Probabilistic Programs as Measures

Sam Staton
University of Oxford

Abstract: This chapter is a tutorial about some of the key issues in semantics of the first-order aspects of probabilistic programming languages for statistical modelling – languages such as Church, Anglican, Venture and WebPPL. We argue that s-finite measures and s-finite kernels provide a good semantic basis.

1 Introduction

This Chapter is about a style of probabilistic programming for building statistical models, the basis of languages such as Church (Goodman et al., 2008), WebPPL (Goodman and Stuhlmüller, 2014), Venture (Mansinghka et al., 2014), Anglican (Wood et al., 2014) and Hakaru (Narayanan et al., 2016).

The key idea of these languages is that the model is a combination of three things:

Sample: A generative model is described by a program involving not only binary random choices but also by sampling from continuous real-valued distributions. In Bayesian terms, we think of this as describing the prior probabilities.

Observe: Observations about data can be incorporated into the model, and these are typically used as weights in a Monte Carlo simulation. In Bayesian terms, we think of this as describing the likelihood of the data.

Normalize: Given a model, we run an inference algorithm over it to calculate the posterior probabilities.

Probabilistic programming languages bring many of the abstract ideas of

high-level programming to bear on statistical modelling. Perhaps the most compelling aspect is the idea of rapid development, first of quickly creating models, and second quickly combining them with inference algorithms.

There remain many practical and theoretical challenges with probabilistic languages of these kinds. The purpose of this chapter is to explain, for simple first order programs, how we can understand them as measures in a compositional way.

We begin in Section 2 by introducing the general approach to probabilistic programming and giving informal consideration to various aspects of the semantics of probabilistic programs. We are led to the issue of unnormalizable posteriors (§2.4). In Section 3 we develop the informal semantics from a measure-theoretic perspective, demonstrating through examples why a naive semantics is not so straightforward (§3.3).

In Section 4 we give a formal semantics for first order probabilistic programs as measures. We do this by understanding expressions with free variables as *s-finite kernels* (Def. 1.6). An s-finite kernel is, roughly, a parameterized measure that is uniformly built from finite measures. Once this semantics is given, one can easily reason about probabilistic programs in a compositional way by using measure theory, the standard basis of probability. We give some simple examples in Section 5.

2 Informal semantics for probabilistic programming

2.1 A first example: discrete samples, discrete observation

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

(i) I have forgotten what day it is.
(ii) There are ten buses per hour in the week and three buses per hour at the weekend.
(iii) I observe four buses in a given hour.
(iv) What is the probability that it is the weekend?

This is a very simple scenario, to illustrate the key points, but in practice, probabilistic programming is used for scenarios with dozens of interconnected random parameters and thousands of observations.

We assume that buses arrive as a Poisson process, meaning that their rate is given but they come independently. So the number of buses forms a Poisson distribution (Figure 1). We model the idea that the day is unknown

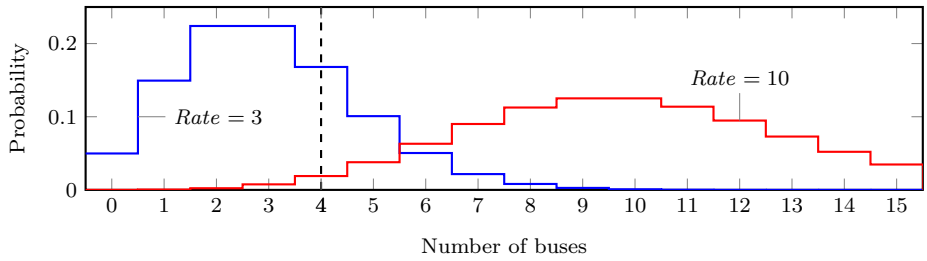

Figure 1 The Poisson distributions with rates 3 and 10.

by putting a prior belief that all the days are equiprobable. The problem would be written in statistical notation as follows:

(i) Prior: $x \sim \text{Bernoulli}(\frac{2}{7})$
(ii) Observation: $d \sim \text{Poisson}(r)$ where $r = 3$ if x and $r = 10$ otherwise;
(iii) $d = 4$;
(iv) What is the posterior distribution on x?

