Commutative Semantics for Probabilistic Programming

Sam Staton(✉)

University of Oxford, Oxford, UK
sam.staton@cs.ox.ac.uk

Abstract. We show that a measure-based denotational semantics for probabilistic programming is commutative.

The idea underlying probabilistic programming languages (Anglican, Church, Hakaru, etc.) is that programs express statistical models as a combination of prior distributions and likelihood of observations. The product of prior and likelihood is an unnormalized posterior distribution, and the inference problem is to find the normalizing constant. One common semantic perspective is thus that a probabilistic program is understood as an unnormalized posterior measure, in the sense of measure theory, and the normalizing constant is the measure of the entire semantic domain.

A programming language is said to be commutative if only data flow is meaningful; control flow is irrelevant, and expressions can be re-ordered. It has been unclear whether probabilistic programs are commutative because it is well-known that Fubini-Tonelli theorems for reordering integration fail in general. We show that probabilistic programs are in fact commutative, by characterizing the measures/kernels that arise from programs as 's-finite', i.e. sums of finite measures/kernels.

The result is of theoretical interest, but also of practical interest, because program transformations based on commutativity help with symbolic inference and can improve the efficiency of simulation.

1 Introduction

The key idea of probabilistic programming is that programs describe statistical models. Programming language theory can give us tools to build and analyze the models. Recall Bayes' law: the posterior probability is proportional to the product of the likelihood of observed data and the prior probability.

$$\text{Posterior} \;\propto\; \text{Likelihood} \times \text{Prior} \tag{1}$$

One way to understand a probabilistic program is that it describes the measure that is the product of the likelihood and the prior. This product is typically not a probability measure, it does not sum to one. The inference problem is to find the normalizing constant so that we can find (or approximate) the posterior probability measure.

A probabilistic programming language is an ML-like programming language with three special constructs, corresponding to the three terms in Bayes' law:

H. Yang (Ed.): ESOP 2017, LNCS 10201, pp. 855–879, 2017.
DOI: 10.1007/978-3-662-54434-1_32

- *sample*, which draws from a prior distribution, which may be discrete (like a Bernoulli distribution) or continuous (like a Gaussian distribution);
- *score*, or *observe*, which records the likelihood of a particular observed data point, sometimes called 'soft conditioning';
- *normalize*, which finds the normalization constant and the posterior probability distribution.

The implementation of normalize typically involves simulation. One hope is that we can use program transformations to improve the efficiency of this simulation, or even to symbolically calculate the normalizing constant. We turn to some transformations of this kind in Sect. 4.1. But a very first program transformation is to reorder the lines of a program, as long as the data dependencies are preserved, e.g.

$$\boxed{\begin{array}{l}\mathsf{let}\ x = t\ \mathsf{in}\\ \mathsf{let}\ y = u\ \mathsf{in}\\ v\end{array}} = \boxed{\begin{array}{l}\mathsf{let}\ y = u\ \mathsf{in}\\ \mathsf{let}\ x = t\ \mathsf{in}\\ v\end{array}} \tag{2}$$

where x not free in u, y not free in t. This is known as *commutativity*. For example, in a traditional programming language with memory, this transformation is valid provided t and u reference different locations. In probabilistic programming, a fundamental intuition is that programs are stateless. From a practical perspective, it is essential to be able to reorder lines and so access more sophisticated program transformations (e.g. Sect. 4.1); reordering lines can also affect the efficiency of simulation. The main contribution of this paper is the result:

Theorem 4 (Sect. 4.2). *The commutativity Eq.* (2) *is always valid in probabilistic programs.*

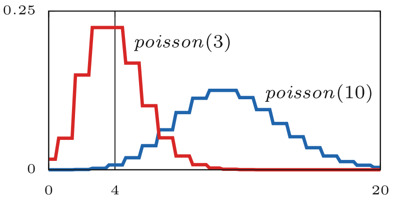
1.1 A First Introduction to Probabilistic Programming

To illustrate the key ideas of probabilistic programming, consider the following simple problem, which we explain in English and then specify as a probabilistic program.

1. A telephone operator has forgotten what day it is.
2. He receives on average ten calls per hour in the week and three calls per hour at the weekend.
3. He observes four calls in a given hour.
4. What is the probability that it is a week day?

We describe this as a probabilistic program as follows:

```
normalize(
    let $x$ = sample($bern(\frac{5}{7})$) in
    let $r$ = if $x$ then 10 else 3 in
    observe 4 from $poisson(r)$;
    return($x$))
```



Lines 2–5 describe the combination of the likelihood and the prior. First, on line 2, we sample from the prior: the chance that it is a week day is $\frac{5}{7}$. On line 3, we set the rate of calls, depending on whether it is a week day. On line 4 we record the observation that six calls were received when the rate was r, using the Poisson distribution. For a discrete distribution, the likelihood is the probability of the observation point, which for the Poisson distribution with rate r is $r^4e^{-r}/4!$.

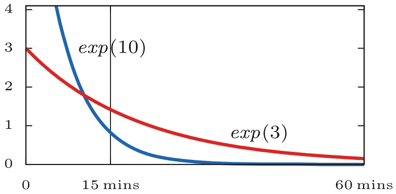
We thus find a semantics for lines 2–5, an unnormalized posterior measure on $\{\mathsf{true}, \mathsf{false}\}$, by considering the only two paths through the program, depending on the outcome of the Bernoulli trial.

- The Bernoulli trial (line 2) produces true with prior probability $\frac{5}{7}$ (it is a week day), and then the rate is 10 (line 3) and so the likelihood of the data is $10^4e^{-10}/4! \approx 0.019$ (line 4). So the unnormalized posterior probability of true is $\frac{5}{7} \times 0.019 \approx 0.014$ (prior×likelihood).
- The Bernoulli trial produces false with prior probability $\frac{2}{7}$ (it is the weekend), and then the likelihood of the observed data is $3^4e^{-3}/4! \approx 0.168$; so the unnormalized posterior measure of false is $\frac{2}{7} \times 0.168 \approx 0.048$.

The measure $(\mathsf{true} \mapsto 0.014, \mathsf{false} \mapsto 0.048)$ is not a probability measure because it doesn't sum to 1. To build a probability measure we divide by $0.014 + 0.048 = 0.062$, to get a posterior probability measure $(\mathsf{true} \mapsto 0.22, \mathsf{false} \mapsto 0.78)$. The normalizing constant, 0.062, is sometimes called model evidence; it is an indication of how well the data fits the model.

Next we consider a slightly different problem. Rather than observing four calls in a given hour, suppose the telephone operator merely observes that the time between two given calls is 15 min. We describe this as a probabilistic program as follows:

```
1. normalize(
2.     let $x$ = sample($bern(\frac{5}{7})$) in
3.     let $r$ = if $x$ then 10 else 3 in
4.     observe ($\frac{15}{60}$) from $exp(r)$;
5.     return($x$))
```



The difference here is that the observation is from the exponential distribution ($exp(r)$), which is a continuous distribution, In Bayesian statistics, the likelihood of a continuous distribution is taken to be the value of the probability density function at the observation point. The density function of the exponential distribution $exp(r)$ with rate r is $(x \mapsto re^{-rx})$. So if the decay rate is 10, the likelihood of time $\frac{15}{60}$ is $10e^{-2.5} \approx 0.82$, and if the decay rate is 3, the likelihood is $3e^{-0.75} \approx 1.42$. We thus find that the unnormalized posterior measure of true is $\frac{5}{7} \times 0.82 \approx 0.586$ (prior×likelihood), and the unnormalized posterior measure of false is $\frac{2}{7} \times 1.42 \approx 0.405$. In this example, the model evidence is $0.586 + 0.405 \approx 0.991$. We divide by this to find the normalized posterior, which is $(\mathsf{true} \mapsto 0.592, \mathsf{false} \mapsto 0.408)$.

In these simple examples, there are only two paths through the program. In general the prior may be a continuous distribution over an uncountable set, such as the uniform distribution on an interval, in which case a simulation can only find an approximate normalizing constant. Suppose that the telephone operator does not know what time it is, but knows a function $f : [0, 24) \to (0, \infty)$ mapping each time of day to the average call rate. Then by solving the following problem, he can ascertain a posterior probability distribution for the current time.

$$\mathsf{normalize}\big(\mathsf{let}\, t = \mathsf{sample}(\mathit{uniform}([0, 24)))\, \mathsf{in}\, \mathsf{observe}\, (\tfrac{15}{60})\, \mathsf{from}\, \mathit{exp}(f(t)); \mathsf{return}(t)\big). \tag{3}$$

Although simulation might only be approximate, we can give a precise semantics to the language using measure theory. In brief,

- programs of type $\mathbb{A}$ are interpreted as measures on $\mathbb{A}$, and more generally expressions of type $\mathbb{A}$ with free variables in Γ are measure kernels $\Gamma \rightsquigarrow \mathbb{A}$;
- sampling from a prior describes a probability measure;
- observations are interpreted by multiplying the measure of a path by the likelihood of the data;
- sequencing is Lebesgue integration: $\mathsf{let}\, x = t\, \mathsf{in}\, u \approx \int t(\mathrm{d}x)\, u$;
- normalization finds the measure of the whole space, the normalizing constant.

To put it another way, the programming language is a language for building measures. For full details, see Sect. 3.2.

1.2 Commutativity and Infinite Measures

If, informally, sequencing is integration, then commutativity laws such as (2) amount to changing the order of integration, e.g.

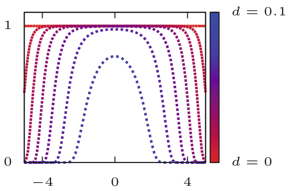
$$\int t(\mathrm{d}x) \int u(\mathrm{d}y)\, v \;=\; \int u(\mathrm{d}y) \int t(\mathrm{d}x)\, v \tag{4}$$

A first non-trivial fact of measure theory is Fubini's theorem: for finite measures, Eq. (4) holds. However, commutativity theorems like this do not hold for arbitrary infinite measures. In fact, if we deal with arbitrary infinite measures, we do not even know whether sequencing $\int t(\mathrm{d}x)\, v$ is a genuine measure kernel. As we will show, for the measures that are definable in our language, sequencing *is* well defined, and commutativity *does* hold. But let us first emphasize that infinite measures appear to be unavoidable because

- there is no known useful syntactic restriction that enforces finite measures;
- a program with finite measure may have a subexpression with infinite measure, and this can be useful.

To illustrate these points, consider the following program, a variation on (3).

