Observation Propagation for Importance Sampling with Likelihood Weighting *

Extended Abstract

Ryan Culpepper
Northeastern University
ryanc@ccs.neu.edu

A *universal probabilistic programming language* [4–6] consists of a general-purpose language extended with two probabilistic features: the ability to make random (probabilistic) choices and the ability to make *observations*. For expressiveness and efficiency, it is useful to consider observations that have nonnegative real *likelihoods* rather than simple boolean truth values. A program in such a language represents a probabilistic process; the chance of producing a particular answer is determined by the random choices made along the way and the likelihoods of the observations.

Existing probabilistic programming languages typically support a constrained form of observation—such as requiring the distribution in explicit form—or a general weighting operation, like **factor**. This work explores the interaction between observation and the other computational features of the language. We present a big-step semantics of importance sampling with likelihood weighting for a core universal probabilistic programming language with *observation propagation*.

1. Introduction

A typical probabilistic programming language (PPL) has features for sampling random variables and conditioning based on predicates. For example, here is a tiny model of colds and coughs:

> In general, I have a 5% chance of having a cold. When I have a cold, I have a 90% chance of having a cough; if I don't have a cold, I have a 2% chance of having a cough anyway. Given that I have a cough, what is the probability I have a cold?

We can represent that model with the following program:[1]

cold = **sample** bernoulli(0.05)
cough = **sample** bernoulli(**if** *cold* **then** 0.9 **else** 0.02)
condition (*cough* = 1)
cold

A naïve implementation can compute an expectation or empirical distribution for this program by simply sampling both *cold* and *cough* and rejecting executions in which the condition is false. This is often called *rejection sampling* (or, more properly, *logic sampling*). But if the condition is unlikely given the priors, many executions will be wasted, and this approach does not work at all if the condition is an equality test on a continuous random variable.

An alternative is *importance sampling with likelihood weighting*, where instead of sampling and checking the observed random variable, we *weight* a program execution by the likelihood (probability or probability density) that the random variable satisfies its condition, given the sampled values of its parameters.

Likelihood weighting is sometimes exposed to the programmer in the form of a feature such as **factor**, which multiplies the current execution's weight by the given likelihood:[2]

cold = **sample** bernoulli(0.05)
factor (**if** *cold* **then** 0.9 **else** 0.02)
cold

For efficiency, we wish to use likelihood weighting instead of the sample, check, and reject approach of logic sampling whenever possible. But **factor** provides likelihood weighting at the expense of model clarity. Consider a simple univariate linear regression model. Given parameters A (slope), B (intercept), and E (noise standard deviation), we would like to define the output function in a natural style as follows:

$f\,x = A * x + B +$ **sample** normal(0, E)

But to score the evidence we must also contort the code into an observing version of f that uses **factor**:

$f\_obs\ x\ y =$ **factor** density(normal(0, E), $y - A * x - B$)

We introduce an **observe** form that propagates observations through computation, including function calls and arithmetic operations. When an observation reaches a use of **sample**, it performs likelihood weighting. This lets us use the single definition of f above for both observation and prediction:

for (x, y) **in** *data*:
  observe y **from** $f\,x$
$f\,x\_to\_predict$

* This material is based upon work sponsored by the Air Force Research Laboratory (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-14-C-0002. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

[1] We avoid the traditional "$X \sim D$" notation, since it does not easily express sampling mixed with computation (e.g., "$X = 1 +$ **sample** D") which is what we want to explore in this paper.

[2] In practice, **factor** usually takes a log-likelihood to avoid floating-point underflow. We ignore such issues in this paper.

The remainder of this paper gives a big-step semantics for observation propagation and discusses its implementation in the Gamble PPL [2].

2. Syntax and Semantics of Core Gamble

We model observation propagation with Core Gamble, a functional language with real arithmetic and probabilistic effects.

$$\begin{array}{rcl} e & ::= & r \mid x \mid \lambda x.\, e \mid e\ e \mid op^n(e_1, \cdots, e_n) \\ & \mid & \textbf{sample}\ e \mid \textbf{observe}\ e\ \textbf{from}\ e \\ r & \in & \mathbb{R} \\ op^1 & ::= & \text{bernoulli} \mid \cdots \\ op^2 & ::= & + \mid - \mid * \mid \div \mid \text{normal} \mid \text{uniform} \mid \text{density} \mid \cdots \end{array}$$

We assume syntactic sugar for **let**, **if**, and sequencing; we omit the **let** keyword when clarity permits.

