ m\ x\ (\lambda h'.\, \mathbf{do}\ \{h'; e'\})\ h\}]\!] = [\![\mathbf{do}\ \{h; x \leftsquigarrow m; e'\}]\!]$

and for all $n : \lfloor \Gamma; \Delta \vdash \mathbb{R} \rfloor$, $[\![(\mathit{bwdExec}\ m\ x\ (\lambda h'.\, \mathbf{do}\ \{h'; e'\})\ h)\{x \mapsto n\}]\!] = [\![\mathit{bwdExec}\ m\ n\ (\lambda h'.\, \mathbf{do}\ \{h'; e'\})\ h]\!]$.

For all $h : \langle \Gamma \vdash \Delta \rangle$, $e : \lceil \Gamma, \Delta \vdash \mathbb{R} \rceil$, $e' : \lceil \Gamma, \Delta, x{:}\mathbb{R} \vdash \mathbb{M}\,\gamma \rceil$, $[\![\mathbf{do}\ \{x \leftsquigarrow \mathbf{lebesgue}; \mathit{bwdEval}\ e\ x\ (\lambda h'.\, \mathbf{do}\ \{h'; e'\})\ h\}]\!] = [\![\mathbf{do}\ \{h; e'\{x \mapsto e\}\}]\!]$

and for all $n : \lfloor \Gamma; \Delta \vdash \mathbb{R} \rfloor$, $[\![(\mathit{bwdEval}\ e\ x\ (\lambda h'.\, \mathbf{do}\ \{h'; e'\})\ h)\{x \mapsto n\}]\!] = [\![\mathit{bwdEval}\ e\ n\ (\lambda h'.\, \mathbf{do}\ \{h'; e'\})\ h]\!]$.