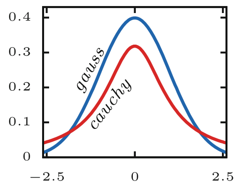
$$\mathsf{let}\ x = \mathsf{sample}(\mathit{gauss}(0, 1))\ \mathsf{in}\ \mathsf{observe}\ d\ \mathsf{from}\ \mathit{exp}(1/f(x));\ \mathsf{return}(x) \quad : \mathbb{R} \tag{5}$$



Here $gauss(0, 1)$ is the standard Gaussian distribution with mean 0 and standard deviation 1; recall that its density f is $f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$. The illustration on the right shows the unnormalized posterior for (5) as the observed data goes from $d = 0.1$ (blue dotted line) to $d = 0$ (red straight line). Notice that at $d = 0$, the resulting unnormalized posterior measure on $\mathbb{R}$ is the flat Lebesgue measure on $\mathbb{R}$, which assigns to each interval (m, n) its size, $(n - m)$. The Lebesgue measure of the entire real line, the would-be normalizing constant, is ∞, so we cannot find a posterior probability measure. A statistician would probably not be very bothered about this, because a tiny change in the observed data yields a finite normalizing constant. But that is not good enough for a semanticist, who must give a meaning to *every* program.

It is difficult to see how a simple syntactic restriction could eliminate program (5) while keeping other useful programs such as (3). Another similar program is

$$\mathsf{let}\, x = \mathsf{sample}(gauss(0, 1))\, \mathsf{in}\, \mathsf{score}(g(x)/f(x)); \mathsf{return}(x) \quad : \mathbb{R} \tag{6}$$



where $g(x) = \frac{1}{\pi(1+x^2)}$ is the density function of the standard Cauchy distribution and $\mathsf{score}(r)$ is shorthand for $(\mathsf{observe}\, 0\, \mathsf{from}\, exp(r))$—recall that the density of the exponential distribution $exp(r)$ at 0 is $r = re^{-r \times 0}$. Program (6) is the importance sampling algorithm for simulating a Cauchy distribution from a Gaussian. To see why this algorithm is correct, i.e. $(6) = \mathsf{sample}(cauchy(0, 1))$, it is helpful to rewrite it:

$$\underline{\mathsf{let}\ x = \mathsf{sample}(gauss(0, 1))\ \mathsf{in}\ \mathsf{score}(1/f(x))}\ ;\ \mathsf{score}(g(x))\ ;\ \mathsf{return}(x) \quad : \mathbb{R}.$$

Notice that the underlined subexpression is the Lebesgue measure, as in (5), and recall that sequencing is integration. So program (6) is correct because it is integrating the density g over the Lebesgue measure; this is equal to the Cauchy probability measure, by definition of density.

1.3 Commutativity Through s-Finite Kernels

It is known that commutativity holds not just for finite measures but also for s-finite measures, which are formed from a countable sum of finite measures. The key contribution of this paper is that all closed probabilistic programs define s-finite measures. To show this compositionally, we must also give a semantics to open programs, which we interpret using a notion of s-finite kernel (Definition 2), which is a countable sum of finite, bounded kernels; these support sequential composition (Lemma 3). Iterated integrals and interchange (4) are no problem for s-finite measures (Proposition 5). We conclude (Theorem 4) that the commutativity Eq. (2) is always valid in probabilistic programs.

Moreover, s-finite kernels are exactly what is needed, because:

Theorem 6 (Sect. 5.1). *The following are equivalent:*

- *a probabilistic program expression of type $\mathbb{A}$ and free variables in Γ;*
- *an s-finite kernel $\Gamma \rightsquigarrow \mathbb{A}$.*

(The probabilistic programming language here is an idealized one that includes countable sum types, all measurable functions, and all probability distributions.)

Summary of Contribution. We use s-finite kernels to provide the first semantic model (Sect. 3.2) of a probabilistic programming language that

- interprets programs such as those in Sect. 1.1;
- supports basic program transformations such as commutativity (Theorem 4);
- justifies program transformations based on statistical ideas such as conjugate priors, importance sampling and resampling, in a compositional way (Sect. 4.1).

In Sect. 6 we relate our contributions with earlier attempts at this problem.

2 Preliminaries

2.1 Measures and Kernels

Measure theory generalizes the ideas of size and probability distribution from countable discrete sets to uncountable sets. To motivate, recall that if we sample a real number from a standard Gaussian distribution then it is impossible that we should sample the precise value 0, even though that is the expected value. We resolve this apparent paradox by recording the probability that the sample drawn lies within an interval, or more generally, a measurable set. For example, a sample drawn from a standard Gaussian distribution will lie in the interval $(-1, 1)$ with probability 0.68. We now recall some rudiments of measure theory; see e.g. [32] for a full introduction.

A *σ-algebra* on a set X is a collection of subsets of X that contains $\emptyset$ and is closed under complements and countable unions. A *measurable space* is a pair (X, Σ_X) of a set X and a σ-algebra Σ_X on it. The sets in Σ_X are called *measurable sets.*

For example, the Borel sets are the smallest σ-algebra on $\mathbb{R}$ that contains the intervals. We will always consider $\mathbb{R}$ with this σ-algebra. Similarly the Borel sets on $[0, \infty]$ are the smallest σ-algebra containing the intervals. For any countable set (e.g. $\mathbb{N}$, $\{0, 1\}$) we will consider the discrete σ-algebra, where all sets are measurable.

A *measure* on a measurable space (X, Σ_X) is a function $\mu : \Sigma_X \to [0, \infty]$ into the set $[0, \infty]$ of extended non-negative reals that takes countable disjoint unions to sums, i.e. $\mu(\emptyset) = 0$ and $\mu(\biguplus_{n\in\mathbb{N}} U_n) = \sum_{n\in\mathbb{N}} \mu(U_n)$ for any $\mathbb{N}$-indexed sequence of disjoint measurable sets U. A *probability measure* is a measure μ such that $\mu(X) = 1$.

For example, the Lebesgue measure λ on $\mathbb{R}$ is generated by $\lambda(a, b) = b - a$. For any $x \in X$, the Dirac measure δ_x has $\delta_x(U) = [x \in U]$. (Here and elsewhere we regard a property, e.g. $[x \in U]$, as its characteristic function $X \to \{0, 1\}$.) To give a measure on a countable discrete measurable space X it is sufficient to assign an element of $[0, \infty]$ to each element of X. For example, the counting measure γ is determined by $\gamma(\{x\}) = 1$ for all $x \in X$.

A function $f : X \to Y$ between measurable spaces is *measurable* if $f^{-1}(U) \in \Sigma_X$ for all $U \in \Sigma_Y$. This ensures that we can form a *pushforward* measure $f_*\mu$ on Y out of any measure μ on X, with $(f_*\mu)(U) = \mu(f^{-1}(U))$.

For example, the arithmetic operations on $\mathbb{R}$ are all measurable. If $U \in \Sigma_X$ then the characteristic function $[- \in U] : X \to \{0, 1\}$ is measurable.

We can integrate a measurable function $f : X \to [0, \infty]$ over a measure μ on X to get number $\int_X \mu(\mathrm{d}x)\, f(x) \in [0, \infty]$. (Some authors use different notation, e.g. $\int f\, \mathrm{d}\mu$.) Integration satisfies the following properties (e.g. [32, Theorem 12]): $\int_X \mu(\mathrm{d}x)\, [x \in U] = \mu(U)$, $\int_X \mu(\mathrm{d}x)\, r f(x) = r \int_X \mu(\mathrm{d}x)\, f(x)$, $\int_X \mu(\mathrm{d}x)\, 0 = 0$, $\int_X \mu(\mathrm{d}x)\, (f(x) + g(x)) = (\int_X \mu(\mathrm{d}x)\, f(x)) + (\int_X \mu(\mathrm{d}x)\, g(x))$, and

$$\lim_i \textstyle\int_X \mu(\mathrm{d}x)\, f_i(x) \;=\; \int_X \mu(\mathrm{d}x)\, (\lim_i f_i(x)) \tag{7}$$

for any monotone sequence $f_1 \leq f_2 \leq \ldots$ of measurable functions $f : X \to [0, \infty]$. These properties entirely determine integration, since every measurable function is a limit of a monotone sequence of simple functions [32, Lemma 11]. It follows that countable sums commute with integration:

$$\int_X \mu(\mathrm{d}x) \Big(\sum_{i \in \mathbb{N}} f_i(x) \Big) \;=\; \sum_{i \in \mathbb{N}} \int_X \mu(\mathrm{d}x)\, f_i(x). \tag{8}$$

For example, integration over the Lebesgue measure on $\mathbb{R}$ is Lebesgue integration, generalizing the idea of the area under a curve. Integration with respect to the counting measure on a countable discrete space is just summation, e.g. $\int_{\mathbb{N}} \gamma(\mathrm{d}i)\, f(i) = \sum_{i \in \mathbb{N}} f(i)$.

We can use integration to build new measures. If μ is a measure on X and $f : X \to [0, \infty]$ is measurable then we define a measure μ_f on X by putting $\mu_f(U) \overset{\text{def}}{=} \int_U \mu(\mathrm{d}x)\, f(x)$. We say f is the *density function* for μ_f. For example, the function $x \mapsto \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$ is the density function for the standard Gaussian probability measure on $\mathbb{R}$ with respect to the Lebesgue measure.

A *kernel* k from X to Y is a function $k : X \times \Sigma_Y \to [0, \infty]$ such that each $k(x, -) : \Sigma_Y \to [0, \infty]$ is a measure and each $k(-, U) : X \to [0, \infty]$ is measurable. Because each $k(x, -)$ is a measure, we can integrate any measurable function $f : Y \to [0, \infty]$ to get $\int_Y k(x, \mathrm{d}y)\, f(y) \in [0, \infty]$. We write $k : X \rightsquigarrow Y$ if k is a kernel. We say that k is a *probability kernel* if $k(x, Y) = 1$ for all $x \in X$.

2.2 s-Finite Measures and Kernels

We begin with a lemma about sums of kernels.

Proposition 1. *Let X, Y be measurable spaces. If $k_1 \dots k_n \dots : X \rightsquigarrow Y$ are kernels then the function $(\sum_{i=1}^{\infty} k_i) : X \times \Sigma_Y \to [0,\infty]$ given by*

$$(\textstyle\sum_{i=1}^{\infty} k_i)(x,U) \stackrel{\text{def}}{=} \displaystyle\sum_{i=1}^{\infty}(k_i(x,U))$$

is a kernel $X \rightsquigarrow Y$. Moreover, for any measurable function $f : Y \to [0,\infty]$,

$$\int_Y (\textstyle\sum_{i=1}^{\infty} k_i)(x,\mathrm{d}y)\, f(y) \;=\; \displaystyle\sum_{i=1}^{\infty} \int_Y k_i(x,\mathrm{d}y)\, f(y).$$

Proof. That $\sum_{i\in\mathbb{N}} k_i : X \times \Sigma_Y \to [0,\infty]$ is a kernel is quite straightforward: it is measurable in X because a countable sum of measurable functions is measurable (e.g. [32, Sect. 2.2]); it is a measure in Y because countable positive sums commute:

$$\textstyle\sum_{i=1}^{\infty}(k_i(x, \biguplus_{j=1}^{\infty} U_j)) = \sum_{i=1}^{\infty}(\sum_{j=1}^{\infty} k_i(x,U_j)) = \sum_{j=1}^{\infty}(\sum_{i=1}^{\infty} k_i(x,U_j))$$

The second part of the proposition follows once we understand that every measurable function $f : Y \to [0,\infty]$ is a limit of simple functions and apply the monotone convergence theorem (7).

