Discrete Probabilistic Programming Languages III: Observation & Sampling[1]

Steven Holtzen
s.holtzen@northeastern.edu

October 6, 2023

[1] CS7470 Fall 2023: Foundations of Probabilistic Programming.

1 *Observation*

- Why observation?
- `observe` is a powerful keyword that lets us *condition* the program. For instance, suppose I want to model the following scenario: "flip two coins and observe that at least one of them is heads. What is the probability that the first coin was heads?"

  We can encode this scenario as a Disc program:

```
x ← flip 0.5;
y ← flip 0.5;
observe x ∨ y;
return x
```

Listing 1: TwoCoins

  This program outputs the probability distribution:

$$[\texttt{true} \mapsto (0.25 + 0.25)/0.75, \texttt{false} \mapsto 0.25/0.75]$$

- Denotational semantics of `observe`:

$$[\![\texttt{observe } \texttt{e}_1; \texttt{e}_2]\!](v) = \sum_{\{v' \mid [\![e_1]\!](v') = T\}} [\![\texttt{e}_2]\!](v)$$

  Note that, with this definition, the semantics of probabilistic terms are now *unnormalized* (i.e., the distribution does not sum to 1). For example:

$$[\![\texttt{false}]\!] = \begin{cases} \texttt{true} & \mapsto 0 \\ \texttt{false} & \mapsto 0 \end{cases}$$

  The TwoCoins case:

$$[\![\text{TwoCoins}]\!] = \begin{cases} \texttt{true} & \mapsto 0.5 \\ \texttt{false} & \mapsto 0.25 \end{cases}$$

- The main semantic object of interest is the *normalized distribution*, which is given by the **normalized semantics**:

$$[\![\texttt{e}]\!]_D(T) = \frac{[\![\texttt{e}]\!](T)}{[\![\texttt{e}]\!](T) + [\![\texttt{e}]\!](F)},$$

  defined analogously for the false case.

- In order to handle observations, we will compile Disc programs into *two* Prop programs: one that computes the unnormalized probability of returning `true`, and one that computes the probability of evidence (i.e. normalizing constant)

- Inductive description has the shape $\mathtt{e} \rightsquigarrow (\mathtt{p}_1, \mathtt{p}_2)$.

  Our goal will be for this compilation to satisfy the following form of semantics-presevation:

  Theorem 1 (Semantics preservation). *For well-typed closed term* $\mathtt{e}$*, assume* $\mathtt{e} \rightsquigarrow (\varphi, \varphi_A, w)$*. Then,* $[\![\mathtt{e}]\!]_D(\mathtt{true}) = [\![(\varphi \wedge \varphi_A, w)]\!] / [\![(\varphi_A, w)]\!]$.

- Compilation relation:

$$\mathtt{true} \rightsquigarrow (\mathtt{true}, \mathtt{true}, \emptyset) \qquad \mathtt{false} \rightsquigarrow (\mathtt{false}, \mathtt{true}, \emptyset)$$

$$x \rightsquigarrow (x, \mathtt{true}, \emptyset) \qquad \frac{\mathtt{fresh}\ x}{\mathtt{flip}\ \theta \rightsquigarrow (x, \mathtt{true}, [x \mapsto \theta, \overline{x} \mapsto 1-\theta])}$$

$$\frac{\mathtt{e}_1 \rightsquigarrow (\varphi, \varphi_A, w) \qquad \mathtt{e}_2 \rightsquigarrow (\varphi', \varphi'_A, w')}{x \leftarrow \mathtt{e}_1; \mathtt{e}_2 \rightsquigarrow (\varphi'[\varphi/x], \varphi'_A[\varphi/x] \wedge \varphi_A, w_1 \cup w_2)}$$

$$\frac{\mathtt{e}_1 \rightsquigarrow (\varphi, \mathtt{true}, \emptyset) \qquad \mathtt{e}_2 \rightsquigarrow (\varphi', \varphi'_A, w)}{\mathtt{observe}\ \mathtt{e}_1; \mathtt{e}_2 \rightsquigarrow (\varphi', \varphi \wedge \varphi'_A, w)}$$

  Most of these rules are unchanged from the previous compilation, except for bind and `observe`.

- Example derivation:

$$\frac{\mathtt{flip}\ \theta \rightsquigarrow (f, \mathtt{true}, [f \mapsto 1/2, \overline{f} \mapsto 1/2]) \qquad \dfrac{x \rightsquigarrow (x, \mathtt{true}, \emptyset) \qquad \dfrac{x \rightsquigarrow (x, \mathtt{true}, \emptyset)}{\mathtt{return}\ x \rightsquigarrow (x, \mathtt{true}, \emptyset)}}{\mathtt{observe}\ x; \mathtt{return}\ x \rightsquigarrow (x, x, \emptyset)}}{x \leftarrow \mathtt{flip}\ \theta;\ \mathtt{observe}\ x; \mathtt{return}\ x \rightsquigarrow (f, f, [f \mapsto 1/2, \overline{f} \mapsto 1/2])}$$

  Check that this satisfies semantics preservation.