We describe this as a probabilistic program as follows:

```
1. normalize(
2.     let x = sample(bernoulli(2/7)) in
3.     let r = if x then 3 else 10 in
4.     observe 4 from poisson(r);
5.     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 the weekend is $\frac{2}{7}$; this matches line (i) above. On line 3, we set the rate r of buses, depending on whether it is a week day. On line 4 we record the observation that four buses passed when the rate was r, using the Poisson distribution. So lines 3 and 4 match lines (ii) and (iii) above (but not individually). The normalize command on line 1 is wrapped around the whole program up to the return value on line 5, and corresponds to line (iv) above.

There are three naive ways to calculate the answer:

Posterior calculation 1: direct calculation using Bayes' law. The first approach is to calculate the posterior probability using Bayes' law directly

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

For a discrete distribution, the likelihood is the probability of the observation point d, which for the Poisson distribution with rate r is $\frac{1}{d!}r^d e^{-r}$.

- The prior probability that it is the weekend is $\frac{2}{7}$, and then the likelihood of the observation is $\frac{1}{4!}3^4 e^{-3} \approx 0.168$; so the posterior probability that it is the weekend is proportional to $0.168 \times \frac{2}{7} \approx 0.048$ (likelihood×prior).
- The prior probability that it is a week day is $\frac{5}{7}$, and then the rate is 10 and the likelihood of the observation is $\frac{1}{4!}10^4 e^{-10} \approx 0.019$. So the posterior probability that it is a week day is proportional to $0.019 \times \frac{5}{7} \approx 0.014$.
- The measure $(\mathsf{true} \mapsto 0.048, \mathsf{false} \mapsto 0.014)$ is not a probability measure because it doesn't sum to 1. To build a probability measure we divide by $0.048 + 0.014 = 0.062$, 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.

Posterior Calculation 2: Monte Carlo simulation with rejection. In more complicated scenarios, it is often impractical to manage a direct numerical calculation like the above, and so people often turn to approximate simulation methods. A simulation with rejection works as follows:

- We run through the inner program (lines 2–5) a large number of times (say N).
- At a sample statement, we randomly sample from the given distribution. In Line 2, there is a Bernoulli trial that produces true with probability $\frac{2}{7}$ and false with probability $\frac{5}{7}$. We might perform this by uniformly generating a random number between 1 and 7 (the day of the week) and then returning true if the number is 6 or 7.
- At an observe statement, we also randomly sample from the given distribution, but we reject the run if the sample does not match the observation. In Line 4, we would sample a number k from the Poisson distribution with rate either 3 or 10, depending on the outcome of Line 2 (according to Line 3) and then reject the run if $k \neq 4$. This amounts to running a simulation of the bus network, but then rejecting the run if the outcome of the simulation did not match our observation. That is to say, we disregard or ignore the runs where the prior sample is inconsistent with the observation.
- Line 5 says that the result of the run is $x = \mathsf{true}$ if it is the weekend on that run.
- Line 1, wrapped around the whole program, says that of the non-rejected runs, we see what proportion of runs returns $x = \mathsf{true}$. As $N \to \infty$, the ratio will tend towards $(0.22 : 0.78)$, the true posterior distribution. Thus

the normalize command converts the sampler described by Lines 2–5 into a proper probability distribution.

Posterior Calculation 3: Monte Carlo simulation with weights. The rejection method is rather wasteful, and doesn't scale clearly to the continuous situations that we turn to later. An alternative is a simulation with likelihood weights, which works as follows:

- We run through the inner program (Lines 2–5) a large number N of times.
- As before, at a sample statement, we randomly sample from the given distribution.
- At an observe statement, we do not sample. Rather, we use the density function of the given distribution to weight the run. In Line 4, the density function of the Poisson distribution is $\frac{1}{d!}r^d e^{-r}$, so we weight the run by either 0.168 or 0.019, depending on the outcome of Line 2. In a program with multiple observations, we accumulate the weights multiplicatively. (In practice it is numerically prudent to use log-weights and add them.)
- Looking at all the runs, we see what weighted proportion of runs returns $x = \mathsf{true}$. As $N \to \infty$, the ratio will tend towards $(0.22 : 0.78)$.

In this discrete setting we can encode rejection sampling using a Monte Carlo simulation with weights, by replacing Line 4 with

$4'.$ $\quad \mathsf{let}\, d = \mathsf{sample}(\mathit{poisson}(r))\, \mathsf{in}\, \mathsf{observe}\, 4\, \mathsf{from}\, \mathit{dirac}(d)$

so that the weight will be either 1 (if $d = 4$) or 0 (if $d \neq 4$). When the weight is zero the run is as good as rejected.

2.2 A second example: discrete samples, continuous observation

Now consider the following situation, which is almost the same but the observation is different: we observe a 15 minute gap rather than four buses.

(i) I have forgotten what day it is.
(ii) There are ten buses per hour in the week and three buses per hour at the weekend.
(iii) I observe a 15 minute gap between two buses.
(iv) What is the probability that it is a week day?

In this example, since the buses are run as a Poisson process, the gap between them is exponentially distributed (Figure 2). The exponential distribution is a continuous probability measure on the positive reals; when the rate is r it has density function $t \mapsto re^{-rt}$. which means that the probability that the gap between events will lie in a given interval U is given by $\int_U re^{-rt}\,\mathrm{d}t$.

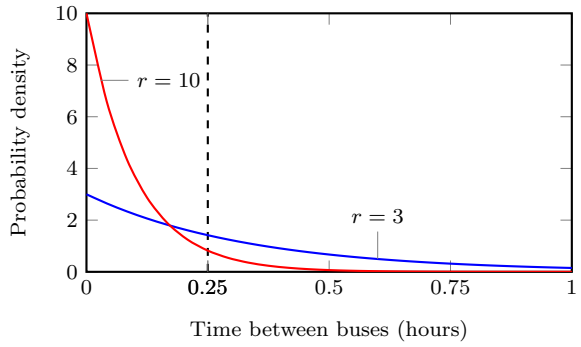

Figure 2 The exponential distributions with rates $r = 3$ and $r = 10$.

In statistical notation, this example would be described as follows:

(i) Prior: $x \sim \text{Bernoulli}(\frac{2}{7})$
(ii) Observation: $d \sim \text{Exponential}(r)$ where $r = 3$ if x and $r = 10$ otherwise;
(iii) $d = \frac{15}{60} = 0.25$;
(iv) What is the posterior distribution on x?

The program for this example differs from the previous one only on Line 4:

4″. observe 0.25 from *exponential*(r);

Posterior calculation 1 (direct mathematical calculation) is easily adapted to this situation. Here the likelihood of the observation (15 mins) is again the value of the density function, which is $3 \times e^{-3\times 0.25} \approx 1.42$ when it is the weekend and $10 \times e^{-10\times 0.25} \approx 0.82$ when it is a week day. So the unnormalized posterior has ($\mathsf{true} \mapsto \frac{2}{7} \times 1.42 \approx 0.405, \mathsf{false} \mapsto \frac{5}{7} \times 0.82 \approx 0.586$). In this example the normalizing constant is 0.991, and the normalized posterior is (0.408 : 0.592). Notice that likelihood is not the same as probability — it is not even less than 1.

Posterior calculation 2 (rejection sampling) cannot easily be adapted to this situation. The problem is that although sampling from an exponential distribution will often produce numbers that are close to 0.25, it will almost never produce exactly 0.25, so almost all the runs will be rejected.

Posterior calculation 3 (weighted sampling) is easily adapted to this situation. The weight on Line 4″ will either be 1.42 or 0.82.

Calculation Method 2 (Rejection) is perhaps the most intuitive, so it is

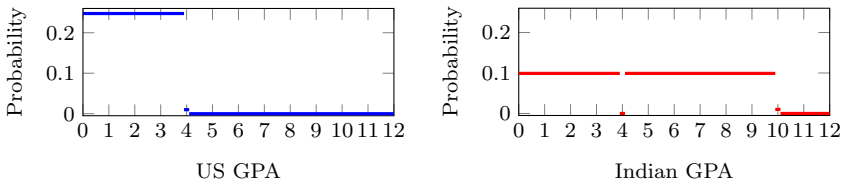

Figure 3 Discontinuous density functions for the GPA problem. See also Wu et al. (2018) and Section 3.2. The idea is this: suppose that grades are distributed uniformly, except the top 1% are given the maximum grade, which is 4 in the US and 10 in India. The problem is: given that I observe a GPA of 4, what is probable nationality of the student? The answer: certainly US.

unfortunate that it does not apply to this situation – not even theoretically. One way to resolve this is to say that our observation is not *precisely* 15 minutes, but $15 \pm \epsilon$ minutes. For all $\epsilon > 0$ we can make a rejection sampling algorithm which rejects all runs where the gap is not within $15 \pm \epsilon$. In an analogous way to line $4'$, we can encode rejection sampling in an interval with weighted sampling, by replacing line $4''$ by

$4'''$a. $\quad$ let $d = \mathsf{sample}(\mathit{exponential}(r))$ in

$4'''$b. $\quad$ observe d from $\mathit{uniform}(0.25 - \epsilon, 0.25 + \epsilon)$

As $\epsilon \to 0$, in this example, the posterior probability from rejection sampling tends to the posterior probability from weighted sampling.

(This is not a practical approach at all because, for small ϵ, the vast majority of runs will be rejected. One practical solution to soften the hard rejection constraint using noise from a normal distribution, e.g.

$4'''$b$'$. $\quad$ observe d from $\mathit{normal}(0.25, \frac{\epsilon}{2})$

Here we use $\frac{\epsilon}{2}$ as a small standard deviation.)