Figure 6. The specification of our lazy partial evaluator and disintegrator

Lazy partial evaluation to head normal form

$$
\begin{array}{lll}
\mathit{fwdExec}\colon \lfloor \Gamma;\Delta \vdash \mathbb{M}\,\alpha \rfloor \to (\forall \Gamma' \geq \Gamma.\, \forall \Delta' \geq \Delta.\, \lfloor \Gamma';\Delta' \vdash \alpha \rfloor \to \langle \Gamma' \vdash \Delta' \rangle \to \lfloor \Gamma';\vdash \mathbb{M}\,\gamma \rfloor) \to \langle \Gamma \vdash \Delta \rangle \to \lfloor \Gamma;\vdash \mathbb{M}\,\gamma \rfloor & & \\
\mathit{fwdExec}\ a & c\ h & = \mathbf{do}\ \{x \leftsquigarrow a;\quad c\ x\ h\} \qquad \text{given } a \text{ is atomic} \\
\mathit{fwdExec}\ \mathbf{lebesgue} & c\ h & = \mathbf{do}\ \{x \leftsquigarrow \mathbf{lebesgue};\, c\ x\ h\} \\
\mathit{fwdExec}\ (\mathbf{return}\ e) & c\ h & = \mathit{fwdEval}\ e\ c\ h \\
\mathit{fwdExec}\ (\mathbf{do}\ \{g;e\}) & c\ h & = \mathit{fwdEval}\ e\ (\lambda m.\, \mathit{fwdExec}\ m\ c)\ (h;g) \\
\mathit{fwdExec}\ \mathbf{mzero} & c\ h & = \mathbf{mzero} \\
\mathit{fwdExec}\ (\mathbf{mplus}\ e_1\ e_2) & c\ h & = \mathbf{mplus}\ (\mathit{fwdEval}\ e_1\ (\lambda m.\, \mathit{fwdExec}\ m\ c)\ h)\ (\mathit{fwdEval}\ e_2\ (\lambda m.\, \mathit{fwdExec}\ m\ c)\ h) \\
\\
\mathit{fwdEval}\colon \lceil \Gamma,\Delta \vdash \alpha \rceil \to (\forall \Gamma' \geq \Gamma.\, \forall \Delta' \geq \Delta.\, \lfloor \Gamma';\Delta' \vdash \alpha \rfloor \to \langle \Gamma' \vdash \Delta' \rangle \to \lfloor \Gamma';\vdash \mathbb{M}\,\gamma \rfloor) \to \langle \Gamma \vdash \Delta \rangle \to \lfloor \Gamma;\vdash \mathbb{M}\,\gamma \rfloor & & \\
\mathit{fwdEval}\ n & c\ h & = c\ n\ h \qquad \text{given } n \text{ is in head normal form} \\
\mathit{fwdEval}\ (\mathbf{fst}\ e_0) & c\ h & = \mathit{fwdEval}\ e_0\ (\lambda n_0.\, \mathit{fwdEval}\ (\mathit{fst}\ n_0)\ c)\ h \qquad \text{unless } e_0 \text{ is atomic} \\
\mathit{fwdEval}\ (\mathbf{snd}\ e_0) & c\ h & = \mathit{fwdEval}\ e_0\ (\lambda n_0.\, \mathit{fwdEval}\ (\mathit{snd}\ n_0)\ c)\ h \qquad \text{unless } e_0 \text{ is atomic} \\
\mathit{fwdEval}\ (-e_0) & c\ h & = \mathit{fwdEval}\ e_0\ (\lambda n_0.\, c\ (-n_0))\ h \\
\mathit{fwdEval}\ (e_0^{-\mathbf{1}}) & c\ h & = \mathit{fwdEval}\ e_0\ (\lambda n_0.\, c\ (n_0^{-1}))\ h \\
\mathit{fwdEval}\ (\mathbf{exp}\ e_0) & c\ h & = \mathit{fwdEval}\ e_0\ (\lambda n_0.\, c\ (\exp n_0))\ h \\
\mathit{fwdEval}\ (\mathbf{log}\ e_0) & c\ h & = \mathit{fwdEval}\ e_0\ (\lambda n_0.\, c\ (\log n_0))\ h \\
\mathit{fwdEval}\ (e_1 + e_2) & c\ h & = \mathit{fwdEval}\ e_1\ (\lambda n_1.\, \mathit{fwdEval}\ e_2\ (\lambda n_2.\, c\ (n_1 + n_2)))\ h \\
\mathit{fwdEval}\ (e_1 \times e_2) & c\ h & = \mathit{fwdEval}\ e_1\ (\lambda n_1.\, \mathit{fwdEval}\ e_2\ (\lambda n_2.\, c\ (n_1 \times n_2)))\ h \\
\mathit{fwdEval}\ (e_1 < e_2) & c\ h & = \mathit{fwdEval}\ e_1\ (\lambda n_1.\, \mathit{fwdEval}\ e_2\ (\lambda n_2.\, c\ (n_1 < n_2)))\ h \\
\mathit{fwdEval}\ x & c\ (h_1; x \leftsquigarrow e; h_2) & = \mathit{fwdEval}\ e \qquad (\lambda m.\, \mathit{fwdExec}\ m\ (\lambda n.\, \lambda h_1'.\, c\ n\ (h_1'; x \leftsquigarrow \mathbf{return}\ n; h_2)))\ \ h_1 \\
\mathit{fwdEval}\ x & c\ (h_1; \mathbf{let\ inl}\ x = e_0; h_2) & = \mathit{fwdEval}\ e_0\ (\lambda n_0.\, \mathit{outl}\ n_0\ \ (\lambda e.\, \mathit{fwdEval}\ e\ (\lambda n.\, \lambda h_1'.\, c\ n\ (h_1'; x \leftsquigarrow \mathbf{return}\ n; h_2))))\ h_1 \\
\mathit{fwdEval}\ x & c\ (h_1; \mathbf{let\ inr}\ x = e_0; h_2) & = \mathit{fwdEval}\ e_0\ (\lambda n_0.\, \mathit{outr}\ n_0\ (\lambda e.\, \mathit{fwdEval}\ e\ (\lambda n.\, \lambda h_1'.\, c\ n\ (h_1'; x \leftsquigarrow \mathbf{return}\ n; h_2))))\ h_1
\end{array}
$$

