Design and Implementation of Anglican Probabilistic Programming Language

David Tolpin   Jan Willem van de Meent   Hongseok Yang
Frank Wood

September 1, 2016

Paper and slides:
https://bitbucket.org/probprog/anglican-white-paper

Outline

Why probabilistic programming?

Why functional?

Intuition

Probabilistic program:

- A program with random computations.
- Distributions are conditioned by 'observations'.
- Values of certain expressions are 'predicted' — **the output**.

Can be written in any language (extended by `sample` and `observe`).

Example: Model Selection

```
(let [;; Guessing a distribution
      dist (sample (categorical
                    [[normal 1] [gamma 1]
                     [uniform-continuous 1]
                     [uniform-discrete 1]]))
      a (sample (gamma 1 1))
      b (sample (gamma 1 1))
      d (dist a b)]

  ;; Observing samples from the distribution
  (loop [data data]
    (when (seq data)
      (let [[x & data] data]
        (observe d x))
      (recur data)))

  ;; Predicting a, b and the distribution
  (predict :a a)
  (predict :b b)
  (predict :d d))
```

Definition

A **probabilistic program** is a stateful deterministic computation $\mathcal{P}$:

Definition

A **probabilistic program** is a stateful deterministic computation $\mathcal{P}$:

- Initially, $\mathcal{P}$ expects no arguments.

Definition

A **probabilistic program** is a stateful deterministic computation $\mathcal{P}$:

- Initially, $\mathcal{P}$ expects no arguments.
- On every call, $\mathcal{P}$ returns
  - a distribution F,
  - a distribution and a value (G, y),
  - a value z,
  - or $\perp$.

Definition

A **probabilistic program** is a stateful deterministic computation $\mathcal{P}$:

- Initially, $\mathcal{P}$ expects no arguments.
- On every call, $\mathcal{P}$ returns
  - a distribution F,
  - a distribution and a value (G, y),
  - a value z,
  - or $\perp$.
- Upon returning F, $\mathcal{P}$ expects $x \sim F$.

Definition

A **probabilistic program** is a stateful deterministic computation $\mathcal{P}$:

- Initially, $\mathcal{P}$ expects no arguments.
- On every call, $\mathcal{P}$ returns
  - a distribution F,
  - a distribution and a value (G, y),
  - a value z,
  - or $\perp$.
- Upon returning F, $\mathcal{P}$ expects $x \sim F$.
- Upon returning $\perp$, $\mathcal{P}$ terminates.

Definition

A **probabilistic program** is a stateful deterministic computation $\mathcal{P}$:

- Initially, $\mathcal{P}$ expects no arguments.
- On every call, $\mathcal{P}$ returns
  - a distribution F,
  - a distribution and a value (G, y),
  - a value z,
  - or $\perp$.
- Upon returning F, $\mathcal{P}$ expects $x \sim F$.
- Upon returning $\perp$, $\mathcal{P}$ terminates.

A program is run by calling $\mathcal{P}$ repeatedly until termination.
The probability of each **trace** is $\propto \prod_{i=1}^{|\boldsymbol{x}|} p_{F_i}(x_i) \prod_{j=1}^{|\boldsymbol{y}|} p_{G_j}(y_j)$.

Evolution of Compiler Development

- 1950s: Ask a programmer to implement an algorithm efficiently: They'll write it on their own in assembly.
  - No good compilers; problem-dependent optimizations that only human expert could see.

Evolution of Compiler Development

- 1950s: Ask a programmer to implement an algorithm efficiently: They'll write it on their own in assembly.
  - No good compilers; problem-dependent optimizations that only human expert could see.
- 1970s: Novice programmers use high-level languages and let compiler work out details, experts still write assembly.
  - Experts still write custom assembly when speed critical.

Evolution of Compiler Development

- 1950s: Ask a programmer to implement an algorithm efficiently: They'll write it on their own in assembly.
  - No good compilers; problem-dependent optimizations that only human expert could see.
- 1970s: Novice programmers use high-level languages and let compiler work out details, experts still write assembly.
  - Experts still write custom assembly when speed critical.
- 2000s: On most problems, even experts can't write faster assembly than optimizing compilers.
  - can automatically profile (JIT).
  - can take advantage of paralellization, complicated hardware, make appropriate choices w.r.t. caching.
  - Compilers embody decades of compiler research

Evolution of Inference

- 2010: Ask a grad student to implement inference efficiently: They'll write it on their own.
  - No good automatic inference engines; problem-dependent optimizations that only human expert can see.