Evaluation is defined via the judgment $\rho, \sigma \vdash e \Downarrow v, w$, where ρ is an environment, $\sigma \in [0,1]^\omega$ is an infinite stream of real numbers from the unit interval, e is the expression to evaluate, v is the resulting value, and w is a likelihood weight. The σ argument acts as the source of randomness—evaluation is a deterministic function of ρ, e, and σ. Rather than threading σ through evaluation like a store, rules with multiple sub-derivations split the stream (e.g., into the even and odd elements for a two-way split).

The rules for the functional subset are unsurprising. Here is the rule for application; we write a closure as $\langle \rho, \lambda x.\, e \rangle$:

$$\frac{\sigma = \text{interleave}(\sigma_1, \sigma_2, \sigma_3) \qquad \rho, \sigma_1 \vdash e_1 \Downarrow \langle \rho', \lambda x.\, e_b \rangle, w_1 \qquad \rho, \sigma_2 \vdash e_2 \Downarrow v_2, w_2 \qquad \rho'[x \mapsto v_2], \sigma_3 \vdash e_b \Downarrow v, w_3}{\rho, \sigma \vdash e_1\ e_2 \Downarrow v, w_1 \cdot w_2 \cdot w_3}$$

The **sample** form produces a value from the given distribution by applying its inverse-CDF to an element from the random stream:

$$\frac{\sigma = \text{interleave}(\sigma_1, \sigma_2) \qquad \rho, \sigma_1 \vdash e \Downarrow dist, w \qquad u = \sigma_2(0) \qquad v = \text{invcdf}(dist, u)}{\rho, \sigma \vdash \textbf{sample}\ e \Downarrow v, w}$$

The **observe** form evaluates its second subexpression using a separate "observing" relation:

$$\frac{\sigma = \text{interleave}(\sigma_1, \sigma_2) \qquad \rho, \sigma_1 \vdash e_1 \Downarrow v, w_1 \qquad \rho, \sigma_2, 1 \vdash e_2 \Downarrow^{obs} v, w_2}{\rho, \sigma \vdash \textbf{observe}\ e_1\ \textbf{from}\ e_2 \Downarrow v, w_1 \cdot w_2}$$

The judgment $\rho, \sigma, s \vdash e \Downarrow^{obs} v, w$ has nearly the same interpretation as the $\Downarrow$ relation, aside from the extra s argument. The difference is that while the $\Downarrow$ relation is used with the v and w components in output mode, the $\Downarrow^{obs}$ relation has only the w component in output mode; the v component becomes an input.

The s component, initially 1, represents an adjustment that must be made to density likelihoods (but not mass likelihoods) due to change of variables. Consider the following example:

```
X = sample bernoulli(0.5)
observe 0 from
  if X = 1
  then sample uniform(−2, 2)
  else 2 ∗ sample uniform(−1, 1)
X
```

Clearly, **sample** uniform$(-2, 2)$ and $2 *$ **sample** uniform$(-1, 1)$ represent the same distribution. Thus the observation should have equal effect for either value of *X*, and the expectation of *X* should be $\frac{1}{2}$. But the density at 0 in uniform$(-2, 2)$ is $\frac{1}{4}$, whereas in uniform$(-1, 1)$ it is $\frac{1}{2}$, so if we simply weighted by the density of the observed value in the sampled distribution, we would get an expectation for *X* of $\frac{1}{3}$. The missing piece of the puzzle is that *densities are derivatives*, so when we propagate the observation through $2 * [\,]$, we must obey the chain rule and divide the likelihood by the derivative—in this case, 2.