The correctness of this argument depends on some continuity issues, which have been investigated in the setting of conditional probability by Tjur (1980, §9.12) and Ackerman et al. (2015). On the other hand, densities that arise in practice are not always continuous: the GPA problem is an example of this that has been studied in the probabilistic programming context (see e.g. Figure 3, and §3.2, and Nitti et al., 2016; Wu et al., 2018).

In order to describe a situation as a program in this way, especially in a way that is amenable to Calculation Method 3 (Weighted sampling), the likelihood function of the observation distribution must be known. Research on automatic density calculation is ongoing (Bhat et al., 2017; Gehr et al., 2016; Ismail and chieh Shan, 2016).

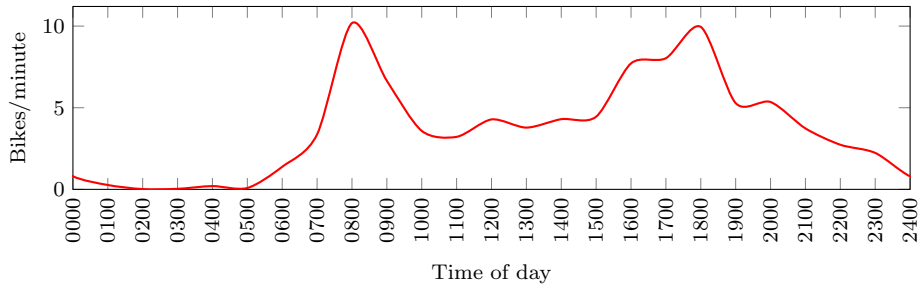

Figure 4 The rate of bikes as a function of the current time. The function is fictitious but based on real observations by "Bells on Bloor" in Toronto (Koehl et al., 2017).

2.3 A third example: continuous samples, continuous observations

For a third example, we use a similar story but now with bikes rather than buses, and rather than guess the day of the week we guess the time of day.

(i) I have forgotten what time it is.
(ii) The rate of bikes per hour is determined by a function of the time of day.
(iii) I observe a 1 minute gap between two bikes.
(iv) What time is it?

We model the idea that the time is unknown by picking the uniform distribution on the continuous interval $(0, 24)$. Suppose that we have some idea of the number of bikes per hour; the rate $f(t)$ will vary according to the time t. A possible f is given in Figure 4. In statistics notation, we would write:

(i) Prior: $t \sim \text{Uniform}(0, 24)$;
(ii) Observation: $d \sim \text{Exponential}(f(t))$;
(iii) $d = 0.0167$;
(iv) What is t?

The program for this example has the same outline as the previous one:

```
1. normalize(
2.     let t = sample(uniform(0, 24)) in
3.     let r = f(t) in
4.     observe 0.0167 from exponential(r);
5.     return(x))
```

Now to make Calculation Method 3 (weighted sampling) work, we need to accept that the prior and posterior distributions are on an uncountable space.

On a discrete computer it is not really possible to sample from an uncountable continuous distribution. One way to deal with this is to approximate the prior (and hence the posterior) by discrete distributions; the finer the granularity the closer the approximation is to the continuous distribution.

A secondary problem is that even a discretized sample space is too large to explore naively; many runs will have low weights (i.e. improbable) which is a waste of resources. There are Monte Carlo algorithms that perform this more efficiently, and can be applied to probabilistic programs, for example:

- Markov Chain Monte Carlo / Metropolis Hastings: with each run, we do not resample all the random choices, but only some, and we randomly reject or accept the resample depending on the change in weight. In other words, we build a Markov chain from the program and perform a random walk over it.
- Sequential Monte Carlo: we can run N times up to a checkpoint (typically an observation), pause, and redistribute the effort so that not too many of the running threads have low weight.

There are elaborations and combinations of these methods, together with other methods (such as variational ones). The introduction by van de Meent et al. (2018) covers many of these different methods.

For Posterior Calculation 1 (direct mathematical calculation), in this instance, we can give a posterior probability in terms of a probability density function. Recall that the meaning of density functions applied to probabilities (as opposed to likelihoods) is as follows: although the probability that the time is exactly 05:30 is zero, we can give a probability that the time is in some interval (more generally, a measurable set), as the integral of the density function. For instance, the posterior probability that the time is between 4am and 7am is shaded in Figure 5. The density function in this case is given by multiplying the likelihood function by the density of the prior distribution, which is uniform:

$$\begin{array}{rcccc} \text{Posterior} & \propto & \text{Likelihood} & \times & \text{Prior} \\ \text{posterior-pdf}(t) & \propto & f(t)e^{-0.016\times f(t)} & \times & \frac{1}{24} \end{array}$$

The density function $t \mapsto f(t)e^{-0.016\times f(t)} \times \frac{1}{24}$ is not normalized, but we can divide by the normalizing constant to get a true posterior density function:

$$t \;\mapsto\; \frac{f(t)e^{-0.016\times f(t)} \times \frac{1}{24}}{\int_0^{24} f(t)e^{-0.016\times f(t)} \times \frac{1}{24}\,\mathrm{d}t} = \frac{f(t)e^{-0.016\times f(t)}}{\int_0^{24} f(t)e^{-0.016\times f(t)}\,\mathrm{d}t} \tag{2}$$

In general, we cannot naively use density functions for a full compositional

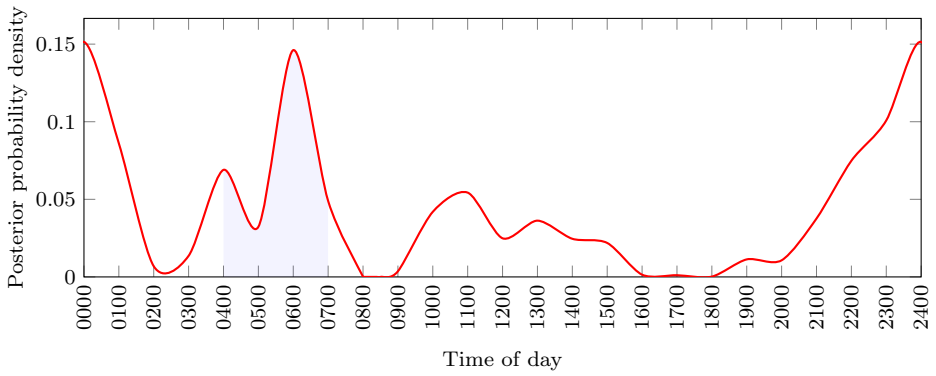

Figure 5 Posterior density for the current time given that I noticed a one minute gap between bikes when the rate is as shown in Figure 4. The probability that the time is between 4am and 7am is the purple area.

semantics because some basic programs do not have density functions. We return to this point in Section 3.

Aside on probabilistic programming for rapid prototyping

To briefly demonstrate the power of probabilistic programming for rapid prototyping, we consider a few elaborations on the last example. Supposing that the frequency $f(t)$ is uncertain, say we only know the frequency ± 1, then we can quickly introduce an extra random variable by changing line 3 to

3'. $\quad \mathsf{let}\, r = \mathsf{sample}(\mathit{normal}(f(t), 1))\, \mathsf{in} \ldots$

If the error in the frequency $f(t)$ is itself unknown, we can introduce yet another random variable σ for the error, for example,

3"a. $\quad \mathsf{let}\, \sigma = \mathsf{sample}(\mathit{inv\text{-}gamma}(2, 1))\, \mathsf{in}$

3"b. $\quad \mathsf{let}\, r = \mathsf{sample}(\mathit{normal}(f(t), \sigma))\, \mathsf{in} \ldots$

2.4 Unnormalizable posteriors

This chapter is about semantics of probabilistic programs and so it is informative to consider some corner cases. Recall that when we calculate a posterior we must divide by a normalizing constant. If this constant is 0 or ∞, we cannot find a posterior. In practice, if the constant is very low or very high, it suggests the model is bad, and it is numerically inconvenient to find the posterior, but if it is 0 or ∞ it is impossible even in theory.

Zero normalizing constant

A normalizing constant of 0 occurs when an observation is not only improbable, but impossible. For example, in the first example, suppose that we say that we claim to observe (-42) buses – a negative number of buses. This is impossible, nonsense, and the likelihood is not just very small but 0. In the rejection sampling semantics, all runs will be rejected.

Whether a normalizing constant is 0 is undecidable in general. For example, consider a Turing machine M with initial tape, and the following scenario.

(i) We toss a coin repeatedly until the outcome is heads. Call the number of tosses k.
(ii) We observe that Turing machine M terminates after exactly k steps.
(iii) What is k?

The prior distribution on k is a geometric distribution. The normalizing constant is non-0 if and only if the machine M terminates, in which case the posterior probability is the Dirac distribution on the number of steps required. For this reason, finding the normalizing constant is undecidable in general.

This manifests in practice as follows. For many Monte Carlo methods, it is guaranteed that sampling will converge eventually. However, it is difficult in practice to know when a Monte Carlo process has converged, and as this example shows, it may be impossible to know.

Infinite normalizing constant

Very high normalizing constants can occur when the observations are considerably more likely for improbable prior parameters. To demonstrate this we consider a scenario of a similar shape to the previous stories. An astronomer has invented a telescopic device which she is using to measure the distance between two stars, which are in fact precisely 1 light-year apart.

(i) The device is unreliable and breaks down every hour on average.
(ii) Every 2.89 hours that she uses the device, she is able to double the precision (inverse variance) of her measurement; the initial precision is $6.3\,\mathrm{ly}^{-2}$. At the point that the machine breaks down, she estimates that the distance is 1 light-year – coinciding with the true distance.
(iii) How long was the scientist using the machine for?

The story is set up so that the likelihood is inverse to the prior. The numbers have been chosen so that the initial precision (6.3) is approximately 2π, and the precision doubles every $\frac{2}{\ln 2}$ hours (≈ 2.9), so that the precision τ_t at time

t is approximately $\tau_t = 2\pi e^{2t}$. If we model the measurement inaccuracy by a normal distribution, the likelihood function of data d is $\sqrt{\frac{\tau_t}{2\pi}} e^{-\frac{1}{2}(d-1)^2\tau_t}$. When $d = 1$, the likelihood is e^t. So the prior density is e^{-t}, but the likelihood is e^t.

In statistical notation:

(i) Prior: $t \sim \text{Exponential}(1)$;
(ii) Likelihood: $d \sim \text{Normal}(1, (2\pi e^{2t})^{-\frac{1}{2}})$, with $d = 1$;
(iii) What is the posterior probability on t?

As a probabilistic program:

```
1. normalize(
2.     let x = sample(exponential(1)) in
3.     observe 1 from normal(1, (2πe^{2t})^{-1/2});
4.     return(x))
```

In the Posterior Calculation Method 3, the problem is that we are very unlikely to pick long times, but when we do they receive very high weights.

In the Calculation Method 1, the unnormalized posterior density is

$$\begin{array}{rcccc} \text{Posterior} & \propto & \text{Likelihood} & \times & \text{Prior} \\ \text{posterior-pdf}(t) & \propto & e^t & \times & e^{-t} \end{array}$$

and so the probability that the time lies in a set U is

$$\int_U e^t e^{-t} \, \mathrm{d}t \;=\; \int_U 1 \, \mathrm{d}t \tag{3}$$

which is the Lebesgue measure. For instance, on an interval (a, b), the unnormalized posterior is $b - a$. Across the entire positive reals $(0, \infty)$, the normalizing constant is infinite. So the question does not have an answer. We cannot form a posterior probability on the time that the scientist used the device: every time is equiprobable.

There are several contrivances in the story, the most ridiculous of which is that the observed distance happens to perfectly match the true distance. If the observed distance had been even slightly different from the true distance, the infinite normalization constant would not occur. Indeed, if the observed distance was very different from the true distance, we could easily conclude that the device broke quickly (see Fig. 6). This means that in practice we do not need to worry about the story, because a problem-causing observation almost never occurs. In principle, however, we do need to consider infinite measures like this, in part because they can legitimately arise as fragments of

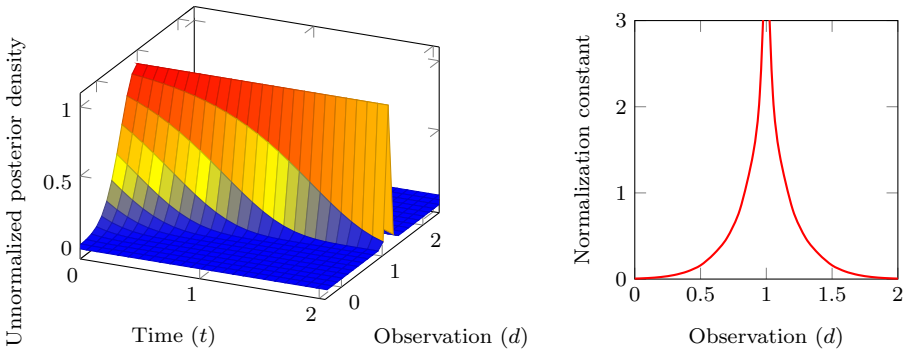

Figure 6 The posterior distribution on time spent using the device t given the observation d, in the context of the story about the scientist measuring the distance between stars. Notice that when $d = 1$ the unnormalized posterior density is constant, and the normalization constant is infinite.

reasonable programs, as we now discuss. (For further examples of improper posteriors such as this, see e.g. Robert, 2007, Ex. 1.49–1.52.)

Improper priors and posteriors

When a normalizing constant is infinite, this is sometimes called an 'improper' distribution. Although an improper distribution is problematic as the end result of an inference problem, the distributions are incredibly useful when used as part of a model. To analyze this we consider a construction $\mathsf{score}(r)$ which weights the current run by r. This is equivalent to $\mathsf{observe}\,0\,\mathsf{from}\,\mathit{exponential}(r)$.

Suppose for a moment that we have a program *Lebesgue*, such as Lines 2-4 of our astronomy example, that behaves as the Lebesgue measure. Suppose too that we have a probability distribution on $[0,\infty)$ that has a probability density function $f\colon [0,\infty) \to [0,\infty)$, and we want to sample from it. We can do this by:

$$\mathsf{let}\, x = \mathit{Lebesgue}\,\mathsf{in}\,\mathsf{score}(f(x)); \mathsf{return}(x)$$

since this is the definition of density functions. This composite program has normalizing constant 1. In fact, when we expand the definition of *Lebesgue* as above, this becomes the "importance sampling algorithm":

$$\mathsf{let}\, x = \mathsf{sample}(\mathit{exponential}(1))\,\mathsf{in}\,\mathsf{score}(e^x); \mathsf{score}(f(x)); \mathsf{return}(x)$$

In words: to build a sampler for one distribution from a sampler for another distribution, sample from the first distribution and then weight each run by the ratio of the density functions.

So although infinite normalizing constants are problematic at the top level,

it is often useful to reason about programs where subexpressions do have infinite normalizing constants.

2.5 Summary of informal semantics

We have discussed three approaches to semantics for probabilistic programs:

- mathematical semantics defined using densities and measures;
- Monte Carlo semantics with rejection;
- Monte Carlo semantics with weighting.

In Section 2.4, we have seen that, no matter what approach is taken, some care is needed because the normalizing constant may be infinite or zero.

3 Introduction to measurability issues

In Section 4 we will give a formal semantics for probabilistic programs in terms of measures. In this section, we introduce the basics of a measure-theoretic approach to probability (see also Pollard, 2002) and use it to illustrate why such a formal semantics is not entirely trivial.

The idea of weighted simulation already gives us an interpretation of a probabilistic program. We define an underlying probability space $\Omega = [0,1]^d$ where d is the number of sample statements in the program. If the program includes recursion, d may be countably infinite, but that is not a problem. We can think of each element of Ω as a list of random seeds. Given such a list, we can execute a program deterministically, leading to a weight (the product of all the observes) and a deterministic result, because the results of the sample statements are fixed.

(Here we are using the fact that uniform random numbers in $[0,1]$ are a sufficient seed for sampling from any probability distribution with parameters. For example, sampling from a Bernoulli distribution can be simulated by testing the position of a uniform random number,

$$\mathsf{sample}(\mathit{bernoulli}(r)) \;=\; \mathsf{let}\, x = \mathsf{sample}(\mathit{uniform})\, \mathsf{in}\, \mathsf{return}(x < r)$$

and more generally, sampling from a general distribution can be simulated using the inverse-cdf method, e.g.:

$$\mathsf{sample}(\mathit{normal}(m,s)) \;= \\ \mathsf{let}\, x = \mathsf{sample}(\mathit{uniform})\, \mathsf{in}\, \mathsf{return}(\mathsf{norm\text{-}invcdf}(m,s,x)).)$$

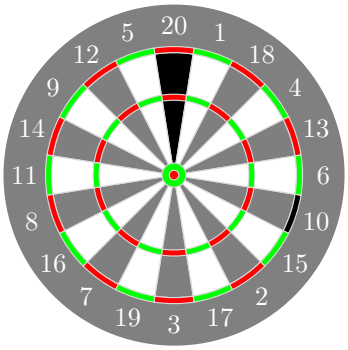

Figure 7 A dartboard with the areas scoring 20 highlighted in black.

Thus a probabilistic program of type X determines two functions:

$$\mathit{result} : \Omega \to X \qquad \mathit{weight} : \Omega \to [0, \infty) \tag{4}$$

and each run of the weighted simulation corresponds to randomly picking seeds $\omega \in \Omega$ and returning the pair $(\mathit{weight}(\omega), \mathit{result}(\omega))$.

In general this (4) is a very intensional representation of a probabilistic program: programs that describe the same probabilistic scenarios have different different representations, because the functions *result* and *weight* will differ. For example, the following two programs implementing $\mathsf{sample}(\mathit{bernoulli}2/7)$:

$$\mathsf{let}\, x = \mathsf{sample}(\mathit{uniform})\, \mathsf{in}\, \mathsf{return}(x < \tfrac{2}{7})$$
$$\mathsf{let}\, x = \mathsf{sample}(\mathit{uniform})\, \mathsf{in}\, \mathsf{return}(x > \tfrac{5}{7})$$

will have different representations; introducing redundant sample statements will give different representations; and so on. What we ultimately care about is the posterior probability on the results. In general, this will be a measure.

Measure theory generalizes the ideas of size and probability distribution from countable discrete sets to uncountable sets. To motivate, think of the game of darts. No matter how good a player I am, the chance of hitting the point at the centre of the dartboard is zero. The chance of hitting any given point is zero. Nonetheless I will hit a point when I throw. We resolve this apparent paradox by giving a probability of hitting each region. The probability of scoring 20 points is the sum of the probabilities of hitting one of the three regions that score 20 points (Figure 7). And so on. We can think of these regions of the dartboard as measurable sets with positive probability.

With this in mind, we are interested in the posterior probability that the

result of a probabilistic program is within a certain set; for example, that the day is a weekend day, or that the time is between 4am and 7am, or that I scored 20 on the dartboard. If we run a weighted simulation k times, picking seeds $\omega_1 \dots \omega_k \in \Omega$, we obtain an empirical posterior probability that the result is in the set U:

$$\frac{\sum_{i=1}^{k} [\mathit{result}(\omega_i) \in U] \cdot \mathit{weight}(\omega_i)}{\sum_{i=1}^{k} \mathit{weight}(\omega_i)} \tag{5}$$

(Here and elsewhere we regard a property, e.g. $[x \in U]$, as its characteristic function $X \to \{0,1\}$.) Although this empirical probability is itself random, in that it depends on the choices ω_i, we would like to use the law of large numbers to understand that as $k \to \infty$ the empirical posterior (5) converges to a true posterior

$$\frac{\int_\Omega [\mathit{result}(\omega) \in U] \cdot \mathit{weight}(\omega)\, \mathrm{d}\omega}{\int_\Omega \mathit{weight}(\omega)\, \mathrm{d}\omega}. \tag{6}$$

Then two programs should be regarded as the same if they give the same posterior probability measure. There are two issues:

- We need to understand why the integrals in (6) exist;
- We need to also understand program fragments in this way, so that we can reason about program equality bit by bit, compositionally.

To address these, we interpret probabilistic programs as unnormalized measures and kernels.

3.1 Rudiments of measure-theoretic probability

We recall some basic definitions of measure theory. These are well-motivated by the illustration in Figure 7: the probability of scoring 20 is the sum of the probabilities of hitting the three regions shown. Thus countable disjoint unions are crucial for formulating measures.

Definition 1.1 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, we equip the set $\mathbb{R}$ of reals with the Borel sets. The Borel sets are the smallest σ-algebra on $\mathbb{R}$ that contains the intervals. The plane $\mathbb{R}^2$ is equipped with the least σ-algebra containing the rectangles $(U \times V)$ with U and V Borel. For example, the dartboard (Fig 7) is a subset of $\mathbb{R}^2$, and the set of points that would score 20 points is measurable.

Definition 1.2 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 is σ-additive, 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 determined by saying that the measure of a line segment is its length ($\lambda(a, b) = b-a$), and the Lebesgue measure on $\mathbb{R}^2$ is determined by saying that the measure of a rectangle is its area. For any $x \in X$, the Dirac measure δ_x has $\delta_x(U) = [x \in U]$. 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 $x \in X$.