Definition 2. Let X, Y be measurable spaces. A kernel $k : X \rightsquigarrow Y$ is *finite* if there is finite $r \in [0,\infty)$ such that, for all x, $k(x,Y) < r$.

A kernel $k : X \rightsquigarrow Y$ is *s-finite* if there is a sequence $k_1 \dots k_n \dots$ of finite kernels and $\sum_{i=1}^{\infty} k_i = k$.

Note that the bound in the finiteness condition, and the choice of sequence in the s-finiteness condition, are uniform, across all arguments to the kernel.

The definition of s-finite kernel also appears in recent work by Kallenberg [20] and Last and Penrose [23, Appendix A]. The idea of s-finite measures is perhaps more established ([9, Lemma 8.6], [39, Sect. A.0]).

3 Semantics of a Probabilistic Programming Language

We give a typed first order probabilistic programming language in Sect. 3.1, and its semantics in Sect. 3.2. The semantics is new: we interpret programs as s-finite kernels. The idea of interpreting programs as kernels is old (e.g. [21]), but the novelty here is that we can treat infinite measures. It is not a priori obvious that a compositional denotational semantics based on kernels makes sense for infinite measures; the trick is to use s-finite kernels as an invariant, via Lemma 3.

3.1 A Typed First Order Probabilistic Programming Language

Our language syntax is not novel: it is the same language as in [43], and as such an idealized, typed, first order version of Anglican [46], Church [11], Hakaru [30], Venture [26] and so on.

Types. The language has types

$$\mathbb{A}, \mathbb{B} ::= \ \mathbb{R} \mid \mathsf{P}(\mathbb{A}) \mid 1 \mid \mathbb{A} \times \mathbb{B} \mid \ \textstyle\sum_{i \in I} \mathbb{A}_i$$

where I ranges over countable, non-empty sets. Alongside the usual sum and product types, we have a special type $\mathbb{R}$ of real numbers and types $\mathsf{P}(\mathbb{A})$ of probability distributions. For example, $(1 + 1)$ is a type of booleans, and $\mathsf{P}(1 + 1)$ is a type of distributions over booleans, and $\sum_{i \in \mathbb{N}} 1$ is a type of natural numbers. This is not a genuine programming language because we include countably infinite sums rather than recursion schemes; this is primarily because countably infinite disjoint unions play such a crucial role in classical measure theory, and constructive measure theory is an orthogonal issue (but see e.g. [1]).

Types $\mathbb{A}$ are interpreted as measurable spaces $[\![\mathbb{A}]\!]$.

- $[\![\mathbb{R}]\!]$ is the measurable space of reals, with its Borel sets.
- $[\![\mathsf{P}(\mathbb{A})]\!]$ is the set $P([\![\mathbb{A}]\!])$ of probability measures on $[\![\mathbb{A}]\!]$ together with the σ-algebra generated by the sets $\{\mu \mid \mu(U) < r\}$ for each $U \in \Sigma_X$ and $r \in [0, 1]$ (the 'Giry monad' [10]).
- $[\![1]\!]$ is the discrete measurable space with one point.
- $[\![\mathbb{A} \times \mathbb{B}]\!]$ is the product space $[\![\mathbb{A}]\!] \times [\![\mathbb{B}]\!]$. The σ-algebra $\Sigma_{[\![\mathbb{A}\times\mathbb{B}]\!]}$ is generated by rectangles $(U \times V)$ with $U \in \Sigma_{[\![\mathbb{A}]\!]}$ and $V \in \Sigma_{[\![\mathbb{B}]\!]}$ (e.g. [32, Definition 16]).
- $[\![\sum_{i \in I} \mathbb{A}_i]\!]$ is the coproduct space $\biguplus_{i \in I} [\![\mathbb{A}_i]\!]$. The σ-algebra $\Sigma_{[\![\sum_{i \in I} \mathbb{A}_i]\!]}$ is generated by sets $\{(i, a) \mid a \in U\}$ for $U \in \Sigma_{[\![\mathbb{A}_i]\!]}$.

Terms. We distinguish typing judgements: $\Gamma \vdash_{\mathsf{d}} t\colon \mathbb{A}$ for deterministic terms, and $\Gamma \vdash_{\mathsf{p}} t\colon \mathbb{A}$ for probabilistic terms. Formally, a context $\Gamma = (x_1\colon \mathbb{A}_1, \ldots, x_n\colon \mathbb{A}_n)$ means a measurable space $[\![\Gamma]\!] \overset{\text{def}}{=} \prod_{i=1}^{n} [\![\mathbb{A}_i]\!]$. Deterministic terms $\Gamma \vdash_{\mathsf{d}} t\colon \mathbb{A}$ denote measurable functions from $[\![\Gamma]\!] \to [\![\mathbb{A}]\!]$, and probabilistic terms $\Gamma \vdash_{\mathsf{p}} t'\colon \mathbb{A}$ denote kernels $[\![\Gamma]\!] \rightsquigarrow [\![\mathbb{A}]\!]$. We give a syntax and type system here, and a semantics in Sect. 3.2.

Sums and Products. The language includes variables, and standard constructors and destructors for sum and product types.

$$\frac{}{\Gamma, x\colon \mathbb{A}, \Gamma' \vdash_{\mathsf{d}} x\colon \mathbb{A}} \qquad \frac{\Gamma \vdash_{\mathsf{d}} t\colon \mathbb{A}_i}{\Gamma \vdash_{\mathsf{d}} (i, t)\colon \sum_{i \in I} \mathbb{A}_i}$$

$$\frac{\Gamma \vdash_{\mathsf{d}} t\colon \sum_{i \in I} \mathbb{A}_i \quad (\Gamma, x\colon \mathbb{A}_i \vdash_{\mathsf{z}} u_i\colon \mathbb{B})_{i \in I}}{\Gamma \vdash_{\mathsf{z}} \mathsf{case}\ t\ \mathsf{of}\ \{(i, x) \Rightarrow u_i\}_{i \in I}\colon \mathbb{B}} \ (\mathsf{z} \in \{\mathsf{d}, \mathsf{p}\})$$

$$\frac{}{\Gamma \vdash_{\mathsf{d}} ()\colon 1} \qquad \frac{\Gamma \vdash_{\mathsf{d}} t_0\colon \mathbb{A}_0 \quad \Gamma \vdash_{\mathsf{d}} t_1\colon \mathbb{A}_1}{\Gamma \vdash_{\mathsf{d}} (t_0, t_1)\colon \mathbb{A}_0 \times \mathbb{A}_1} \qquad \frac{\Gamma \vdash_{\mathsf{d}} t\colon \mathbb{A}_0 \times \mathbb{A}_1}{\Gamma \vdash_{\mathsf{d}} \pi_j(t)\colon \mathbb{A}_j}$$

In the rules for sums, I may be infinite. In the last rule, j is 0 or 1. We use some standard syntactic sugar, such as false and true for the injections in the type $\mathsf{bool} = 1 + 1$, and if for case in that instance.

Sequencing. We include the standard constructs for sequencing (e.g. [25,29]).

$$\frac{\Gamma \vdash_{\mathsf{d}} t\colon \mathbb{A}}{\Gamma \vdash_{\mathsf{p}} \mathsf{return}(t)\colon \mathbb{A}} \qquad \frac{\Gamma \vdash_{\mathsf{p}} t\colon \mathbb{A} \quad \Gamma, x\colon \mathbb{A} \vdash_{\mathsf{p}} u\colon \mathbb{B}}{\Gamma \vdash_{\mathsf{p}} \mathsf{let}\ x = t\ \mathsf{in}\ u\colon \mathbb{B}}$$

Language-Specific Constructs. So far the language is very standard. We also include constant terms for all measurable functions.

$$\frac{\Gamma \vdash_{\mathsf{d}} t\colon \mathbb{A}}{\Gamma \vdash_{\mathsf{d}} f(t)\colon \mathbb{B}}\ (f\colon [\![\mathbb{A}]\!] \to [\![\mathbb{B}]\!] \text{ measurable}) \tag{9}$$

Thus the language contains all the arithmetic operations (e.g. $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$) and predicates (e.g. $(=) : \mathbb{R}\times\mathbb{R} \to \mathsf{bool}$). Moreover, all the families of probability measures are in the language. For example, the Gaussian distributions $\mathit{gauss} : \mathbb{R} \times \mathbb{R} \to P(\mathbb{R})$ are parameterized by mean and standard deviation, so that we have a judgement $\mu : \mathbb{R}, \sigma : \mathbb{R} \vdash_{\mathsf{d}} \mathit{gauss}(\mu, \sigma)\colon \mathsf{P}(\mathbb{R})$. (Some families are not defined for all parameters, e.g. the standard deviation should be positive, but we make ad-hoc safe choices throughout rather than using exceptions or subtyping.)

The core of the language is the constructs corresponding to the terms in Bayes' law (1): sampling from prior distributions, recording likelihood scores,

$$\frac{\Gamma \vdash_{\mathsf{d}} t\colon \mathsf{P}(\mathbb{A})}{\Gamma \vdash_{\mathsf{p}} \mathsf{sample}(t)\colon \mathbb{A}} \qquad \frac{\Gamma \vdash_{\mathsf{d}} t\colon \mathbb{R}}{\Gamma \vdash_{\mathsf{p}} \mathsf{score}(t)\colon 1}$$

and calculating the normalizing constant and a normalized posterior.

$$\frac{\Gamma \vdash_{\mathsf{p}} t\colon \mathbb{A}}{\Gamma \vdash_{\mathsf{d}} \mathsf{normalize}(t)\colon \mathbb{R} \times \mathsf{P}(\mathbb{A}) + 1 + 1}$$

Normalization will fail if the normalizing constant is zero or infinity. Notice that normalization produces a probability distribution; in a complex model this could then be used as a prior and sampled from. This is sometimes called a 'nested query'.