2 *Some notes on the project*

- Instead of compiling to Prop, you can work directly on the BDD

- The basic primitive operation you will perform on BDDs is *weighted model counting*, which is a more general term for what we've so far calling the semantics of Prop:

Definition 1 (Weighted model count). *Let φ be a propositional formula and w be a map from literals (assignments to variables) to real-valued weights. The* weighted model count *is defined:*

$$\texttt{WMC}(\varphi, w) \triangleq \sum_{I \models \varphi} \prod_{\ell \in I} w(\ell). \tag{1}$$

You can see an example of running a weighted model count in `disc/lib/kc.ml`

3 *Sampling & approximate reasoning*

- Up until now we have been exclusively discussing *exact reasoning*: computing the exact probability that a program will output a particular value
- Problems with exact reasoning:
  - State-space explosion
  - Limited expressive power: how can we handle continuous probability, or loops that may never terminate?
  - "All-or-nothing": exact answer or nothing at all
- An alternative is *approximate reasoning*. Many of the most popular PPLs in use today support exclusively this mode of reasoning.[2]

[2] For example, Stan [Carpenter et al., 2017].

- There is an entirely separate school of PPLs that reason by *sampling*.
- The crucial mechanism is the *sample mean*, which gives an estimate of the expectation of a random variable:

  Definition 2 (Expectation). *Let $(\Omega, \Pr)$ be a probability space and $f : \Omega \to \mathbb{R}$ be a random variable. The* expectation *(or* average value*) of f with respect to* $\Pr$ *is defined:*

  $$\mathbb{E}_{\Pr[f]} \triangleq \sum_{\omega \in \Omega} \Pr(\omega) f(\omega). \tag{2}$$

  Definition 3 (Sample mean). *Let $(\Omega, \Pr)$ be a probability space and $f : \Omega \to \mathbb{R}$ be a random variable. Then, the* sample mean *of f with N samples is defined:*

  $$\frac{1}{N} \sum_{\omega_i \sim \Pr}^{N} f(\omega_i), \tag{3}$$

  where the notation $\omega_i \sim \Pr$ denotes drawing a sample ω_i from the probability distribution $\Pr$.

One may wonder *how quickly* a particular estimate of the mean approaches the true value (i.e., how many samples one must draw in order to have an accurate estimate with high probability). There are many bounds of this sort known broadly as *concentration inequalities*; Shalev-Shwartz and Ben-David [2014] has a nice summary of some of the useful concentration inequalities that arise in practice in the appendix.

- The reason why we use the sample estimator is that the *law of large numbers* guarantees that, as $N \to \infty$, the sample mean approaches the expectation, i.e.:

$$\lim_{N\to\infty} \frac{1}{N} \sum_{\omega_i \sim \Pr}^{N} f(\omega_i) = \mathbb{E}_{\Pr}[f]. \tag{4}$$

- What will do is give a semantics to programs in terms of expectations, and then use the expectation estimator in order to get an approximation for the program's behavior

Sampling semantics for Disc

This secton is based on the semantics given by Culpepper and Cobb [2017].

- **Goal**: Give a semantics that draws samples from $[\![\texttt{e}]\!]$, where `e` is a (probabilistic) Disc term
- We still want our semantics to be a *deterministic relation* on terms; how can we draw samples using a deterministic relation?
- Solution: Add a *source of randomness* to our context (just like how your computer has `/dev/rand`)
- To get our feet wet with this new style of semantics, let's consider a tiny sub-language of Disc with only the following syntax with only fair coin-flips and no observations:

```
e ::= flip 1/2 | x ← e; e | x | e ∧ e | e ∨ e | ¬ e
```

  The denotation of this sub-language is inherited from Disc.

- To draw samples from this language, we add to our evaluation judgment an *infinite stream of fair coin-flips* $\sigma \in \mathbb{B}^{\mathbb{N}}$. To make this formal, we will need a way of representing a probability distribution on $\mathbb{B}^{\mathbb{N}}$. This is actually much trickier to define than it seems at first!