Disintegration from head normal form

$$
\begin{array}{lll}
\mathit{bwdExec}\colon \lfloor \Gamma;\Delta \vdash \mathbb{M}\,\mathbb{R} \rfloor \to \lfloor \Gamma;\Delta \vdash \mathbb{R} \rfloor \to (\forall \Gamma' \geq \Gamma.\, \forall \Delta' \geq \Delta.\, \langle \Gamma' \vdash \Delta' \rangle \to \lfloor \Gamma';\vdash \mathbb{M}\,\gamma \rfloor) \to \langle \Gamma \vdash \Delta \rangle \to \lfloor \Gamma;\vdash \mathbb{M}\,\gamma \rfloor & & \\
\mathit{bwdExec}\ \mathbf{lebesgue} & n\ c\ h & = c\ h \\
\mathit{bwdExec}\ (\mathbf{return}\ e) & n\ c\ h & = \mathit{bwdEval}\ e\ n\ c\ h \\
\mathit{bwdExec}\ (\mathbf{do}\ \{g;e\}) & n\ c\ h & = \mathit{fwdEval}\ e\ (\lambda m.\, \mathit{bwdExec}\ m\ n\ c)\ (h;g) \\
\mathit{bwdExec}\ \mathbf{mzero} & n\ c\ h & = \mathbf{mzero} \\
\mathit{bwdExec}\ (\mathbf{mplus}\ e_1\ e_2) & n\ c\ h & = \mathbf{mplus}\ (\mathit{fwdEval}\ e_1\ (\lambda m.\, \mathit{bwdExec}\ m\ n\ c)\ h)\ (\mathit{fwdEval}\ e_2\ (\lambda m.\, \mathit{bwdExec}\ m\ n\ c)\ h) \\
\\
\mathit{bwdEval}\colon \lceil \Gamma,\Delta \vdash \mathbb{R} \rceil \to \lfloor \Gamma;\Delta \vdash \mathbb{R} \rfloor \to (\forall \Gamma' \geq \Gamma.\, \forall \Delta' \geq \Delta.\, \langle \Gamma' \vdash \Delta' \rangle \to \lfloor \Gamma';\vdash \mathbb{M}\,\gamma \rfloor) \to \langle \Gamma \vdash \Delta \rangle \to \lfloor \Gamma;\vdash \mathbb{M}\,\gamma \rfloor & & \\
\mathit{bwdEval}\ (\mathbf{fst}\ e_0) & n\ c\ h & = \mathit{fwdEval}\ e_0\ (\lambda n_0.\, \mathit{bwdEval}\ (\mathit{fst}\ n_0)\ n\ c)\ h \qquad \text{unless } e_0 \text{ is atomic} \\
\mathit{bwdEval}\ (\mathbf{snd}\ e_0) & n\ c\ h & = \mathit{fwdEval}\ e_0\ (\lambda n_0.\, \mathit{bwdEval}\ (\mathit{snd}\ n_0)\ n\ c)\ h \qquad \text{unless } e_0 \text{ is atomic} \\
\mathit{bwdEval}\ (-e_0) & n\ c\ h & = \mathit{bwdEval}\ e_0\ (-n)\ c\ h \\
\mathit{bwdEval}\ (e_0^{-\mathbf{1}}) & n\ c\ h & = \mathbf{do}\ \{\mathbf{factor}\ (n \times n)^{-1};\, \mathit{bwdEval}\ e_0\ n^{-1}\ c\ h\} \\
\mathit{bwdEval}\ (\mathbf{exp}\ e_0) & n\ c\ h & = \mathit{outl}\ (0 < n)\ (\lambda\_.\, \lambda h'.\, \mathbf{do}\ \{\mathbf{factor}\ n^{-1};\, \mathit{bwdEval}\ e_0\ (\log n)\ c\ h'\})\ h \\
\mathit{bwdEval}\ (\mathbf{log}\ e_0) & n\ c\ h & = \mathbf{do}\ \{\mathbf{factor}\ (\exp n);\, \mathit{bwdEval}\ e_0\ (\exp n)\ c\ h\} \\
\mathit{bwdEval}\ (e_1 + e_2) & n\ c\ h & = \mathit{fwdEval}\ e_1\ (\lambda n_1.\, \mathit{bwdEval}\ e_2\ (n + (-n_1))\ c)\ h \\
 & & \sqcup\, \mathit{fwdEval}\ e_2\ (\lambda n_2.\, \mathit{bwdEval}\ e_1\ (n + (-n_2))\ c)\ h \\
\mathit{bwdEval}\ (e_1 \times e_2) & n\ c\ h & = \mathit{fwdEval}\ e_1\ (\lambda n_1.\, \mathit{abs}\ n_1\ (\lambda n_1'.\, \lambda h'.\, \mathbf{do}\ \{\mathbf{factor}\ n_1'^{-1};\, \mathit{bwdEval}\ e_2\ (n \times n_1^{-1})\ c\ h'\}))\ h \\
 & & \sqcup\, \mathit{fwdEval}\ e_2\ (\lambda n_2.\, \mathit{abs}\ n_2\ (\lambda n_2'.\, \lambda h'.\, \mathbf{do}\ \{\mathbf{factor}\ n_2'^{-1};\, \mathit{bwdEval}\ e_1\ (n \times n_2^{-1})\ c\ h'\}))\ h \\
\mathit{bwdEval}\ x & n\ c\ (h_1; x \leftsquigarrow e; h_2) & = \mathit{fwdEval}\ e \qquad (\lambda m.\, \mathit{bwdExec}\ m\ n\ (\lambda h_1'.\, c\ (h_1'; x \leftsquigarrow \mathbf{return}\ n; h_2)))\ \ h_1 \\
\mathit{bwdEval}\ x & n\ c\ (h_1; \mathbf{let\ inl}\ x = e_0; h_2) & = \mathit{fwdEval}\ e_0\ (\lambda n_0.\, \mathit{outl}\ n_0\ \ (\lambda e.\, \mathit{bwdEval}\ e\ n\ (\lambda h_1'.\, c\ (h_1'; x \leftsquigarrow \mathbf{return}\ n; h_2))))\ h_1 \\
\mathit{bwdEval}\ x & n\ c\ (h_1; \mathbf{let\ inr}\ x = e_0; h_2) & = \mathit{fwdEval}\ e_0\ (\lambda n_0.\, \mathit{outr}\ n_0\ (\lambda e.\, \mathit{bwdEval}\ e\ n\ (\lambda h_1'.\, c\ (h_1'; x \leftsquigarrow \mathbf{return}\ n; h_2))))\ h_1
\end{array}
$$

[TODO: uncurry heap argument?]

Figure 7. The implementation of our lazy partial evaluator and disintegrator