Note About Observations. Often a probability distribution d has a widely understood density function f with respect to some base measure. For example, the exponential distribution with rate r is usually defined in terms of the density function $x \mapsto re^{-rx}$ with respect to the Lebesgue measure on $\mathbb{R}$. The score construct is typically called with a density. In this circumstance, we use the informal notation $\mathsf{observe}\, t\, \mathsf{from}\, d$ for $\mathsf{score}(f(t))$. For example, $\mathsf{observe}\, t\, \mathsf{from}\, \mathit{exp}(r)$ is informal notation for $\mathsf{score}(re^{-r \times t})$. In a more realistic programming language, this informality is avoided by defining a 'distribution object' to be a pair of a probability measure and a density function for it. There is no difference in expressivity between an observe construction and a score construct. For example, $\mathsf{score}(r)$ can be understood as $\mathsf{observe}\, 0\, \mathsf{from}\, \mathit{exp}(r)$, since $re^{-r0} = r$.

(Technical point: although density functions can be understood as Radon-Nikodym derivatives, these are not uniquely determined on measure-zero sets, and so a distribution object does need to come with a given density function. Typically the density is continuous with respect to some metric so that the likelihood is not vulnerable to small inaccuracies in observations. See e.g. [43, Sect. 9] for more details.)

3.2 Denotational Semantics

Recall that types $\mathbb{A}$ are interpreted as measurable spaces $[\![\mathbb{A}]\!]$. We now explain how to interpret a deterministic term in context, $\Gamma \vdash_{\mathsf{d}} t\colon \mathbb{A}$ as a measurable function $[\![t]\!] : [\![\Gamma]\!] \to [\![\mathbb{A}]\!]$, and how to interpret a probabilistic term in context, $\Gamma \vdash_{\mathsf{p}} t\colon \mathbb{A}$, as an s-finite kernel $[\![t]\!] : [\![\Gamma]\!] \rightsquigarrow [\![\mathbb{A}]\!]$.

The semantics is given by induction on the structure of terms. Before we begin we need a lemma.

Lemma 3. *Let X, Y, Z be measurable spaces, and let $k : X \times Y \rightsquigarrow Z$ and $l : X \rightsquigarrow Y$ be s-finite kernels (Definition 2). Then we can define a s-finite kernel $(k \star l) : X \rightsquigarrow Z$ by*

$$(k \star l)(x, U) \stackrel{\text{def}}{=} \int_Y l(x, \mathrm{d}y)\, k(x, y, U)$$

so that

$$\int_Z (k \star l)(x, \mathrm{d}z)\, f(z) \;=\; \int_Y l(x, \mathrm{d}y) \int_Z k(x, y, \mathrm{d}z)\, f(z)$$

Proof. Suppose $k = \sum_{i=1}^{\infty} k_i$ and $l = \sum_{j=1}^{\infty} l_j$ are s-finite kernels, and that the k_i's and l_j's are finite kernels. We need to show that $k \star l$ is a kernel and moreover s-finite. We first show that each $k_i \star l_j$ is a finite kernel. Each $(k_i \star l_j)(x, -) : \Sigma_Z \to [0, \infty]$ is a measure:

$$\begin{aligned}
(k_i \star l_j)(x, \textstyle\biguplus_{a=1}^{\infty} U_a) &= \textstyle\int_Y l_j(x, \mathrm{d}y)\, k_i(x, y, \biguplus_{a=1}^{\infty} U_a) & \\
&= \textstyle\int_Y l_j(x, \mathrm{d}y)\, \sum_{a=1}^{\infty} k_i(x, y, U_a) & \quad k \text{ is a kernel} \\
&= \textstyle\sum_{a=1}^{\infty} \int_Y l_j(x, \mathrm{d}y)\, k_i(x, y, U_a) & \quad \text{Eq. (8)}
\end{aligned}$$

The measurability of each $(k_i \star l_j)(-, U) : X \to [0, \infty]$ follows from the general fact that for any measurable function $f : X \times Y \to [0, \infty]$, the function $\int_Y l_j(-, \mathrm{d}y)\, f(-, y) : X \to [0, \infty]$ is measurable (e.g. [32, Theorem 20(ii)]). Thus $(k_i \star l_j)$ is a kernel. This step crucially uses the fact that each measure $l_j(x, -)$ is finite.

To show that $(k_i \star l_j)$ is a *finite* kernel, we exhibit a bound. Since k_i and l_j are finite, we have $r, s \in (0, \infty)$ such that $k_i(x, y, Z) < r$ and $l_j(x, Y) < s$ for all x, y. Now rs is a bound on $(k_i \star l_j)$ since

$$(k_i \star l_j)(x, Z) = \textstyle\int_Y l_j(x, \mathrm{d}y)\, k(x, y, Z) < \int_Y l_j(x, \mathrm{d}y)\, r = r l_j(x, Y) < rs.$$

So each $(k_i \star l_j)$ is a finite kernel. Note that here we used the uniformity in the definition of finite kernel.

We conclude that $(k \star l)$ is an s-finite kernel by showing that it is a countable sum of finite kernels:

$$\begin{aligned}(k \star l)(x, U) &= ((\textstyle\sum_i k_i) \star (\sum_j l_j))(x, U) \\ &= \textstyle\int_Y \sum_j (l_j(x, \mathrm{d}y)) \sum_i (k_i(x, y, U)) \\ &= \textstyle\sum_i \int_Y \sum_j (l_j(x, \mathrm{d}y))\, k_i(x, y, U) && \text{Eq. (8)} \\ &= \textstyle\sum_i \sum_j \int_Y l_j(x, \mathrm{d}y)\, k_i(x, y, U) && \text{Proposition 1} \\ &= \textstyle\sum_i \sum_j (k_i \star l_j)(x, U)\end{aligned}$$

The final part of the statement follows by writing f as a limit of a sequence of simple functions and using the monotone convergence property (7).

Remark. It seems unlikely that we can drop the assumption of s-finiteness in Lemma 3. The difficulty is in showing that $(k \star l) : X \times \Sigma_Z \to [0, \infty]$ is measurable in its first argument without some extra assumption. (I do not have a counterexample, but then examples of non-measurable functions are hard to find.)

Semantics. We now explain the semantics of the language, beginning with variables, sums and products, which is essentially the same as a set-theoretic semantics.

$$\begin{aligned}&[\![x]\!]_{\gamma,d,\gamma'} \stackrel{\text{def}}{=} d \qquad [\![(i, t)]\!]_\gamma \stackrel{\text{def}}{=} (i, [\![t]\!]_\gamma) \\ &[\![\mathsf{case}\ t\ \mathsf{of}\ \{(i, x) \Rightarrow u_i\}_{i \in I}]\!]_\gamma \stackrel{\text{def}}{=} [\![u_i]\!]_{\gamma,d} \quad \text{if } [\![t]\!]_\gamma = (i, d) \\ &[\![()]\!]_\gamma \stackrel{\text{def}}{=} () \qquad [\![(t_0, t_1)]\!]_\gamma \stackrel{\text{def}}{=} ([\![t_0]\!]_\gamma, [\![t_1]\!]_\gamma) \qquad [\![\pi_j(t)]\!]_\gamma \stackrel{\text{def}}{=} d_i \quad \text{if } [\![t]\!]_\gamma = (d_0, d_1)\end{aligned}$$

Here we have only treated the case expressions when the continuation is deterministic; we return to the probabilistic case later.

The semantics of sequencing are perhaps the most interesting: return is the Dirac delta measure, and let is integration.

$$[\![\mathsf{return}(t)]\!]_{\gamma,U} \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } [\![t]\!]_\gamma \in U \\ 0 & \text{otherwise} \end{cases} \qquad [\![\mathsf{let}\, x = t\, \mathsf{in}\, u]\!]_{\gamma,U} \stackrel{\text{def}}{=} \int_{\mathbb{A}} [\![t]\!]_{\gamma,\mathrm{d}x}\ [\![u]\!]_{\gamma,x,U}$$

The interpretation $[\![\mathsf{return}(t)]\!]$ is finite, hence s-finite. The fact that $[\![\mathsf{let}\, x = t\, \mathsf{in}\, u]\!]$ is an s-finite kernel is Lemma 3: this is the most intricate part of the semantics.

We return to the case expression where the continuation is probabilistic:

$$[\![\mathsf{case}\ t\ \mathsf{of}\ \{(i, x) \Rightarrow u_i\}_{i \in I}]\!]_{\gamma,U} \stackrel{\text{def}}{=} [\![u_i]\!]_{\gamma,d,U} \quad \text{if } [\![t]\!]_\gamma = (i, d).$$

We must show that this is an s-finite kernel. Recall that $[\![u_i]\!] : [\![\Gamma \times \mathbb{A}_i]\!] \rightsquigarrow [\![\mathbb{B}]\!]$, s-finite. We can also form $\overline{[\![u_i]\!]} : [\![\Gamma]\!] \times \biguplus_j [\![\mathbb{A}_j]\!] \rightsquigarrow [\![\mathbb{B}]\!]$ with

$$\overline{[\![u_i]\!]}_{\gamma,(j,a),U} \stackrel{\text{def}}{=} \begin{cases} [\![u_i]\!]_{\gamma,a,U} & i = j \\ 0 & \text{otherwise} \end{cases}$$

and it is easy to show that $\overline{[\![u_i]\!]}$ is an s-finite kernel. Another easy fact is that a countable sum of s-finite kernels is again an s-finite kernel, so we can build an s-finite kernel $(\sum_i \overline{[\![u_i]\!]}) : [\![\varGamma]\!] \times \biguplus_j [\![\mathbb{A}_j]\!] \rightsquigarrow [\![\mathbb{B}]\!]$. Finally, we use a simple instance of Lemma 3 to compose $(\sum_i \overline{[\![u_i]\!]})$ with $[\![t]\!] : [\![\varGamma]\!] \to \biguplus_j [\![\mathbb{A}_j]\!]$ and conclude that $[\![\mathsf{case}\ t\ \mathsf{of}\ \{(i,x) \Rightarrow u_i\}_{i\in I}]\!]$ is an s-finite kernel.

The language specific constructions are straightforward.

$$[\![\mathsf{sample}(t)]\!]_{\gamma,U} \overset{\text{def}}{=} [\![t]\!]_\gamma(U) \qquad [\![\mathsf{score}(t)]\!]_{\gamma,U} \overset{\text{def}}{=} \begin{cases} |[\![t]\!]_\gamma| & \text{if } U = \{()\} \\ 0 & \text{if } U = \emptyset. \end{cases}$$

In the semantics of sample, we are merely using the fact that to give a measurable function $X \to P(Y)$ is to give a probability kernel $X \rightsquigarrow Y$. Probability kernels are finite, hence s-finite.