Evolution of Inference

- 2010: Ask a grad student to implement inference efficiently: They'll write it on their own.
  - No good automatic inference engines; problem-dependent optimizations that only human expert can see.
- 2015: Novice grad students use automatic inference engines and let compiler work out details, experts still write their own inference.
  - Experts still write custom inference when speed critical.

Evolution of Inference

- 2010: Ask a grad student to implement inference efficiently: They'll write it on their own.
  - No good automatic inference engines; problem-dependent optimizations that only human expert can see.
- 2015: Novice grad students use automatic inference engines and let compiler work out details, experts still write their own inference.
  - Experts still write custom inference when speed critical.
- 2020: On most problems, even experts can't write faster inference than mature automatic inference engines.
  - Can use paralellization, sophisticated hardware
  - Can automatically choose appropriate methods (meta-reasoning?).
  - Inference engines will embody 1 decade (!) of PP research.

Outline

Why probabilistic programming?

Why functional?

Inference Objective

- Suggest **most probable explanation** (MPE) - most likely assignment for all non-evidence variable given evidence.

Inference Objective

- Suggest **most probable explanation** (MPE) - most likely assignment for all non-evidence variable given evidence.
- Approximately **compute integral** of the form

$$\Phi = \int_{-\infty}^{\infty} \varphi(x) p(x) dx$$

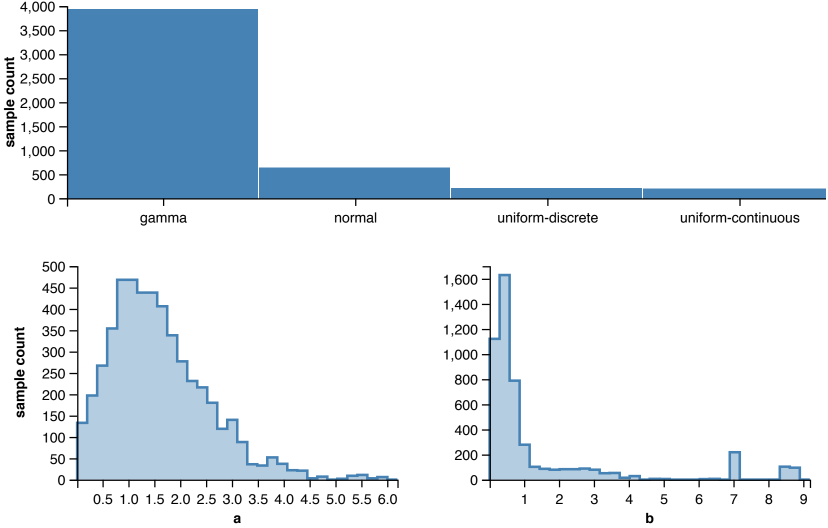
Inference Objective

- Suggest **most probable explanation** (MPE) - most likely assignment for all non-evidence variable given evidence.
- Approximately **compute integral** of the form

$$\Phi = \int_{-\infty}^{\infty} \varphi(x) p(x) dx$$

- Continuously and **infinitely generate a sequence of samples** drawn from the distribution of the output expression — so that someone else puts it in good use (vague but common). ✓

Example: Inference Results



Importance Sampling

loop

Run $\mathcal{P}$, computing weight $w = \prod_{j=1}^{|\boldsymbol{y}|} p_{G_j}(y_j)$.

output $\boldsymbol{z}, w$.

end loop

- Simple — good.
- Slow convergence (unless one knows the answer) — bad.

Can we do better?

Lightweight Metropolis-Hastings (LMH)

Run $\mathcal{P}$ once, remember $\boldsymbol{x}, \boldsymbol{z}$.
loop
    Uniformly select x_i.
    Propose a value for x_i.
    Run $\mathcal{P}$, remember $\boldsymbol{x'}, \boldsymbol{z'}$.
    Accept $(\boldsymbol{x}, \boldsymbol{z} = \boldsymbol{x'}, \boldsymbol{z'})$ or reject with MH probability.
    Output $\boldsymbol{z}$.
end loop

Can we do better?

- Particle Markov Chain Monte Carlo
- Variational Inference
- ...

Challenges

- 'Transformational compilation' — limited languages, some less ugly than others.
- Slow inference — Markov Chain Monte Carlo is **always slow** (but there are Hybrid Monte Carlo, Variational Inference, ...)
- How to learn persistently — what is the learned model? Transfer learning?
- Handling indeterminism (learning policies).

Thank you!

Questions?