Measures can be equivalently understood as integration operators. A function between measurable spaces, $f\colon X \to Y$, is said to be measurable if $f^{-1}(U) \in \Sigma_X$ when $U \in \Sigma_Y$. If $f\colon X \to [0, \infty]$ is measurable and μ is a measure on X then we can integrate f with respect to μ, written $\int_\mu f(x)\,\mathrm{d}x$, giving a number in $[0, \infty]$.

3.2 Relationship to Bayesian statistics

The measure-theoretic semantics that we discuss in this chapter is inspired by Bayes' law, but it is not tied to it. Indeed, sometimes a language for weighted Monte Carlo simulation is useful without a formal Bayesian intuition; for example, one might use weights coming from image similarity without making a formal connection to likelihood. Nonetheless in this section we make a connection with the measure-theoretic treatment of Bayes' law.

Measures are closely related to density functions.

Definition 1.3 If $f\colon X \to [0, \infty]$ is measurable, and μ is a measure on X, then

$$\nu(U) = \int_\mu [x \in U] f(x)\,\mathrm{d}x$$

is also a measure. We say that ν has *density* f with respect to μ. A density is sometimes called a Radon-Nikodym derivative. If $\nu(X) = 1$, it is a *probability density*. If a measurable function $f : X \times Y \to [0, \infty]$ has the property that $\int_\mu f(x, y)\,\mathrm{d}x = 1$ for all y then it is a *conditional probability density* with respect to μ.

For example, the density function of the exponential distribution $(r, x) \mapsto$

re^{-x} is a conditional density with respect to the Lebesgue measure, and this induces the exponential probability measures on $\mathbb{R}$. The Dirac measure has no density with respect to the Lebesgue measure, but it does have a density with respect to itself, as does every measure.

Throughout the above analysis, we have used densities as weights. The observed data has been fixed in our examples, for example, 4 buses or 15 minutes, but it would be reasonable to make the function $\mathit{weight}\colon \Omega \to [0,\infty)$ parametrized in the data. Thus, supposing our data lies in a space D, the data-parameterized weight function is a measurable function $\mathit{likelihood}\colon D \times \Omega \to [0,\infty)$, such that $\mathit{weight}(\omega) = \mathit{likelihood}(d,\omega)$ where d is the specific data that is hard-coded into the program. The Bayesian approach is that *likelihood* should be a conditional probability density with respect to some measure λ on D.

The posterior (6) can then be made a measurable function of $y \in D$, i.e. a regular conditional probability:

$$q_y(U) = \frac{\int_\Omega [\mathit{result}(\omega) \in U] \cdot \mathit{likelihood}(y,\omega)\,\mathrm{d}\omega}{\int_\Omega \mathit{likelihood}(y,\omega)\,\mathrm{d}\omega}.$$

This can also now be connected formally to Bayes' theorem of conditional probability, see e.g. Schervish (1995, Thm. 1.31). In Section 2.4 we discussed the point that although the denominator may be 0 or ∞, for a whole program, this almost-never happens. This can now be made precise:

$$\gamma(U_{0,\infty}) = 0$$

where $\gamma(V) = \int_D \int_\Omega [y \in V] \cdot \mathit{likelihood}(y,\omega)\,\mathrm{d}\omega\,\mathrm{d}y$ is the prior predictive measure, and $U_{0,\infty} = \{y \mid \int_\Omega \mathit{likelihood}(y,\omega)\,\mathrm{d}\omega \in \{0,\infty\}\}$.

We conclude by mentioning, as an aside, that in complex situations, the Bayesian requirement of a single base measure λ on D can be subtle. The density functions for the GPA problem in Figure 3 are densities with respect to the mixed measure $(\mathit{lebesgue} + \delta_4 + \delta_{10})$. The theory of conditional probability densities requires a single common base measure for all the different parameters. The following program will only give the right result if we use the same base measure $(\mathit{lebesgue} + \delta_4 + \delta_{10})$ on $\mathbb{R}$ for the likelihood functions for all the different if-then-else branches.

$\mathsf{let}\ american = \mathsf{sample}(bernoulli(0.5))\ \mathsf{in}$
$\mathsf{let}\ brilliant = \mathsf{sample}(bernoulli(0.01))\ \mathsf{in}$
$\mathsf{if}\ american\ \mathsf{then}$
  $\mathsf{if}\ brilliant\ \mathsf{then}\ \mathsf{observe}\ 4\ \mathsf{from}\ dirac(4)\ \mathsf{else}\ \mathsf{observe}\ 4\ \mathsf{from}\ uniform(0,4)$
else
  $\mathsf{if}\ brilliant\ \mathsf{then}\ \mathsf{observe}\ 4\ \mathsf{from}\ dirac(10)\ \mathsf{else}\ \mathsf{observe}\ 4\ \mathsf{from}\ uniform(0,10)$
$\mathsf{return}(american)$

This is subtle because the density of the Indian distribution $uniform(0, 10)$ with respect to the base *lebesgue* measure is the constant 0.1 function, but the density of $uniform(0, 10)$ with respect to the base measure $(lebesgue + \delta_4 + \delta_{10})$ must take value 0 at 4, as in Figure 3. Overall, then, the program is a Dirac measure at $american = \mathsf{true}$.

In summary, the meaning of a closed probabilistic program is an unnormalized measure, thought of as the nominator in Bayes' rule. For a program expression that has free variables, its interpreation should be measurable in the valuation of those variables.

- Sampling from a probability measure is a measure.
- An observation $\mathsf{observe}\ x\ \mathsf{from}\ d$ is a one point measure whose value is the density of d at x.
- The sequencing $\mathsf{let}\ x = t\ \mathsf{in}\ u$ means, roughly, integration: $\int_t u\,\mathrm{d}x$.
- The simple statement $\mathsf{return}(t)$ means the Dirac delta measure.

We make this precise in Section 4.

3.3 Obstacles to measurability

We now illustrate why measurability of programs is not entirely trivial. Our counterexamples are based on the counting measure on the real numbers. This is an unnormalized distribution that assigns 1 to every singleton set. It turns out that although some infinite measures are definable in a probabilistic programming language, the counting measure on $\mathbb{R}$ is not definable – we show this in Section 5.2. But for now let us suppose that we add it to our language, as a command *counting*, and see what chaos ensues. (For now, we retain an intuitive view of measurability; precise definitions are in Section 4, with a precise version of the arguments in this section given in Section 5.2.)

As before, for any set U we can consider a function $[x \in U]$ which returns true if $x \in U$ and false otherwise. For example, we might write $[x \in \{0,1,2,3\}]$, $[x > 0]$, $[x = 42]$, and so on. The following lemma gives some intuition for the counting measure.

Lemma 1.4 *For any (measurable) set U, the program*

$$\mathsf{let}\, r = \mathit{counting}\, \mathsf{in}\, \mathsf{return}[r \in U]$$

gives weight $\#U$ to true *and $\#(\mathbb{R} \setminus U)$ to* false*, where $\#U$ is the cardinality of U if U is finite, or ∞ otherwise.*

In this extended language, the fundamental law of exchangeability is violated: the order of draws matters, as we now explain. Notice that $\mathsf{let}\, s = \mathit{counting}\, \mathsf{in}\, \mathsf{return}[r = s]$ has the same semantics as $\mathsf{return}(\mathsf{true})$, for all r, because there is exactly one s that is equal to any given r (Lemma 1.4). So

$$\mathsf{let}\, r = \mathit{uniform}(0,1)\, \mathsf{in}\, \mathsf{let}\, s = \mathit{counting}\, \mathsf{in}\, \mathsf{return}[r = s] \tag{7}$$

is an equivalent program to $\mathsf{return}(\mathsf{true})$. But

$$\mathsf{let}\, r = \mathit{uniform}(0,1)\, \mathsf{in}\, \mathsf{return}[r = s]$$

has the same semantics $\mathsf{return}(\mathsf{false})$, for all s, because any r is almost surely different from a given s. So

$$\mathsf{let}\, s = \mathit{counting}\, \mathsf{in}\, \mathsf{let}\, r = \mathit{uniform}(0,1)\, \mathsf{in}\, \mathsf{return}[r = s] \tag{8}$$

has the same semantics as $\mathsf{return}(\mathsf{false})$. Comparing (8) to (7), we see that programs involving the counting measure cannot be reordered.

In fact, the measure-theoretic semantics of the language extended with *counting* is not always even fully defined. For an example of this, we recall that there exist Borel-measurable subsets U of the plane $\mathbb{R}^2$ for which the projection $\pi[U] \stackrel{\text{def}}{=} \{x \mid \exists y.\, (x,y) \in U\}$ is not Borel-measurable in $\mathbb{R}$. (In general $\pi[U]$ is called 'analytic'.) Now the program

$$\mathsf{let}\, s = \mathit{counting}\, \mathsf{in}\, \mathsf{return}[(r,s) \in U]$$

puts a non-zero weight on true if and only if $r \in \pi[U]$. So this program is not measurable in r, and so programs built from it, such as

$$\mathsf{let}\, r = \mathit{uniform}(0,1)\, \mathsf{in}\, \mathsf{let}\, s = \mathit{counting}\, \mathsf{in}\, \mathsf{return}[(r,s) \in U]$$

are not well defined.

As we will see in Section 4 (Lemma 1.7), this problem cannot arise in the language without the *counting* measure: every term is compositionally well-behaved.

4 Formal semantics of probabilistic programs as measures

We now turn to give a precise semantics of probabilistic programs. To this end we set up a typed language with a precise syntax.

In the previous section we have considered programs as Bayesian statistical models. However, this is only an intuition, and the semantics is given in terms of weighted simulations and measure theory. Moreover, some applications of weighted simulation are beyond the realms of Bayesian statistics.

For these reasons, the precise language that we now consider will have the keyword $\mathsf{score}(r)$, which weights the run by r, instead of the keyword $\mathsf{observe}$. The two are inter-definable:

$$\mathsf{observe}\, r \,\mathsf{from}\, p = \mathsf{score}(f(r)), \quad \text{where } f \text{ is the density of } p$$
$$\mathsf{score}(r) = \mathsf{observe}\, 0 \,\mathsf{from}\, \mathit{exponential}(r)$$

4.1 Types

In what follows it is helpful to consider a typed programming language. We will consider types such as natural numbers, real numbers, tuples of real numbers, and lists of real numbers. In practice many probabilistic programming languages do not perform type checking, but having a type greatly simplifies the mathematical semantics. Moreover, types play an intuitive role, because a probabilistic program may describe a measure on the space of natural numbers, or the space of real numbers, or on the real plane. With this intuition, a type is just a syntactic description of a space. For instance, we can understand an expression of real type as a measure on the real line; an expression of integer type as a measure on the space of integers, and so on.

Our types are generated by the following grammar:

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