The semantics of score is a one point space whose measure is the argument. (We take the absolute value of $[\![t]\!]_\gamma$ because measures should be non-negative. An alternative would be to somehow enforce this in the type system.) We need to show that $[\![\mathsf{score}(t)]\!]$ is an s-finite kernel. Although $[\![\mathsf{score}(t)]\!]_{\gamma,1}$ is always finite, $[\![\mathsf{score}(t)]\!]$ is not necessarily a *finite kernel* because we cannot find a uniform bound. To show that it is *s-finite*, for each $i \in \mathbb{N}_0$, define a kernel $k_i : [\![\varGamma]\!] \rightsquigarrow 1$

$$k_i(\gamma, U) \overset{\text{def}}{=} \begin{cases} [\![\mathsf{score}(t)]\!]_{\gamma,U} & \text{if } [\![\mathsf{score}(t)]\!]_{\gamma,U} \in [i, i+1) \\ 0 & \text{otherwise} \end{cases}$$

So each k_i is a finite kernel, bounded by $(i+1)$, and $[\![\mathsf{score}(t)]\!] = \sum_{i=0}^{\infty} k_i$, so it is s-finite.

We give a semantics to normalization by finding the normalizing constant and dividing by it, as follows. Consider $\varGamma \vdash_{\mathsf{p}} t \colon \mathbb{A}$ and let $\text{evidence}_t \overset{\text{def}}{=} [\![t]\!]_{\gamma,[\![\mathbb{A}]\!]}$.

$$[\![\mathsf{normalize}(t)]\!]_\gamma \overset{\text{def}}{=} \begin{cases} (0, (\text{evidence}_t, \frac{[\![t]\!]_{\gamma,(-)}}{\text{evidence}_t})) & \text{evidence}_t \in (0,\infty) \\ (1, ()) & \text{evidence}_t = 0 \\ (2, ()) & \text{evidence}_t = \infty \end{cases}$$

(In practice, the normalization will only be approximate. We leave it for future work to develop semantic notions of approximation in this setting, e.g. [27].)

4 Properties and Examples

4.1 Examples of Statistical Reasoning

Lebesgue Measure, Densities and Importance Sampling. The Lebesgue measure on $\mathbb{R}$ is not a primitive in our language, because it is not a probability measure, but it is definable. For example, we can score the standard Gaussian by the inverse of its density function, $f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$.

$$
\begin{aligned}
&[\![\vdash_{\mathsf{p}} \mathsf{let}\, x = \mathsf{sample}(\mathit{gauss}(0,1))\, \mathsf{in}\, \mathsf{score}(\tfrac{1}{f(x)}); \mathsf{return}(x) \ : \mathbb{R}]\!]_{-,U} \qquad (10)\\
&= \int_U \mathit{gauss}(0,1)(\mathrm{d}x)\,(\tfrac{1}{f(x)})\\
&= \int_U \mathit{lebesgue}(\mathrm{d}x)\ (f(x))(\tfrac{1}{f(x)}) \qquad \text{since } \mathit{gauss}(0,1)(V) = \int_V \mathit{lebesgue}(\mathrm{d}x)\, f(x)\\
&= \mathit{lebesgue}(U)
\end{aligned}
$$

(On the third line, we use the definition of density function.)

Some languages (such as Stan [40], also Core Hakaru [38]) encourage the use of the Lebesgue measure as an ‘improper prior’. We return to the example of importance sampling, proposed in the introduction. Consider a probability measure p with density g. Then

$$
[\![\mathsf{sample}(p)]\!] \;=\; [\![\mathsf{let}\, x = \mathit{lebesgue}\, \mathsf{in}\, \mathsf{observe}\, x\, \mathsf{from}\, p; \mathsf{return}(x)]\!] \qquad (11)
$$

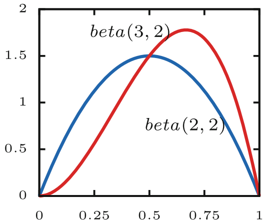
— a simple example of how an improper prior can lead to a proper posterior. We can derive the importance sampling algorithm for p by combining (11) with (10):

$$
\begin{aligned}
[\![\mathsf{sample}(p)]\!] &= [\![\mathsf{let}\, x = \mathit{lebesgue}\, \mathsf{in}\, \mathsf{observe}\, x\, \mathsf{from}\, p; \mathsf{return}(x)]\!]\\
&= [\![\mathsf{let}\, x = \mathit{gauss}(0,1)\, \mathsf{in}\, \mathsf{score}(\tfrac{1}{f(x)}); \mathsf{score}(g(x)); \mathsf{return}(x)]\!]\\
&= [\![\mathsf{let}\, x = \mathit{gauss}(0,1)\, \mathsf{in}\, \mathsf{score}(\tfrac{g(x)}{f(x)}); \mathsf{return}(x)]\!].
\end{aligned}
$$

Conjugate Prior Relationships and Symbolic Bayesian Update. A key technique for Bayesian inference involves conjugate prior relationships. In general, inference problems are solved by simulation, but sometimes we can work symbolically, when there is a closed form for updating the parameters of a prior according to an observation. In a probabilistic programming language, this symbolic translation can be done semi-automatically as a program transformation (see e.g. [5]).

Recall that $\mathit{beta}(\alpha,\beta)$ is a probability measure on $[0,1]$ describing the distribution of a bias of a coin from which we have observed $(\alpha-1)$ heads and $(\beta-1)$ tails. This has a conjugate prior relationship with the Bernoulli distribution. For instance,

$$
\begin{aligned}
&[\![\mathsf{let}\, x = \mathsf{sample}(\mathit{beta}(2,2))\, \mathsf{in}\, \mathsf{observe}\, 1\, \mathsf{from}\, \mathit{bern}(x); x]\!]\\
&=\\
&[\![\mathsf{observe}\, 1\, \mathsf{from}\, \mathit{bern}(\tfrac{2}{2+2}); \mathsf{sample}(\mathit{beta}(2+1,2))]\!]
\end{aligned}
$$



In the graph, notice that $\mathit{beta}(3,2)$ depicts the updated posterior belief of the bias of the coin after an additional observation: it is more probable that the coin is biassed to heads.

Resampling. In many situations, particularly in Sequential Monte Carlo simulations, it is helpful to freeze a simulation and resample from a histogram that has been built (e.g. [31]). In practical terms, this avoids having too many threads of low weight. Resampling in this way is justified by the following program equation:

$$\begin{aligned}[\![t]\!] \;=\; [\![&\mathsf{case}\,\mathsf{normalize}(t)\,\mathsf{of}\,(1,(e,d)) \Rightarrow \mathsf{score}(e);\mathsf{sample}(d)\\ &\qquad\qquad\qquad\qquad\;\;\mid (2,()) \Rightarrow \mathsf{score}(0);t\\ &\qquad\qquad\qquad\qquad\;\;\mid (3,()) \Rightarrow t \qquad\qquad]\!]\end{aligned}$$

Notice that we cannot resample if the model evidence is ∞. For example, we cannot resample from the expression above computing the Lebesgue measure (10), although of course this doesn't prevent us from resampling from programs that contain it (e.g. (11)).

Hard Constraints. A hard constraint is a score of 0; a non-zero score is a soft constraint. In our language, every type is inhabited, so for each type $\mathbb{A}$ we can define a term

$$\mathsf{fail}_{\mathbb{A}} \stackrel{\text{def}}{=} \mathsf{score}(0); f() : \mathbb{A} \tag{12}$$

picking arbitrary $f : 1 \to [\![\mathbb{A}]\!]$ at each type $\mathbb{A}$. The semantics is $[\![\mathsf{fail}_{\mathbb{A}}]\!]_{\gamma,U} = 0$.

Hard constraints suffice for scores below 1, because then

$$[\![\mathsf{score}(r)]\!] = [\![\mathsf{if}\;\mathsf{sample}(\mathit{bern}(r))\;\mathsf{then}\;()\;\mathsf{else}\;\mathsf{fail}_1]\!].$$

Hard constraints cannot express scores above 1, which can arise from continuous likelihoods — for instance, in the example in the introduction, the likelihoods were 0.82 and 1.42. Inference algorithms often perform better when soft constraints are used.

4.2 Basic Semantic Properties

Standard β/η laws and associativity of let. The standard β/η laws for sums and products hold. These are easy to verify. For instance,

$$[\![\mathsf{case}\,(i,t)\,\mathsf{of}\,\{(j,x) \Rightarrow u_j\}_{j\in I}]\!] \;=\; [\![u_i[t/x]]\!].$$

We also have the standard associativity and identity laws for let:

$$[\![\mathsf{let}\,x = \mathsf{return}(t)\,\mathsf{in}\,u]\!] = [\![u[t/x]]\!] \qquad\qquad [\![\mathsf{let}\,x = u\,\mathsf{in}\,\mathsf{return}(x)]\!] = [\![u]\!]$$

$$[\![\mathsf{let}\,y = (\mathsf{let}\,x = t\,\mathsf{in}\,u)\,\mathsf{in}\,v]\!] \;=\; [\![\mathsf{let}\,x = t\,\mathsf{in}\,\mathsf{let}\,y = u\,\mathsf{in}\,v]\!]$$

For instance, the associativity law follows from Lemma 3.

Commutativity.

Theorem 4. *For any terms $\Gamma \vdash_{\mathsf{p}} t \colon \mathbb{A}$, $\Gamma \vdash_{\mathsf{p}} u \colon \mathbb{B}$, $\Gamma, x : \mathbb{A}, y : \mathbb{B} \vdash_{\mathsf{p}} v \colon \mathbb{C}$, we have*

$$[\![\mathsf{let}\, x = t\, \mathsf{in}\, \mathsf{let}\, y = u\, \mathsf{in}\, v]\!] \;=\; [\![\mathsf{let}\, y = u\, \mathsf{in}\, \mathsf{let}\, x = t\, \mathsf{in}\, v]\!].$$

This theorem is an immediate consequence of Proposition 5:

Proposition 5. *Let μ and λ be s-finite measures on X and Y respectively, and let $f : X \times Y \to [0, \infty]$ be measurable. Then*

$$\int_X \mu(\mathrm{d}x) \int_Y \lambda(\mathrm{d}y)\, f(x,y) = \int_Y \lambda(\mathrm{d}y) \int_X \mu(\mathrm{d}x)\, f(x,y)$$

Proof. This result is known (e.g. [39]) and it is easy to prove. Since μ and λ are s-finite, we have $\mu = \sum_{i=1}^{\infty} \mu_i$ and $\lambda = \sum_{j=1}^{\infty} \lambda_j$, with the μ_i's and λ_j's all finite. Now,

$$\begin{aligned}
&\textstyle\int_X (\sum_i \mu_i)(\mathrm{d}x)\ \int_Y (\sum_j \lambda_j)(\mathrm{d}y)\, f(x,y) && \\
&\textstyle= \sum_i \int_X \mu_i(\mathrm{d}x)\ \sum_j \int_Y \lambda_j(\mathrm{d}y)\, f(x,y) && \text{using Proposition 2} \\
&\textstyle= \sum_i \sum_j \int_X \mu_i(\mathrm{d}x)\ \int_Y \lambda_j(\mathrm{d}y)\, f(x,y) && \text{using (8)} \\
&\textstyle= \sum_i \sum_j \int_Y \lambda_j(\mathrm{d}y)\ \int_X \mu_i(\mathrm{d}x)\, f(x,y) && \text{finite measures commute, [32, Theorem 25]} \\
&\textstyle= \sum_i \int_Y (\sum_j \lambda_j)(\mathrm{d}y)\ \int_X \mu_i(\mathrm{d}x)\, f(x,y) && \text{using Proposition 2} \\
&\textstyle= \int_Y (\sum_j \lambda_j)(\mathrm{d}y)\ \sum_i \int_X \mu_i(\mathrm{d}x)\, f(x,y) && \text{using (8)} \\
&\textstyle= \int_Y (\sum_j \lambda_j)(\mathrm{d}y)\ \int_X (\sum_i \mu_i)(\mathrm{d}x)\, f(x,y) && \text{using Proposition 2.}
\end{aligned}$$

(The commutativity for finite measures is often called Fubini's theorem.)