The following rule shows propagation through binary operations. We always propagate to the second argument, by arbitrary convention.

$$\frac{\sigma = \text{interleave}(\sigma_1, \sigma_2) \qquad \rho, \sigma_1 \vdash e_1 \Downarrow v_1, w_1 \qquad \{v_2\} = op^{-1}(v \mid v_1) \qquad s' = s \div \left|\frac{\partial op(v_1, v_2)}{\partial v_2}(v_1, v_2)\right| \qquad \rho, \sigma_2, s' \vdash e_2 \Downarrow^{obs} v_2, w_2}{\rho, \sigma, s \vdash op(e_1, e_2) \Downarrow^{obs} v, w_1 \cdot w_2}$$

When **sample** occurs in an observing context, it updates the likelihood weight. If the distribution is continuous, the likelihood is a density and we scale it by s; otherwise, it is a mass and s is irrelevant.

$$\frac{\rho, \sigma \vdash e \Downarrow dist, w_1 \qquad w_2 = \begin{cases} \text{pmf}(dist, v) & dist \text{ is discrete} \\ \text{pdf}(dist, v) \cdot s \cdot \epsilon & dist \text{ is continuous} \end{cases}}{\rho, \sigma, s \vdash \textbf{sample}\ e \Downarrow^{obs} v, w_1 \cdot w_2}$$

We represent the likelihood as a monomial over ϵ; the degree indicates how many densities are included. That allows us to distinguish between the branches in the following program:

```
X = sample (bernoulli 0.5)
observe 0 from
  if X = 1
  then sample bernoulli(0.5)
  else sample uniform(−1, 1)
X
```

The expectation of *X* is 1, because the likelihood of the first branch, 0.5, is greater than the likelihood of the second, 0.5ϵ.

3. Implementation

We have implemented observation propagation in Gamble [2], our probabilistic programming language embedded in Racket [3]. Gamble uses the same observation propagation translation to support importance sampling, Metropolis-Hastings, and enumeration. The translation extends the lightweight-MH translation [9]. Our translation adds two arguments to every function definition and call: `ADDR` and `OBS`. `ADDR` carries the call-site stack, as in the lightweight-MH translation. The `OBS` argument is either `#false` to indicate normal evaluation ($\Downarrow$) or a record containing the observed value and the change-of-variable density scaling factor s for observation contexts ($\Downarrow^{obs}$).

The following example shows the translation of a function containing both a tail call and a non-tail call:

```
(define (f x)                (define (f ADDR OBS x)
  (g (h x)))          ⇒        (g (cons 'cs_1 ADDR) OBS
                                 (h (cons 'cs_2 ADDR) #false x)))
```

For brevity, we omit the translation of observation-propagating primitives such as `+` and `*`.

Gamble uses Racket's macro system to register and apply the rewrite rules for observation propagator. We provide a hook that enables library authors can declare new observation propagators. For example, the Gamble binding of the Racket array library declares `array+` as an observation propagator with 4 lines of code.

4. Ongoing and Future Work

We will explore the contextual equivalence relation generated from our semantics using an intermediate function that integrates likelihood with respect to σ; the likelihood is 0 except on pre-images of the evaluation function [8]. We can then try to connect contextual equivalence to our denotational semantics work. We will also attempt to connect likelihood weighting and disintegration [1, 7].

References

[1] Joseph T Chang and David Pollard. Conditioning as disintegration. *Statistica Neerlandica*, 51(3):287–317, 1997.

[2] Ryan Culpepper. Gamble. `https://github.com/rmculpepper/gamble`, 2015.

[3] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. `http://racket-lang.org/tr1/`.

[4] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *In UAI*, pages 220–229, 2008.

[5] Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *ArXiv e-prints*, March 2014.

[6] Brooks Paige and Frank Wood. A compilation target for probabilistic programming languages. *ArXiv e-prints*, March 2014.

[7] Chung-Chieh Shan and Norman Ramsey. Symbolic bayesian inference by lazy partial evaluation. Unpublished, July 2015. See `http://www.cs.tufts.edu/~nr/pubs/disintegrator-abstract.html`.

[8] Neil Toronto, Jay McCarthy, and David Van Horn. Running probabilistic programs backwards. In *European Symposium on Programming*, April 2015.

[9] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proc. of the 14th Artificial Intelligence and Statistics*, 2011.