  Let's start by defining the probability of any *finite subsequence* of bits:

$$\Pr((v_1, v_2, \dots, v_n)) = \frac{1}{2^n} \tag{5}$$

  Clearly as $n \to \infty$, this probability approaches 0. This seems broken – how can it be that this defines a valid probability distribution (i.e., sums to 1) on all elements $\sigma \in \mathbb{B}^{\mathbb{N}}$? We will see more on this next time, but for now, accept that this is a valid definition of probability, and we will make this more formal later.

  We will assume that standard properties of probability distributions (such as marginalization) hold of Pr, for instance:

$$\sum_{(v_2, v_3, v_4, \dots) \in \mathbb{B}^{\mathbb{N}}} \Pr(v_1 = \texttt{true}, v_2, v_3, v_4, \dots) = 1/2 \tag{6}$$

- To handle binding, we will need a way to split this bit-stream up. We notate this π_L and π_R, which split σ into two disjoint random sets of bits. We can define these projections in a number of ways, but a simple way is to define π_L to take all even bits and π_R to take all odd bits:

$$\pi_L(v_1, v_2, v_3, \dots) \triangleq (v_2, v_4, v_6, ...) \qquad \pi_R(v_1, v_2, v_3, \dots) \triangleq (v_1, v_3, v_5, ...)$$

- Given access to this context, we can define a relation $\sigma \vdash \mathtt{e} \Downarrow^S v$ for our tiny language above (showing only the probabilistic terms):

$$v :: \sigma \vdash \mathtt{flip} \Downarrow^S v \qquad \frac{\pi_L(\sigma) \vdash \mathtt{e}_1 \Downarrow v_1 \qquad \pi_R(\sigma) \vdash \mathtt{e}_2[v_1/x] \Downarrow v_2}{\sigma \vdash x \leftarrow \mathtt{e}_1 ; \mathtt{e}_2 \Downarrow^S v_2}$$

- Since our semantics are deterministic, can define a function $ev(\sigma, \mathtt{e})$ that produces the value that the (closed) term $\mathtt{e}$ evaluates to for σ:

Theorem 2. *Let* Pr *be the distribution on the infinite bit streams* $\mathbb{B}^{\mathbb{N}}$. *Assume* $\Gamma \vdash \mathtt{e}$. *Then, for any* $\gamma \in [\![\Gamma]\!]$, $\mathbb{E}_{\sigma\sim\Pr}[\mathbb{1}(ev(\sigma, \mathtt{e}[\gamma]) = \mathtt{true})] = [\![\mathtt{e}[\gamma]]\!]\,(\mathtt{true})$.

Proof. The non-probabilistic cases are straightforward. Let's see the `flip` case first. We can disregard γ since this term is closed.

$$\begin{aligned}\mathbb{E}_{\sigma\sim\Pr}[\mathbb{1}(ev(\sigma, \mathtt{e}))] &= \sum_{(v::\sigma)\in\mathbb{B}^{\mathbb{N}}} \Pr(v :: \sigma)\mathbb{1}[(ev(v :: \sigma, \mathtt{e})) = \mathtt{true})] \\ &= \Pr(v = \mathtt{true})\mathbb{1}[\mathtt{true} = \mathtt{true}] + \Pr(v = \mathtt{false})\mathbb{1}[\mathtt{false} = \mathtt{true}] \\ &= 1/2 = [\![\mathtt{flip\ 1/2}]\!]\,(\mathtt{true}).\end{aligned}$$

Now for the bind case, $x \leftarrow \mathtt{e}_1 ; \mathtt{e}_2$. Assume by induction that:

- $\mathbb{E}_{\sigma\sim\Pr}[\mathbb{1}(ev(\sigma, \mathtt{e}_1) = \mathtt{true})] = [\![\mathtt{e}_1]\!]\,(\mathtt{true})$.
- $\mathbb{E}_{\sigma\sim\Pr}[\mathbb{1}(ev(\sigma, \mathtt{e}_2) = \mathtt{true})] = [\![\mathtt{e}_2]\!]\,(\mathtt{true})$.

Proceeding:

$$\mathbb{E}_{\sigma\sim\Pr}[\mathbb{1}(ev(x \leftarrow \mathtt{e}_1 ; \mathtt{e}_2) = \mathtt{true})]$$

□

- Now we are left with one final problem: how can we "run these semantics" on a computer? I.e., how can we effectively sample from an infinite stream of random bits, which seems to require infinite memory (and in which each sample has probability 0)? We can *sample the bits lazily*: each time a fresh random bit is required, flip a fair coin to sample it.

- Now, let's compare sampling against exact: when might one prefer sampling over exact, and vise versa?

References

Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76, 2017.

Ryan Culpepper and Andrew Cobb. Contextual equivalence for probabilistic programs with continuous random variables and scoring. In *European Symposium on Programming*, pages 368–392. Springer, 2017.

Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.