Iteration. We did not include iteration in our language but in fact it is definable. In brief, we can use the probabilistic constructs to guess how many iterations are needed for termination. (We do not envisage this as a good implementation strategy, we merely want to show that the language and semantic model can accommodate reasoning about iteration.)

In detail, we define a construction $\mathsf{iterate}\, t\, \mathsf{from}\, x{=}u$, that keeps calling t, starting from $x{=}u$; if t returns $u' : \mathbb{A}$, then we repeat with $x{=}u'$, if t finally returns in $\mathbb{B}$, then we stop. This has the following derived typing rule:

$$\frac{\Gamma, x : \mathbb{A} \vdash_{\mathsf{p}} t \colon (\mathbb{A} + \mathbb{B}) \qquad \Gamma \vdash_{\mathsf{d}} u \colon \mathbb{A}}{\Gamma \vdash_{\mathsf{p}} \mathsf{iterate}\, t\, \mathsf{from}\, x{=}u \colon \mathbb{B}}$$

We begin by defining the counting measure on $\mathbb{N}$, which assigns to each set its size. This is not a primitive, because it isn't a probability measure, but we can define it in a similar way to the Lebesgue measure:

$$\mathit{counting}_{\mathbb{N}} = [\![\, \vdash_{\mathsf{p}} \mathsf{let}\, x = \mathsf{sample}(\mathit{poisson}(1))\, \mathsf{in}\, \mathsf{score}(x!e); \mathsf{return}(x) \colon \mathbb{N}]\!] \qquad (13)$$

(Recall that the Poisson distribution has $\mathit{poisson}(1)(\{x\}) = \frac{1}{x!e}$.)

Now we can define

$$\mathsf{iterate}\,t\,\mathsf{from}\,x = u \overset{\text{def}}{=} \mathsf{case}\ \mathit{counting}_{\mathbb{N}}\ \mathsf{of}\ (n, ()) \Rightarrow \mathsf{iterate}^n\,t\,\mathsf{from}\,x = u$$

where $\Gamma \vdash_{\mathsf{p}} \mathsf{iterate}^n\,t\,\mathsf{from}\,x{=}u \colon \mathbb{B}$ is the program that returns $v : \mathbb{B}$ if t returns v after exactly n iterations and fails otherwise:

$$\begin{aligned}\mathsf{iterate}^1\,t\,\mathsf{from}\,x = u &\overset{\text{def}}{=} \mathsf{case}\ t[u/x]\ \mathsf{of}(1, u') \Rightarrow \mathsf{fail}\\ &\qquad\qquad\qquad\quad |(2, v) \Rightarrow \mathsf{return}(v)\\ \mathsf{iterate}^{n+1}\,t\,\mathsf{from}\,x = u &\overset{\text{def}}{=} \mathsf{case}\ t[u/x]\ \mathsf{of}(1, u') \Rightarrow \mathsf{iterate}^n\,t\,\mathsf{from}\,x = u'\\ &\qquad\qquad\qquad\quad |(2, v) \Rightarrow \mathsf{fail}\end{aligned}$$

For a simple illustration, von Neumann's trick for simulating a fair coin from a biassed one d can be written $d : \mathsf{P}(\mathsf{bool}) \vdash_{\mathsf{p}} \mathsf{iterate}\,t\,\mathsf{from}\,x{=}() \colon \mathsf{bool}$ where

$$\begin{aligned}t \overset{\text{def}}{=} &\ (\mathsf{let}\,y = \mathsf{sample}(d)\ \mathsf{in}\\ &\quad \mathsf{let}\,z = \mathsf{sample}(d)\ \mathsf{in}\ \mathsf{if}\ y \neq z\ \mathsf{then}\ \mathsf{return}(2, y)\ \mathsf{else}\ \mathsf{return}(1, ())) \quad : 1 + \mathsf{bool}\end{aligned}$$

We leave for future work the relation between this iteration and other axiomatizations of iteration (e.g. [12, Chap. 3]).

5 Remarks About s-Finite Kernels

5.1 Full Definability

Theorem 6. *If* $k : [\![\Gamma]\!] \rightsquigarrow [\![\mathbb{A}]\!]$ *is s-finite then there is a term* $\Gamma \vdash_{\mathsf{p}} t \colon \mathbb{A}$ *such that* $k = [\![t]\!]$.

Proof. We show that probability kernels are definable. Consider a probability kernel $k : [\![\Gamma]\!] \rightsquigarrow [\![\mathbb{A}]\!]$ with $\Gamma = (x_1 : \mathbb{B}_1 \dots x_n : \mathbb{B}_n)$. This corresponds to a measurable function $f : [\![\prod_{i=1}^n \mathbb{B}]\!] \to P([\![\mathbb{A}]\!])$, with $f(b_1, \dots, b_n)(U) = k(b_1, \dots, b_n, U)$, and $k = [\![\Gamma \vdash_{\mathsf{p}} \mathsf{sample}(f(x_1, \dots, x_n)) \colon \mathbb{A}]\!]$.

We move on to subprobability kernels, which are kernels $k : [\![\Gamma]\!] \rightsquigarrow [\![\mathbb{A}]\!]$ such that $k(\gamma, [\![\mathbb{A}]\!]) \leq 1$ for all γ. We show that they are all definable. Recall that to give a subprobability kernel $k : [\![\Gamma]\!] \rightsquigarrow [\![\mathbb{A}]\!]$ is to give a probability kernel $\bar{k} : [\![\Gamma]\!] \rightsquigarrow [\![\mathbb{A} + 1]\!]$. Define

$$\bar{k}(\gamma, U) = \begin{cases} k(\gamma, \{a \mid (1, a) \in U\}) + (1 - k(\gamma, [\![\mathbb{A}]\!])) & (2, ()) \in U \\ k(\gamma, \{a \mid (1, a) \in U\}) & \text{otherwise} \end{cases}$$

This probability kernel $\bar{k}$ is definable, with $\bar{k} = [\![t]\!]$, say, and this has the property that

$$k \;=\; [\![\mathsf{case}\,t\,\mathsf{of}\ (1, x) \Rightarrow \mathsf{return}(x) \mid (2, ()) \Rightarrow \mathsf{fail}]\!].$$

where fail is the zero kernel defined in (12). So the subprobability kernel k is definable.

Next, we show that all finite kernels $k : \llbracket \Gamma \rrbracket \rightsquigarrow \llbracket \mathbb{A} \rrbracket$ are definable. If k is finite then there is a bound $r \in (0, \infty)$ such that $k(\gamma, \llbracket \mathbb{A} \rrbracket) < r$ for all γ. Then $\frac{1}{r}k$ is a subprobability kernel, hence definable, so we have t such that $\frac{1}{r}k = \llbracket t \rrbracket$. So $k = r\llbracket t \rrbracket = \llbracket \mathsf{score}(r); t \rrbracket$.

Finally, if k is s-finite then there are finite kernels $k_i : \llbracket \Gamma \rrbracket \rightsquigarrow \llbracket \mathbb{A} \rrbracket$ such that $k = \sum_{i=1}^{\infty} k_i$. Since the k_i's are finite, we have terms t_i with $k_i = \llbracket t_i \rrbracket$. Recall that a countable sum is merely integration over the counting measure on $\mathbb{N}$, which we showed to be definable in (13). So we have $k = \llbracket \mathsf{case}\ \mathit{counting}_{\mathbb{N}}\ \mathsf{of}\ i \Rightarrow t_i \rrbracket$.

5.2 Failure of Commutativity in General

The standard example of the failure of Tonelli's theorem (e.g. [32, Chap. 4., Example 12] can be used to explain why the commutativity program Eq. (2) fails if we allow arbitrary measures as programs.

Let *lebesgue* be the Lebesgue measure on $\mathbb{R}$, and let $\mathit{counting}_{\mathbb{R}}$ be the counting measure on $\mathbb{R}$. Recall that $\mathit{counting}_{\mathbb{R}}(U)$ is the cardinality of U if U is finite, and ∞ if U is infinite. Then

$$\int_{\mathbb{R}} \mathit{lebesgue}(\mathrm{d}r) \int_{\mathbb{R}} \mathit{counting}_{\mathbb{R}}(\mathrm{d}s)\,[r = s] \;=\; \int_{\mathbb{R}} \mathit{lebesgue}(\mathrm{d}r)\,1 \;=\; \infty$$

$$\int_{\mathbb{R}} \mathit{counting}_{\mathbb{R}}(\mathrm{d}s) \int_{\mathbb{R}} \mathit{lebesgue}(\mathrm{d}r)\,[r = s] \;=\; \int_{\mathbb{R}} \mathit{counting}_{\mathbb{R}}(\mathrm{d}s)\,0 \;=\; 0$$

So, by Proposition 5, the counting measure on $\mathbb{R}$ is not s-finite, and hence it is not definable in our language. (This is in contrast to the counting measure on $\mathbb{N}$, see (13).)

Just for this subsection, we suppose that we can add the counting measure on $\mathbb{R}$ to our language as a term constructor $\vdash_{\mathsf{p}} \mathit{counting}_{\mathbb{R}} \colon \mathbb{R}$ and that we can extend the semantics to accommodate it. (This would require some extension of Lemma 3.) The Lebesgue measure is already definable in our language (10). In this extended language we would have

$$\llbracket \vdash_{\mathsf{p}} \mathsf{let}\, r = \mathit{lebesgue}\,\mathsf{in}\,\mathsf{let}\, s = \mathit{counting}_{\mathbb{R}}\,\mathsf{in}\,[r = s] \colon \mathsf{bool} \rrbracket_{(),\{\mathsf{true}\}} \;= \infty$$

$$\llbracket \vdash_{\mathsf{p}} \mathsf{let}\, s = \mathit{counting}_{\mathbb{R}}\,\mathsf{in}\,\mathsf{let}\, r = \mathit{lebesgue}\,\mathsf{in}\,[r = s] \colon \mathsf{bool} \rrbracket_{(),\{\mathsf{true}\}} \;= 0.$$

So if such as language extension was possible, we would not have commutativity.