where I ranges over countable, non-empty sets. The type $\coprod_{i \in I} \mathbb{A}_i$ is sometimes called a labelled variant or a tagged union. The type $\mathsf{P}(\mathbb{A})$ is a type of distributions on $\mathbb{A}$. Here are some examples of types in the grammar:

- The type $\mathbb{R}$ of the real line, and type $\mathbb{R} \times \mathbb{R}$ of the plane;
- The type $(1 + 1)$ of booleans (true/false), the type $\coprod_{i \in \mathbb{N}} 1$ of natural numbers;
- The type $\coprod_{i \in \mathbb{N}} \mathbb{R}^i$ of sequences of reals of arbitrary length;
- The type $\mathsf{P}(1 + 1)$ of probability distributions over the booleans, and the type $\mathsf{P}(\mathbb{R})$ of probability distributions on the reals.

To keep things simple we do include function types such as $(\mathbb{R} \to \mathbb{R})$ and $(\mathbb{R} \to \mathbb{R}) \to \mathbb{R}$. Also, this is not a type system that can be automatically checked in a computer because we include infinite sum types rather than recursion schemes. We do this 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. Ackerman et al. (2011)).

4.2 Types as measurable spaces

Types $\mathbb{A}$ are interpreted as measurable spaces $[\![\mathbb{A}]\!]$, by induction on their structure, as follows. To be precise we distinguish between the syntactic name of the type $\mathbb{A}$ and the space $[\![\mathbb{A}]\!]$ which interprets it.

- $[\![\mathbb{R}]\!]$ is the measurable space of reals, with its Borel sets. 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.
- $[\![1]\!]$ is the unique 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 the least one containing the rectangles $(U \times V)$ with $U \in \Sigma_{[\![\mathbb{A}]\!]}$ and $V \in \Sigma_{[\![\mathbb{B}]\!]}$ (e.g. Pollard, 2002, Def. 16)).
- $[\![\coprod_{i\in I} \mathbb{A}_i]\!]$ is the coproduct space $\biguplus_{i\in I} [\![\mathbb{A}_i]\!]$, the disjoint union. The σ-algebra $\Sigma_{[\![\coprod_{i\in I}\mathbb{A}_i]\!]}$ is least one containing the sets $\{(i,a) \mid a \in U\}$ for $U \in \Sigma_{[\![\mathbb{A}_i]\!]}$. For example, the type $\mathbb{N}$ is interpreted as the space $[\![\mathbb{N}]\!]$ of natural numbers with the discrete σ-algebra, where all sets are measurable.
- We let $[\![\mathsf{P}(\mathbb{A})]\!]$ be the set $P([\![\mathbb{A}]\!])$ of probability measures on $[\![\mathbb{A}]\!]$ together with the least σ-algebra containing the sets $\{\mu \mid \mu(U) < r\}$ for each $U \in \Sigma_X$ and $r \in [0,1]$ (the 'Giry monad' (Giry, 1982)).

4.3 Typed program expressions

We consider programs built from the following grammar:

$$\begin{aligned} t, t_0, t_1 \ ::= {} & (i,t) \mid \mathsf{case}\ t\ \mathsf{of}\ \{(i,x) \Rightarrow u_i\}_{i\in I} \mid () \mid (t_0,t_1) \mid \mathrm{proj}_j(t) \mid f(t) \mid x \\ & \mid \mathsf{return}(t) \mid \mathsf{let}\, x = t\, \mathsf{in}\, u \mid \mathsf{sample}(t) \mid \mathsf{score}(t) \mid \mathsf{normalize}(t) \end{aligned} \tag{9}$$

The first line of (9) contains standard deterministic expressions, for example destructing union and product types, with intended equations such as the following:

$$\Big(\mathsf{case}\ (j,t)\ \mathsf{of}\ \{(i,x) \Rightarrow u_i\}\Big) \ = \ u_j[t/x] \qquad \mathrm{proj}_j(t_0,t_1) = t_j.$$

We also include some basic functions f, and in fact, we may as well include all measurable functions in our language, including arithmetic operations and constants (e.g. $+$, $\times$, k_{10}), comparison predicates (e.g. $=$, $<$), and parameterized probability measures (e.g. *normal*, *bernoulli*). There are also variables x that are bound by case and let.

In a real computer language, operations over infinite structures such as lists and numbers are given by induction or recursion. In this chapter, rather than worry about this, we simply allow the programmer to give a different case for every index into the infinite structure. This means that the case syntax is potentially infinite, since the set I might be (countably) infinite. It is routine to build a finite language with inductive primitives and translate it into this one.

The second line of (9) contains ways of combining programs (let) and sequencing, as well as the three crucial primitives of probabilistic programming: sample, score and $\mathsf{normalize}$.

In this simple language, there is little syntactic sugar, and so the program about buses in Section 2.1 would be written:

$$
\begin{array}{ll}
1. & \mathsf{normalize}(\\
2. & \quad \mathsf{let}\, x = \mathsf{sample}(\mathit{bernoulli}(\tfrac{2}{7}))\, \mathsf{in} \\
3. & \quad \mathsf{let}\, r = \mathsf{case}\, x\, \mathsf{of}\, \{(1, \_) \Rightarrow \mathsf{return}(k_3()) \,, (2, \_) \Rightarrow \mathsf{return}(k_{10}())\}\, \mathsf{in} \\
4. & \quad \mathsf{let}\, \_ = \mathsf{score}(\tfrac{1}{4!} r^4 e^{-r})\, \mathsf{in} \\
5. & \quad \mathsf{return}(x))
\end{array}
\tag{10}
$$

where $k_3, k_{10} : 1 \to \mathbb{R}$ are the obvious constant functions, which are measurable.

Typed 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. Here the context Γ is of the form $(x_1 \colon \mathbb{B}_1, \ldots, x_n \colon \mathbb{B}_n)$. The intuition is that if $\Gamma \vdash_{\mathsf{z}} t \colon \mathbb{A}$ then the free variables of t are contained in $x_1 \ldots x_n$, and given values of the right type for each free variable, then the expression t will return something of type $\mathbb{A}$, either deterministically or probabilistically. For example, the entire program in (10) is a deterministic term returning a distribution, whereas lines 2–5 form a probabilistic term of type $(1+1)$. Neither have any free variables. The term $\mathsf{score}(\frac{1}{4!} r^4 e^{-r})$ is a probabilistic term with a real free variable $r \colon \mathsf{real}$, so we write $r \colon \mathsf{real} \vdash_{\mathsf{p}} \mathsf{score}(\frac{1}{4!} r^4 e^{-r}) \colon 1$.

We have already explained that each type $\mathbb{A}$ is understood as a measurable space. Formally, a context $\Gamma = (x_1 \colon \mathbb{A}_1, \ldots, x_n \colon \mathbb{A}_n)$ is also interpreted as a measurable space $[\![\Gamma]\!] \stackrel{\text{def}}{=} \prod_{i=1}^{n} [\![\mathbb{A}_i]\!]$ of well-typed valuations for the variables.

As will be seen in the next section, deterministic terms $\Gamma \vdash_{\mathsf{d}} t \colon \mathbb{A}$ denote measurable functions from $[\![\Gamma]\!] \to [\![\mathbb{A}]\!]$, closed probabilistic terms $\vdash_{\mathsf{p}} t' \colon \mathbb{A}$ denote measures on $[\![\mathbb{A}]\!]$, and open 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 Section 4.4.

We specify the valid judgements $\Gamma \vdash_{\mathsf{d}} t \colon \mathbb{A}$ and $\Gamma \vdash_{\mathsf{p}} t \colon \mathbb{A}$ as the least relations closed under the following rules.

Sums and products. The type system allows variables, and standard constructors and destructors for sum and product types.

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

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

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

If the reader is not familiar with type systems, they might consult the early chapters of Harper (2016). We give an example of a typing derivation later, in (12). For instance, the rule for (t_0, t_1) says that "if term t_0 has type $\mathbb{A}_0$ and term t_1 has type $\mathbb{A}_1$ then the pair (t_0, t_1) has type $(\mathbb{A}_0 \times \mathbb{A}_1)$".

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. The continuations of case expressions may be either deterministic or probabilistic, as indicated.

Sequencing. We include the standard constructs for sequencing (e.g. Levy et al., 2003; Moggi, 1991).

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

Notice that, in this simple language, everything probabilistic must be explicitly sequenced. For example, if $\Gamma \vdash_{\mathsf{p}} t_0 \colon \mathbb{A}_0$ and $\Gamma \vdash_{\mathsf{p}} t_1 \colon \mathbb{A}_1$, we cannot conclude that $\Gamma \vdash_{\mathsf{p}} (t_0, t_1) \colon \mathbb{A}_0 \times \mathbb{A}_1$. Rather, we have to explicitly write

$$\begin{aligned} &\Gamma \vdash_{\mathsf{p}} \mathsf{let}\, x_0 = t_0\, \mathsf{in}\, \mathsf{let}\, x_1 = t_1\, \mathsf{in}\, \mathsf{return}(x_0, x_1) \colon \mathbb{A}_0 \times \mathbb{A}_1 \\ \text{or} \quad &\Gamma \vdash_{\mathsf{p}} \mathsf{let}\, x_1 = t_1\, \mathsf{in}\, \mathsf{let}\, x_0 = t_0\, \mathsf{in}\, \mathsf{return}(x_0, x_1) \colon \mathbb{A}_0 \times \mathbb{A}_1 \end{aligned}$$

Later (§5.1) we will show that the order of evaluation doesn't matter, so we

could use (t_0, t_1) as an unambiguous syntactic sugar, but it makes the formal semantics simpler to insist that the order of evaluation is given explicitly.

Language-specific constructs. We also include constant terms for all measurable functions. Recall that a function $f\colon X \to Y$ between measurable spaces is itself measurable if the inverse image of a measurable set is again measurable.

$$\frac{\Gamma \vdash_{\mathsf{d}} t\colon \mathbb{A}}{\Gamma \vdash_{\mathsf{d}} f(t)\colon \mathbb{B}}\ (f\colon [\![\mathbb{A}]\!] \to [\![\mathbb{B}]\!] \text{ measurable}) \tag{11}$$

Thus we assign suitable types to the arithmetic operations and constants (e.g. $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, $k_{10} : 1 \to \mathbb{R}$), predicates (e.g. $(=) : \mathbb{R} \times \mathbb{R} \to \mathsf{bool}$) and probability measures (e.g. $\mathit{normal} : \mathbb{R} \times \mathbb{R} \to P(\mathbb{R})$). For instance, we have a judgement $\mu : \mathbb{R}, \sigma : \mathbb{R} \vdash_{\mathsf{d}} \mathit{normal}(\mu, \sigma)\colon \mathsf{P}(\mathbb{R})$. (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.)

For example, the expression $(\mathsf{if}\, x\, \mathsf{then}\, 3\, \mathsf{else}\, 10)$ is shorthand for

$$(\mathsf{case}\, x\, \mathsf{of}\, \{(1, \_) \Rightarrow k_3()\ ;\ (2, \_) \Rightarrow k_{10}()\})$$

We derive that the expression has type $\mathbb{R}$ when x has type bool, by deriving it from the rules as follows.

$$\frac{\dfrac{\dfrac{-}{x\colon \mathsf{bool} \vdash_{\mathsf{d}} x\colon \mathsf{bool}} \quad \dfrac{\dfrac{-}{x\colon \mathsf{bool}, z\colon 1 \vdash_{\mathsf{d}} ()\colon 1}}{x\colon \mathsf{bool}, z\colon 1 \vdash_{\mathsf{d}} k_3()\colon \mathbb{R}} \quad \dfrac{\dfrac{-}{x\colon \mathsf{bool}, z\colon 1 \vdash_{\mathsf{d}} ()\colon 1}}{x\colon \mathsf{bool}, z\colon 1 \vdash_{\mathsf{d}} k_{10}()\colon \mathbb{R}}}{x\colon \mathsf{bool} \vdash_{\mathsf{d}} \mathsf{case}\, x\, \mathsf{of}\, \{(1, z) \Rightarrow k_3()\ ;\ (2, z) \Rightarrow k_{10}()\}\colon \mathbb{R}} \tag{12}$$