5.3 Variations on s-Finiteness

Infinite versions of Fubini/Tonelli theorems are often stated for σ-finite measures. Recall that a measure μ on X is $\sigma - finite$ if $X = \biguplus_{i=1}^{\infty} U_i$ with each $U_i \in \Sigma_X$ and each $\mu(U_i)$ finite. The restriction to σ-finite measures is too strong for our purposes. For example, although the Lebesgue measure (*lebesgue*) is σ-finite, and definable (10), the measure $\llbracket \vdash_{\mathsf{p}} \mathsf{let}\, x = \mathit{lebesgue}\,\mathsf{in}\,() \colon 1 \rrbracket$ is the infinite measure on the one-point space, which is not σ-finite. This illustrates the difference between σ-finite and s-finite measures:

Proposition 7. *A measure is s-finite if and only if it is a pushforward of a σ-finite measure.*

Proof. From left to right, let $\mu = \sum_{i=1}^{\infty} \mu_i$ be a measure on X with each μ_i finite. Then we can form a σ-finite measure ν on $\mathbb{N} \times X$ with $\nu(U) = \sum_{i=1}^{\infty} \mu_i(\{x \mid (i,x) \in U\})$. The original measure μ is the pushforward of ν along the projection $\mathbb{N} \times X \to X$.

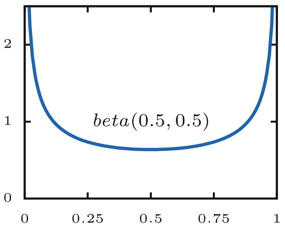
From right to left, let ν be a σ-finite measure on $X = \biguplus_{i=1}^{\infty} U_i$ with each restricted measure $\nu(U_i)$ finite. Let $f : X \to Y$ be measurable. For $i \in \mathbb{N}$, let $\mu_i(V) = \nu(\{x \in U_i \mid f(x) \in V\})$. Then each μ_i is a finite measure on Y and $\sum_{i=1}^{\infty} \mu_i$ is the pushforward of ν along f, as required. (See also [9, Lemma 8.6].)

However, this does not mean that s-finite kernels (Definition 2) are 'just' kernels whose images are pushforwards of σ-finite measures. In the proof of commutativity, we did only need kernels $k : X \rightsquigarrow Y$ such that $k(x)$ is an s-finite measure for all $x \in X$. This condition is implied by the definition of s-finite kernel (Definition 2) but the definition of s-finite kernel seems to be strictly stronger because of the uniformity in the definition. (This is not known for sure; see also the discussion about σ-finite kernels in [32, Sect. 4.10].) The reason we use the notion of s-finite kernel, rather than this apparently weaker notion, is that Lemma 3 (and hence the well-defined semantics of let) appears to require the uniformity in the definition of finite and s-finite kernels. In brief, the stronger notion of s-finite kernel provides a compositional semantics giving s-finite measures.

6 Concluding Remarks

6.1 Related Work on Commutativity for Probabilistic Programs

Work Using Finite Kernels. Several other authors have given a semantics for probabilistic programs using kernels. Subprobability kernels and finite measures already appear in Kozen's work on probabilistic programming [21]. Ramsey and Pfeffer [34] focus on a language like ours but without score or normalize; they give a semantics in terms of probability kernels. The measure-transformer-semantics of Börgstrom et al. [3] incorporates observations by moving to finite kernels; their semantics is similar to ours (Sect. 3.2), but they are able to make do with finite kernels by considering a very limited language. In the more recent operational semantics by Börgstrom et al. [4], problems of commutativity are avoided by requiring scores to be less than 1, so that all the measures are subprobability measures. Jacobs and Zanasi [18] also impose this restriction to make a connection with an elegant mathematical construction. With discrete countable distributions, this is fine because density functions and likelihoods lie below 1. But when dealing with continuous distributions, it is artificial to restrict to scores below 1, since the likelihood of a continuous distribution may just as well lie above 1 as below it. For example, the subprobability semantics could not handle the example in Sect. 1.1. This is not merely a matter of scaling, because density functions are sometimes unbounded, as shown in *beta*$(0.5, 0.5)$ on the right. Our results here show that, by using s-finite kernels, one can consider arbitrary likelihoods without penalty.



Verification Conditions for Commutativity. Shan and Ramsey [38] use a similar semantics to ours to justify their disintegration program transformation. They interpret a term $\Gamma \vdash_{\mathsf{p}} t \colon \mathbb{A}$ as a measurable function into a monad $\mathbb{M}$ of measures, $[\![t]\!]_{\mathrm{SR}} : [\![\Gamma]\!] \to \mathbb{M}(\mathbb{A})$, which actually amounts to the same thing as a kernel. However, there is a problem with the semantics in this style: we do not know a proof for Lemma 3 without the s-finiteness restriction. In other words, we do not know whether the monad of all measures $\mathbb{M}$ is a strong monad. A strength is needed to give a semantics for the let construction. So it is not clear whether the semantics is well-defined. Even if $\mathbb{M}$ is strong, it is certainly not commutative, as we have discussed in Sect. 5.2, a point also emphasized by Ramsey [35]. Shan and Ramsey [38] regain commutativity by imposing additional verification conditions. Our results here show that these conditions are always satisfied because all definable programs are s-finite kernels and hence commutative.

Contextual Equivalence. Very recently, Culpepper and Cobb [6] have proposed an operational notion of contextual equivalence for a language with arbitrary likelihoods, and shown that this supports commutativity. The relationship between their argument and s-finite kernels remains to be investigated.

Sampling Semantics. An alternative approach to denotational semantics for probabilistic programs is based on interpreting an expression $\Gamma \vdash_{\mathsf{p}} t \colon \mathbb{A}$ as a probability kernel $[\![t]\!]' : [\![\Gamma]\!] \rightsquigarrow ([0,\infty) \times [\![\mathbb{A}]\!])$, so that $[\![t]\!]'(\gamma)$ is a probability measure on pairs (r, x) of a result x and a weight r. In brief, the probability measure comes from sampling priors, and the weight comes from scoring likelihoods of observations. Börgstrom et al. [4] call this a *sampling* semantics by contrast with the *distribution* semantics that we have considered here. This sampling semantics, which has a more intensional flavour and is closer to an operational intuition, is also considered by Ścibor et al. [37] and Staton et al. [43], as well as Doberkat [7]. The two methods are related because every probability kernel $k : X \rightsquigarrow ([0,\infty) \times Y)$ induces a measure kernel $\bar{k} : X \rightsquigarrow Y$ by summing over the possible scores:

$$\bar{k}(x, U) \overset{\text{def}}{=} \int_{[0,\infty)\times U} k(x, \mathrm{d}(r, y))\, r \tag{14}$$

An advantage to the sampling semantics is that it is clearly commutative, because it is based on a commutative monad $(P([0,\infty) \times (-)))$, built by combining the commutative Giry monad P and the commutative monoid monad transformer. However, the sampling semantics does not validate many of the semantic equations in Sect. 4.1: importance sampling, conjugate priors, and resampling are only sound in the sampling semantics if we wrap the programs in $\mathsf{normalize}(\dots)$. (See e.g. [43].) This makes it difficult to justify applying program transformations compositionally. The point of this paper is that we can verify the semantic equations in Sect. 4.1 directly, while retaining commutativity, by using the measure based (distributional) semantics.

As an aside we note that the probability kernels $X \rightsquigarrow ([0,\infty) \times Y)$ used in the sampling semantics are closely related to the s-finite kernels advocated in this paper:

Proposition 8. *A kernel $l : X \rightsquigarrow Y$ is s-finite if and only if there exists a probability kernel $k : X \rightsquigarrow ([0,\infty) \times Y)$ and $l(x,U) = \int_{[0,\infty)\times U} k(x, \mathrm{d}(r,y))\, r$.*

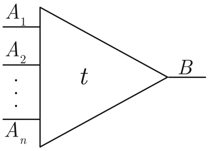
Proof (notes). We focus on the case where $X = [\![\mathbb{A}]\!]$ and $Y = [\![\mathbb{B}]\!]$. From left to right: build a probability kernel from an s-finite kernel by first understanding it as a probabilistic program (via Theorem 6) and then using the denotational semantics in [43]. From right to left: given a probability kernel $k : [\![\mathbb{A}]\!] \rightsquigarrow ([0,\infty) \times [\![\mathbb{B}]\!])$, we build an s-finite kernel

$$[\![x : \mathbb{A} \vdash_{\mathsf{p}} \mathsf{let}\,(r,y) = \mathsf{sample}(k(x))\,\mathsf{in}\,\mathsf{score}(r); \mathsf{return}(y) : \mathbb{B}]\!] : [\![\mathbb{A}]\!] \rightsquigarrow [\![\mathbb{B}]\!].$$

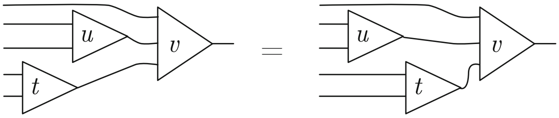
Valuations Versus Measures. Some authors advocate using valuations on topological spaces instead of measures on measurable spaces. This appears to rule out the problematic examples, such as the counting measure on $\mathbb{R}$. Indeed, Vickers [45] has shown that a monad of valuations on locales is commutative. This suggests a constructive or topological model of probabilistic programming (see [8,15]) but a potential obstacle is that conditioning is not always computable [1].

6.2 Related Work on Commutativity More Generally

Multicategories and Data Flow Graphs. An early discussion of commutativity is in Lambek's work on deductive systems and categories [22]. A judgement $x_1 : A_1, \ldots, x_n : A_n \vdash t : B$ is interpreted as a multimorphism $(A_1 \ldots A_n) \to B$. These could be drawn as triangles:



(This hints at a link with the graphical ideas underlying several probabilistic programming languages e.g. Stan [40].) Alongside requiring associativity of composition, Lambek requires commutativity:



which matches with our commutativity condition (2). (See also [42].) In this diagrammatic notation, commutativity says that the semantics is preserved under topological transformations. Without commutativity, one would need extra control flow wires to give a topological description of what rewritings are acceptable (e.g. [19,28]). Our main technical results (Lemma 3 and Proposition 5) can be phrased as follows:

> *Measurable spaces and s-finite kernels $X_1 \times \cdots \times X_n \rightsquigarrow Y$ form a multicategory.*

Monoidal Categories, Monads and Arrows. There is a tight connection between multicategories and monoidal categories [13,24,42]. Our main technical results (Lemma 3 and Proposition 5) together with the basic facts in Sect. 4.2 can be phrased as follows:

> *Consider the category whose objects are measurable spaces and morphisms are s-finite kernels. The cartesian product of spaces extends to a monoidal structure which distributes over the coproduct structure.*

From this point of view, the key step is that given s-finite kernels $k : X_1 \rightsquigarrow Y_1$ and $k_2 : X_2 \rightsquigarrow Y_2$, we can form $(k_1 \otimes k_2) : X_1 \times X_2 \rightsquigarrow Y_1 \times Y_2$, with

$$(k_1 \otimes k_2)((x_1, x_2), U) = \int_{X_1} k_1(x_1, \mathrm{d}y_1) \int_{X_2} k_2(x_2, \mathrm{d}y_2)[(y_1, y_2) \in U]$$

and the interchange law holds, in particular, $(k_1 \otimes \mathrm{id}) \circ (\mathrm{id} \otimes k_2) = (\mathrm{id} \otimes k_2) \circ (k_1 \otimes \mathrm{id})$.

One way of building monoidal categories is as Kleisli categories for commutative monads. For example, the monoidal category of probability kernels is the Kleisli category for the Giry monad [10]. However, we conjecture that s-finite kernels do *not* form a Kleisli category for a commutative monad on the category of measurable spaces. One could form a space $M_{\mathsf{sfin}}(Y)$ of s-finite measures on a given space Y, but, as discussed in Sect. 5.3, it is unlikely that every measurable function $X \to M_{\mathsf{sfin}}(Y)$ is an s-finite kernel in general, because of the uniformity in the definition (Definition 2). This makes it difficult to ascertain whether M_{sfin} is a strong commutative monad. Having a monad would give us a slightly-higher-order type constructor $T(A)$ and the rules

$$\frac{\Gamma \models_{\mathsf{p}} t \colon \mathbb{A}}{\Gamma \models_{\mathsf{d}} \mathsf{thunk}(t) \colon T(\mathbb{A})} \qquad \frac{\Gamma \models_{\mathsf{d}} t \colon T(\mathbb{A})}{\Gamma \models_{\mathsf{p}} \mathsf{force}(t) \colon \mathbb{A}}$$

allowing us to thunk (suspend, freeze) a probabilistic computation and then force (resume, run) it again [25,29]. The rules are reminiscent of, but not the same as, the rules for $\mathsf{normalize}$ and sample. Although monads are a convenient way of building a semantics for programming languages, they are not essential for first order languages such as the language in this paper.

As a technical aside we recall that Power, Hughes and others have eschewed monads and given categorical semantics for first order languages in terms of Freyd categories [25] or Arrows [16] (see also [2,17,41]), and the idea of structuring the finite kernels as an Arrow already appears in the work of Börgstrom et al. [3] (see also [36,44]). Our semantics based on s-finite kernels forms a 'countably distributive commutative Freyd category', which is to say that the identity-on-objects functor

$$\begin{pmatrix} \text{measurable spaces} \\ \text{\& measurable functions} \end{pmatrix} \longrightarrow \begin{pmatrix} \text{measurable spaces} \\ \text{\& s-finite kernels} \end{pmatrix}$$

preserves countable sums and is monoidal. In fact every countably distributive commutative Freyd category $\mathcal{C} \to \mathcal{D}$ corresponds to a commutative monad, not on the category $\mathcal{C}$ but on the category of countable-product-preserving functors $\mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ (e.g. [33,43]). This functor category is cartesian closed, and so it is also a fairly canonical semantics for higher order programs. (For a more concrete variant, see also [14].)

6.3 Summary

We have given a denotational semantics for a probabilistic programming language using s-finite kernels (Sect. 3.2). Compositionality relied on a technical lemma (Lemma 3). This semantic model supports reasoning based on statistical techniques (Sect. 4.1), such as conjugate priors, as well as basic equational reasoning (Sect. 4.2), such as commutativity (Theorem 4). The model is actually completely described by the syntax, according to our full definability theorem (Theorem 6).

Acknowledgements. I am profoundly grateful to my coauthors on [43] for many discussions about the problems and examples in this subject (C. Heunen, O. Kammar, F. Wood, H. Yang). The MFPS 2016 special session on Probabilistic Programming was also illuminating and it was helpful to discuss these issues with the participants (J Börgstrom, D Roy, CC Shan and others). Thanks too to Adam Ścibior and the ESOP reviewers.

Research supported by a Royal Society University Research Fellowship.

References

1. Ackerman, N.L., Freer, C.E., Roy, D.M.: Noncomputable conditional distributions. In: Proceedings of the LICS 2011 (2011)
2. Atkey, R.: What is a categorical model of arrows? In: Proceedings of the MSFP 2008 (2008)
3. Borgström, J., Gordon, A.D., Greenberg, M., Margetson, J., van Gael, J.: Measure transformer semantics for Bayesian machine learning. LMCS **9**(3), 11 (2013)
4. Borgström, J., Lago, U.D., Gordon, A.D., Szymczak, M.: A lambda-calculus foundation for universal probabilistic programming. In: Proceedings of the ICFP (2016)
5. Carette, J., Shan, C.-C.: Simplifying probabilistic programs using computer algebra. In: Gavanelli, M., Reppy, J. (eds.) PADL 2016. LNCS, vol. 9585, pp. 135–152. Springer, Cham (2016). doi:10.1007/978-3-319-28228-2_9
6. Culpepper, R., Cobb, A.: Contextual equivalence for probabilistic programs with continuous random variables and scoring. In: Proceedings of the ESOP 2017 (2017, to appear)
7. Doberkat, E.E.: Stochastic Relations: Foundations for Markov Transition Systems. Chapman & Hall, London (2007)
8. Faissole, F., Spitters, B.: Synthetic topology in homotopy type theory for probabilistic programming. In: Proceedings of the PPS 2017 (2017)
9. Getoor, R.K.: Excessive Measures. Birkhäuser (1990)
10. Giry, M.: A categorical approach to probability theory. Categorical Aspects Topology Anal. **915**, 68–85 (1982)

11. Goodman, N., Mansinghka, V., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. In: UAI (2008)
12. Haghverdi, E.: A categorical approach to linear logic, geometry of proofs and full completeness. Ph.D. thesis, Ottawa (2000)
13. Hermida, C.: Representable multicategories. Adv. Math. **151**, 164–225 (2000)
14. Heunen, C., Kammar, O., Staton, S., Yang, H.: A convenient category for higher-order probability theory (2017). arXiv:1701.02547
15. Huang, D., Morrisett, G.: An application of computable distributions to the semantics of probabilistic programs: part 2. In: Proceedings of the PPS 2017 (2017)
16. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. **37**(1–3), 67–111 (2000)
17. Jacobs, B., Heunen, C., Hasuo, I.: Categorical semantics for arrows. J. Funct. Program. **19**(3–4), 403–438 (2009)
18. Jacobs, B., Zanasi, F.: A predicate/state transformer semantics for Bayesian learning. In: Proceedings of the MFPS 2016 (2016)
19. Jeffrey, A.: Premonoidal categories and a graphical view of programs. Unpublished (1997)
20. Kallenberg, O.: Stationary and invariant densities and disintegration kernels. Probab. Theory Relat. Fields **160**, 567–592 (2014)
21. Kozen, D.: Semantics of probablistic programs. J. Comput. Syst. Sci. **22**, 328–350 (1981)
22. Lambek, J.: Deductive systems and categories II. In: Hilton, P.J. (ed.) Category Theory, Homology Theory and Their Applications. LNM, vol. 86, pp. 76–122. Springer, Heidelberg (1969)
23. Last, G., Penrose, M.: Lectures on the Poisson process. CUP (2016)
24. Leinster, T.: Higher operads, higher categories. CUP (2004)
25. Levy, P.B., Power, J., Thielecke, H.: Modelling environments in call-by-value programming languages. Inf. Comput. **185**(2), 182–210 (2003)
26. Mansinghka, V.K., Selsam, D., Perov, Y.N.: Venture: a higher-order probabilistic programming platform with programmable inference (2014). http://arxiv.org/abs/1404.0099
27. Mardare, R., Panangaden, P., Plotkin, G.: Quantitative algebraic reasoning. In: Proceedings of the LICS 2016 (2016)
28. Møgelberg, R.E., Staton, S.: Linear usage of state. Logical Methods Comput. Sci. **10** (2014)
29. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991)
30. Narayanan, P., Carette, J., Romano, W., Shan, C., Zinkov, R.: Probabilistic inference by program transformation in Hakaru (system description). In: Kiselyov, O., King, A. (eds.) FLOPS 2016. LNCS, vol. 9613, pp. 62–79. Springer, Cham (2016). doi:10.1007/978-3-319-29604-3_5
31. Paige, B., Wood, F.: A compilation target for probabilistic programming languages. In: ICML (2014)
32. Pollard, D.: A user's guide to measure theoretic probability. CUP (2002)
33. Power, J.: Generic models for computational effects. TCS **364**(2), 254–269 (2006)
34. Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: POPL (2002)
35. Ramsey, N.: All you need is the monad.. what monad was that again? In: PPS Workshop (2016)
36. Scherrer, C.: An exponential family basis for probabilistic programming. In: Proceedings of the PPS 2017 (2017)

37. Ścibor, A., Ghahramani, Z., Gordon, A.D.: Practical probabilistic programming with monads. In: Proceedings of the Haskell Symposium. ACM (2015)
38. Shan, C.C., Ramsey, N.: Symbolic Bayesian inference by symbolic disintegration (2016)
39. Sharpe, M.: General Theory of Markov Processes. Academic Press, Cambridge (1988)
40. Stan Development Team: Stan: A C++ library for probability and sampling, version 2.5.0 (2014). http://mc-stan.org/
41. Staton, S.: Freyd categories are enriched Lawvere theories. In: Algebra, Coalgebra and Topology, ENTCS, vol. 303 (2013)
42. Staton, S., Levy, P.: Universal properties for impure programming languages. In: Proceedings of the POPL 2013 (2013)
43. Staton, S., Yang, H., Heunen, C., Kammar, O., Wood, F.: Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In: Proceedings of the LICS 2016 (2016)
44. Toronto, N., McCarthy, J., Van Horn, D.: Running probabilistic programs backwards. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 53–79. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46669-8_3
45. Vickers, S.: A monad of valuation locales available from the author's website (2011)
46. Wood, F., van de Meent, J.W., Mansinghka, V.: A new approach to probabilistic programming inference. In: AISTATS (2014)