The core of the language is the constructs corresponding to the terms in Bayes' law: 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.

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

As we discussed in Section 2.4, normalization will fail if the normalizing constant is zero or infinity; so it produces either a normalization constant together with a normalized posterior distribution ($\mathbb{R} \times \mathsf{P}(\mathbb{A})$), or exceptionally one of the two failure possibilities ($+1 + 1$). In a complex model the

normalized posterior could subsequently be used as a prior and sampled from. This is sometimes called a 'nested query' (see for instance Stuhlmuller and Goodman, 2014), but it remains to be seen whether it is computationally practical (Rainforth et al., 2018).

4.4 Expressions as s-finite kernels, programs as measures

In this section we will give an interpretation of closed programs $\vdash_{\mathsf{p}} t\colon \mathbb{A}$ as measures on $\mathbb{A}$. To do this, we must also interpret open programs $\Gamma \vdash_{\mathsf{p}} t\colon \mathbb{A}$, which will be families of measures on $[\![\mathbb{A}]\!]$ that are indexed by the valuations of the context $[\![\Gamma]\!]$. These are called *kernels*. (Warning: the word kernel is over-used and has other meanings.)

s-Finite kernels

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 $\int_{k(x)} f(y)\,\mathrm{d}y \in [0,\infty]$. 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 $x \in X$.

We need to further refine the notion of kernels, because arbitrary kernels do not behave well. The following result is a step towards the central notion of *s-finite* kernel.

Proposition 1.5 *Let X, Y be measurable spaces. If $k_1 \dots k_n \cdots : X \rightsquigarrow Y$ are kernels then the function $(\sum_{i=1}^{\infty} k_i) : X \times \Sigma_Y \to [0,\infty]$ given by*

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

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

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

Definition 1.6 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.

If the reader is familiar with the notion of σ-finite measure, they will

note that this is different. In fact, an s-finite measure is the same thing as the push-forward of a σ-finite measure (Getoor, 1990; Sharpe, 1988). The definition of s-finite kernel is not so common but appears in recent work by Kallenberg (2014) and Last and Penrose (2016, App. A). It was proposed as a foundation for probabilistic programming by the author (Staton, 2017), but it has since attracted further use and development (e.g. Bichsel et al., 2018; Ong and Vákár, 2018).

Composition of kernels

Before we give the semantics of our language, we need a lemma which is central to the interpretation of let.

Lemma 1.7 *Let X, Y, Z be measurable spaces, and let $k : X \times Y \rightsquigarrow Z$ and $l : X \rightsquigarrow Y$ be s-finite kernels (Def. 1.6). Then we can define a s-finite kernel $(k \star l) : X \rightsquigarrow Z$ by*

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

so that

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

A proof is given in (Staton, 2017), building on a well-known fact that the the property holds for finite kernels (e.g. Pollard, 2002, Thm. 20(ii)). The example in Section 3.3 shows that if we generalize to arbitrary kernels, we cannot construct $k \star l$ in general. In detail, let $X = Y = \mathbb{R}$ and let $Z = 1 = \{*\}$. Pick a Borel subset $U \subseteq \mathbb{R} \times \mathbb{R}$ whose projection is not Borel. Let $k(x, y, \{*\}) = [(x, y) \in U]$, and let $l(x, -)$ be the counting measure on $\mathbb{R}$. Then $(k \star l)(x, \{*\})$ is non-zero if and only if $x \in \pi[U]$, and so it is not measurable in x, and so it is not a kernel.

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 of the language, beginning with variables, sums and products, is roughly the same as a set-theoretic semantics. For each typed term $\Gamma \vdash_{\mathsf{d}} t \colon \mathbb{A}$, and each valuation $\gamma \in [\![\Gamma]\!]$ of values for variables, we define an element $[\![t]\!]_\gamma$ of $\mathbb{A}$, in such a way that the assignment is measurable in γ. We

do this by induction on the structure of typing derivations:

$$\llbracket x \rrbracket_\gamma \stackrel{\text{def}}{=} \gamma_x \qquad \llbracket (i,t) \rrbracket_\gamma \stackrel{\text{def}}{=} (i, \llbracket t \rrbracket_\gamma)$$
$$\llbracket \mathsf{case}\ t\ \mathsf{of}\ \{(i,x) \Rightarrow u_i\}_{i\in I} \rrbracket_\gamma \stackrel{\text{def}}{=} \llbracket u_i \rrbracket_{\gamma,d} \quad \text{if } \llbracket t \rrbracket_\gamma = (i,d)$$
$$\llbracket () \rrbracket_\gamma \stackrel{\text{def}}{=} () \qquad \llbracket (t_0,t_1) \rrbracket_\gamma \stackrel{\text{def}}{=} (\llbracket t_0 \rrbracket_\gamma, \llbracket t_1 \rrbracket_\gamma) \qquad \llbracket \pi_j(t) \rrbracket_\gamma \stackrel{\text{def}}{=} d_i \quad \text{if } \llbracket t \rrbracket_\gamma = (d_0,d_1)$$

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

For each typed probabilistic term $\Gamma \vdash_{\mathsf{p}} t \colon \mathbb{A}$, and each valuation $\gamma \in \llbracket \Gamma \rrbracket$, and each measurable set $U \in \Sigma_{\llbracket \mathbb{A} \rrbracket}$, we define a measure $\llbracket t \rrbracket_{\gamma;U} \in [0,\infty]$, in such a way that $\llbracket t \rrbracket$ is an s-finite kernel $\llbracket \Gamma \rrbracket \rightsquigarrow \llbracket \mathbb{A} \rrbracket$ (Def. 1.6). The semantics of sequencing are perhaps the most interesting: return is the Dirac delta measure, and let is integration.

$$\llbracket \mathsf{return}(t) \rrbracket_{\gamma;U} \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } \llbracket t \rrbracket_\gamma \in U \\ 0 & \text{otherwise} \end{cases} \qquad \llbracket \mathsf{let}\, x = t\, \mathsf{in}\, u \rrbracket_{\gamma;U} \stackrel{\text{def}}{=} \int_{\llbracket t \rrbracket_\gamma} \llbracket u \rrbracket_{\gamma,x;U}\, \mathrm{d}x$$

The interpretation $\llbracket \mathsf{return}(t) \rrbracket$ is finite, hence s-finite. The fact that $\llbracket \mathsf{let}\, x = t\, \mathsf{in}\, u \rrbracket$ is an s-finite kernel is Lemma 1.7: this is the most intricate part of the semantics.

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

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

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

$$\overline{\llbracket u_i \rrbracket}_{\gamma,(j,a);U} \stackrel{\text{def}}{=} \begin{cases} \llbracket u_i \rrbracket_{\gamma,a;U} & i = j \\ 0 & \text{otherwise} \end{cases}$$

and it is easy to show that $\overline{\llbracket u_i \rrbracket}$ 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 $(\sum_i \overline{\llbracket u_i \rrbracket}) : \llbracket \Gamma \rrbracket \times \biguplus_j \llbracket \mathbb{A}_j \rrbracket \rightsquigarrow \llbracket \mathbb{B} \rrbracket$. Finally, we use a simple instance of Lemma 1.7 to compose $(\sum_i \overline{\llbracket u_i \rrbracket})$ with $\llbracket t \rrbracket : \llbracket \Gamma \rrbracket \to \biguplus_j \llbracket \mathbb{A}_j \rrbracket$ and conclude that $\llbracket \mathsf{case}\ t\ \mathsf{of}\ \{(i,x) \Rightarrow u_i\}_{i\in I} \rrbracket$ is an s-finite kernel.

The language specific constructions are straightforward.

$$\llbracket \mathsf{sample}(t) \rrbracket_{\gamma;U} \stackrel{\text{def}}{=} \llbracket t \rrbracket_\gamma(U) \qquad \llbracket \mathsf{score}(t) \rrbracket_{\gamma;U} \stackrel{\text{def}}{=} \begin{cases} |\llbracket t \rrbracket_\gamma| & \text{if } U = \{()\} \\ 0 & \text{if } U = \emptyset. \end{cases}$$

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 $k_i : [\![\Gamma]\!] \rightsquigarrow 1$

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

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 $\Gamma \vdash_{\mathsf{p}} t \colon \mathbb{A}$ and let $\mathrm{evidence}_{\gamma,t} \stackrel{\text{def}}{=} [\![t]\!]_{\gamma;[\![\mathbb{A}]\!]}$.

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

5 Reasoning with measures

Once a formal semantics of probabilistic programs as measures is given, one can reason about programs by reasoning about measures. Moreover, since the semantics is compositional, one can build up properties of programs in a compositional way. We consider two examples.

5.1 Reasoning example: Commutativity

We can quickly verify the following law

$$\begin{aligned} & \mathsf{let}\, x_0 = t_0 \,\mathsf{in}\, \mathsf{let}\, x_1 = t_1 \,\mathsf{in}\, \mathsf{return}(x_0, x_1) \\ = \quad & \mathsf{let}\, x_1 = t_1 \,\mathsf{in}\, \mathsf{let}\, x_0 = t_0 \,\mathsf{in}\, \mathsf{return}(x_0, x_1) \end{aligned} \tag{13}$$

whenever $\Gamma \vdash_{\mathsf{p}} t_0 \colon \mathbb{A}_0$ and $\Gamma \vdash_{\mathsf{p}} t_1 \colon \mathbb{A}_1$. To do this we recall that $[\![t_0]\!]_{\gamma;-}$ and $[\![t_1]\!]_{\gamma;-}$ are measures on $\mathbb{A}_0$ and $\mathbb{A}_1$ respectively, and calculate that

$$[\![\mathsf{let}\, x_0 = t_0 \,\mathsf{in}\, \mathsf{let}\, x_1 = t_1 \,\mathsf{in}\, \mathsf{return}(x_0, x_1)]\!]_{\gamma;U} = \int_{[\![t_0]\!](\gamma)} \int_{[\![t_1]\!](\gamma)} [(x_0, x_1) \in U] \,\mathrm{d}x_1 \,\mathrm{d}x_0$$

is the definition of the product measure on $\mathbb{A}_0 \times \mathbb{A}_1$. Product measures are not well-defined in general, but they are well-defined for finite measures, and this extends to s-finite measures. Indeed to conclude (13), one would notice that for any s-finite measures μ_0, μ_1 on $\mathbb{A}_0$ and $\mathbb{A}_1$, the product measures on $\mathbb{A}_0 \times \mathbb{A}_1$ are equal:

$$\int_{\mu_0}\int_{\mu_1} [(x_0, x_1) \in U] \, \mathrm{d}x_1 \, \mathrm{d}x_0 = \int_{\mu_1}\int_{\mu_0} [(x_0, x_1) \in U] \, \mathrm{d}x_0 \, \mathrm{d}x_1$$

This is known as the Fubini-Tonelli theorem, which holds for s-finite measures (e.g. Sharpe, 1988; Staton, 2017).

5.2 Reasoning example: Non-definability

We have seen in Section 3.3 that the counting measure on $\mathbb{R}$, which assigns to each set its size, is problematic for a probabilistic programming language. We now show that it is not definable. It is sufficient to show that it is not s-finite, since every definable program describes an s-finite measure. To show this we show that for every s-finite measure μ, the set $\{r \mid \mu(\{r\}) > 0\}$ is countable. The counting measure violates this invariant. Since a countable union of countable sets is countable, it suffices to show that $\{r \mid \mu(\{r\}) > 0\}$ is countable when μ is a finite measure. To see this, notice that for each positive integer n the set $\{r \mid \mu(\{r\}) > \frac{1}{n}\}$ must be finite, and so $\{r \mid \mu(\{r\}) > 0\} = \bigcup_{n \in \mathbb{Z}^+} \{r \mid \mu(\{r\}) > \frac{1}{n}\}$ must be countable.

6 Other approaches to semantics and open questions

6.1 Different approaches to semantic definitions

In other work (Staton et al., 2016) we have considered a semantics based on a monad

$$X \mapsto P([0, \infty) \times X)$$

on the category of measurable spaces. This arises from combining the writer monad for the monoid $([0, \infty), +, 0)$ of scores with the probability monad P. This naturally matches the two constructions (score for $[0, \infty)$ and sample for P), and it fits the weighted simulation semantics: the meaning of a program is a distribution over runs, each of which has a weight and a result. This semantics distinguishes things that should arguably be considered equal. For example, the semantics will distinguish

$$\mathsf{let}\, x = \mathsf{sample}(\mathit{bernoulli}(0.5))\, \mathsf{in}\, \mathsf{if}\ x\ \mathsf{then}\ \mathsf{score}(4)\ \mathsf{else}\ \mathsf{score}(6); \mathsf{return}(42)$$

from

$$\mathsf{score}(5); \mathsf{return}(42)$$

This semantics can be translated to the less discriminating semantics in this chapter as follows. Every measurable function

$$f \colon Y \to P([0,\infty) \times X)$$

can be translated to an s-finite kernel $f^\sharp \colon Y \rightsquigarrow X$ where

$$f^\sharp(y, U) \;=\; \int_{f(y)} r \cdot [x \in U] \,\mathrm{d}(r, x).$$

In fact, every s-finite kernel arises in this way. This translation preserves all the structure. Thus the monadic interpretation of the language can be translated into the s-finite semantics compositionally.

In Section 3 we considered an even more fine-grained approach, where a program $- \vdash_{\mathsf{p}} t \colon \mathbb{A}$ is interpreted as a measurable function $\Omega \to [\![\mathbb{A}]\!]$, i.e. a random variable on some basic probability space, together with a separate likelihood function $\Omega \to [0,\infty)$. (See also e.g. Holtzen et al., 2018; Hur et al., 2015). By considering the law of the pairing $\Omega \to [0,\infty) \times [\![\mathbb{A}]\!]$ we arrive at a probability measure in $P([0,\infty) \times [\![\mathbb{A}]\!])$, and every such probability measure arises as the law of some such pairing. Another way to include weightings is to consider Ω to be a subset of some plane $\mathbb{R}^n$ with an unnormalized Lebesgue measure. It turns out that an s-finite measure on a standard Borel space X is the same thing as the pushforward measure of a Lebesgue measure along a measurable function $\Omega \to X$, where $\Omega \subseteq \mathbb{R}^n$. So these different semantic methods all agree on what can be considered.

Although s-finite measures and kernels behave very well and have many characterizations, it is currently an open question whether the category of s-finite kernels is itself the Kleisli category for a monad. Recently we have proposed to use quasi-Borel spaces as generalized measurable spaces. S-finite kernels between quasi-Borel spaces do form the Kleisli category for a monad (Scibior et al., 2018).

6.2 Other semantic issues

In this chapter we have focused on giving a simple, measure-theoretic semantics to the programs in the simple first-order language through s-finite kernels. The semantics is clear, but subtle, because of issues of infinite normalization constants and measurability issues. But this simple semantics is only a very first step. Beyond:

- Statisticians and probabilists are interested in other issues such as convergence and relative entropy, which might also be analyzed in a compositional way, together with their relationships to computability (e.g. Ackerman et al., 2011; Huang and Morrisett, 2017).
- We might also add different modes of conditioning, such as conditioning by disintegration rather than density (e.g. Shan and Ramsey, 2016).
- We might add other typical language features such as higher order functions (e.g. Staton et al., 2016; Heunen et al., 2017), higher order recursion (e.g. Ehrhard et al., 2018; Vákár et al., 2019), and abstract types (e.g. Staton et al., 2018).
- Other languages have additional, non-functional primitives, based on logic programming (e.g. Nitti et al., 2016; Wu et al., 2018).

Acknowledgements.

I have benefited from discussing this topic with a great many people at various meetings over the last three years. I would particularly like to thank Chris Heunen, Ohad Kammar, Sean Moss, Matthijs Vákár, Frank Wood, Hongseok Yang. The examples in Sections 2 and 3 arose in preparing various invited talks lectures on the subject, including ICALP 2018 and OPLSS 2019, and I am grateful for these opportunities. The material in Section 4 is based on my paper in ESOP 2017 (Staton, 2017), and I am grateful to the reviewers of that article.

My research is supported by a Royal Society University Research Fellowship.

Dartboard image (Fig. 7) based on TikZ example by Roberto Bonvallet (Creative Commons attribution license), with changes made.

Bibliography

Ackerman, N L, Freer, Cameron E, and Roy, Daniel M. 2011. Noncomputable Conditional Distributions. In: *Proc. LICS 2011*.

Ackerman, Nathanael L., Freer, Cameron E., and Roy, Daniel M. 2015. *On computability and disintegration*.

Bhat, Sooraj, Borgström, Johannes, Gordon, Andrew D., and Russo, Claudio V. 2017. Deriving Probability Density Functions from Probabilistic Functional Programs. *Logical Methods in Computer Science*, **13**(2).

Bichsel, Benjamin, Gehr, Timon, and Vechev, Martin. 2018. Fine-Grained Semantics for Probabilistic Programs. In: *Proc. ESOP 2018*.

Ehrhard, Thomas, Pagani, Michele, and Tasson, Christine. 2018. Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming. In: *Proc. POPL 2018*.

Gehr, Timon, Misailovic, Sasa, and Vechev, Martin T. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. Pages 62–83 of: *Proc. CAV 2016*.

Getoor, R K. 1990. *Excessive Measures*. Birkhäuser.

Giry, M. 1982. A Categorical Approach to Probability Theory. *Categorical Aspects of Topology and Analysis*, **915**, 68–85.

Goodman, N. D., and Stuhlmüller, A. 2014. *The Design and Implementation of Probabilistic Programming Languages*.

Goodman, Noah, Mansinghka, Vikash, Roy, Daniel M, Bonawitz, Keith, and Tenenbaum, Joshua B. 2008. Church: a language for generative models. In: *UAI*.

Harper, Robert. 2016. *Practical Foundations for Programming Languages*. CUP.

Heunen, C, Kammar, O, Staton, S, and Yang, H. 2017. *A convenient category for higher-order probability theory*. arXiv:1701.02547.

Holtzen, Steven, den Broeck, Guy Van, and Millstein, Todd D. 2018. Sound Abstraction and Decomposition of Probabilistic Programs. In: *Proc. ICML 2018*.

Huang, Daniel, and Morrisett, Greg. 2017. An application of computable distributions to the semantics of probabilistic programs: part 2. In: *Proc. PPS 2017*.

Hur, Chung-Kil, Nori, Aditya V., Rajamani, Sriram K., and Samuel, Selva. 2015. A Provably Correct Sampler for Probabilistic Programs. In: *FSTTCS*.

Ismail, Wazim Mohammed, and chieh Shan, Chung. 2016. Deriving a probability density calculator (functional pearl). In: *Proc. ICFP 2016*.

Kallenberg, O. 2014. Stationary and invariant densities and disintegration kernels. *Probab. Theory Relat. Fields*, **160**, 567–592.

Koehl, Albert, Rupasinghe, Kevin, and Lee, Rachel. 2017 (Sept.). *Bloor St. bike lane used by over 6,000 cyclists per day.* Press release. Available at https://bellsonbloor.wordpress.com.

Last, G, and Penrose, M. 2016. *Lectures on the Poisson process.* CUP.

Levy, Paul Blain, Power, John, and Thielecke, Hayo. 2003. Modelling environments in call-by-value programming languages. *Inf. Comput.*, **185**(2).

Mansinghka, Vikash K., Selsam, Daniel, and Perov, Yura N. 2014. Venture: a higher-order probabilistic programming platform with programmable inference.

Moggi, Eugenio. 1991. Notions of computation and monads. *Inf. Comput.*, **93**(1), 55–92.

Narayanan, P, Carette, J, Romano, W, Shan, C-C, and Zinkov, R. 2016. Probabilistic inference by program transformation in Hakaru (system description). In: *Proc. FLOPS 2016.*

Nitti, Davide, Laet, Tinne De, and Raedt, Luc De. 2016. Probabilistic logic programming for hybrid relational domains. *Mach. Learn.*, **103**, 407–449.

Ong, Luke, and Vákár, Matthijs. 2018. S-finite Kernels and Game Semantics for Probabilistic Programming. In: *Proc. PPS 2018.*

Pollard, D. 2002. *A user's guide to measure theoretic probability.* CUP.

Rainforth, Tom, Cornish, Robert, Yang, Hongseok, Warrington, Andrew, and Wood, Frank. 2018. On Nesting Monte Carlo Estimators. In: *Proc. ICML 2018.*

Robert, Christian P. 2007. *The Bayesian Choice: From Decision-Theoretic Foundations to Computational Implementation.* Springer.

Schervish, Mark J. 1995. *Theory of Statistics.* Springer.

Scibior, Adam, Kammar, Ohad, Vákár, Matthijs, Staton, Sam, Yang, Hongseok, Cai, Yufei, Ostermann, Klaus, Moss, Sean K., Heunen, Chris, and Ghahramani, Zoubin. 2018. Denotational validation of higher-order Bayesian inference. In: *Proc. POPL 2018.*

Shan, Chung-Chieh, and Ramsey, Norman. 2016. *Symbolic Bayesian Inference by Symbolic Disintegration.*

Sharpe, M. 1988. *General theory of Markov Processes.* Academic Press.

Staton, S., Yang, H., Heunen, C., Kammar, O., and Wood, F. 2016. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In: *Proc. LICS 2016.*

Staton, Sam. 2017. Commutative semantics for probabilistic programming. In: *Proc. ESOP 2017.*

Staton, Sam, Stein, Dario, Yang, Hongseok, and Nathanael L.. Ackerman, Cameron Freer, Daniel M Roy. 2018. The Beta-Bernoulli Process and Algebraic Effects. In: *Proc. ICALP 2018.*

Stuhlmuller, Andreas, and Goodman, Noah D. 2014. Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs. *Cognitive Systems Research*, **28**, 80–99.

Tjur, Tue. 1980. *Probability based on Radon measures*. Wiley.

Vákár, Matthijs, Kammar, Ohad, and Staton, Sam. 2019. A domain theory for statistical probabilistic programming. In: *Proc. POPL 2019*.

van de Meent, Jan-Willem, Paige, Brooks, Yang, Hongseok, and Wood, Frank. 2018. *An Introduction to Probabilistic Programming*. arxiv:1809.10756.

Wood, Frank, van de Meent, Jan Willem, and Mansinghka, Vikash. 2014. A New Approach to Probabilistic Programming Inference. In: *AISTATS*.

Wu, Yi, Srivastava, Siddharth, Hay, Nicholas, Du, Simon, and Russell, Stuart J. 2018. Discrete-Continuous Mixtures in Probabilistic Programming: Generalized Semantics and Inference Algorithms. Pages 5339–5348 of: *Proc. ICML 2018*.