Design and Implementation of Anglican Probabilistic Programming Language

David Tolpin Jan Willem van de Meent Hongseok Yang
Frank Wood

September 2, 2016

https://bitbucket.org/probprog/anglican-white-paper
https://bitbucket.org/probprog/anglican
http://www.robots.ox.ac.uk/~fwood/anglican/index.html

Outline

Motivation

Design Outline

Implementation Highlights

Inference Algorithms

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))
```

More examples

- *Intruder detection* — given a log of **times** and **amounts** of payments in a bank account, how likely that the baccount was compromised?

More examples

- *Intruder detection* — given a log of **times** and **amounts** of payments in a bank account, how likely that the baccount was compromised?
- *Counterfactual reasoning* — There are **two routes** from Jerusalem to Tel Aviv: 1 and 443. Based on traffic reports, I chose route 1 and was late. Would I arrive on time If I chose 443 instead?

More examples

- *Intruder detection* — given a log of **times** and **amounts** of payments in a bank account, how likely that the baccount was compromised?
- *Counterfactual reasoning* — There are **two routes** from Jerusalem to Tel Aviv: 1 and 443. Based on traffic reports, I chose route 1 and was late. Would I arrive on time If I chose 443 instead?
- *(Due to Stuart Russell)* If you observe that a student GPA is exactly 4.0 in a model of transcripts of students from the USA (GPA's from 0.0 to 4.0) and India (GPA's from 0.0 to 10.0) what is the probability that the student is from India?

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

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

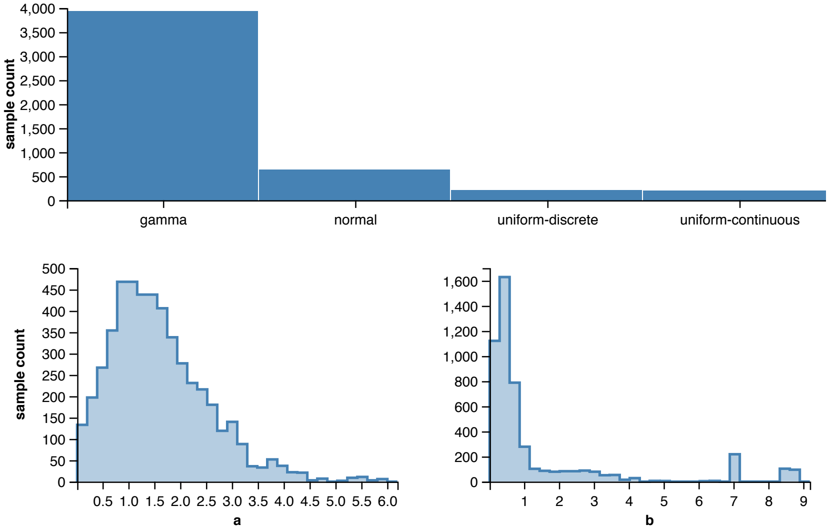
Inference Objective

- 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 program, computing weight based on observations.
Output result and weight.
end loop

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

Can we do better?

Lightweight Metropolis-Hastings (LMH)

Run program once, remembering random choices.
loop
    Uniformly select one random choice.
    Propose a new value for the choice.
    Re-run the program.
    Accept or reject with MH probability.
    Output result.
end loop

Can we do better?

- Particle methods
- Variational inference
- ...

Why functional?

We want a functional language because an inference algorithm controls the execution:

- A program is run many (often many hundreds of thousands) of times (with almost any algorithm).
- A program must be partially re-executed multiple times from different positions (particle methods).
- We want to reason about the distribution defined by the program.

Why Closure?

- Runs on JVM — easy deployment and access to libraries.

Why Closure?

- Runs on JVM — easy deployment and access to libraries.
- A Lisp — we (ab)use the macro facility.

Why Closure?

- Runs on JVM — easy deployment and access to libraries.
- A Lisp — we (ab)use the macro facility.
- Church (`https://en.wikipedia.org/wiki/Church_(programming_language)`) is derived from Scheme.

Why Closure?

- Runs on JVM — easy deployment and access to libraries.
- A Lisp — we (ab)use the macro facility.
- Church (`https://en.wikipedia.org/wiki/Church_(programming_language)`) is derived from Scheme.

Others use:

- Scheme — (Church, Venture).
- Scala — Figaro.
- Haskell — Hakaru, Model-Bayes.
- ...

Why Closure?

- Runs on JVM — easy deployment and access to libraries.
- A Lisp — we (ab)use the macro facility.
- Church (`https://en.wikipedia.org/wiki/Church_(programming_language)`) is derived from Scheme.

Others use:

- Scheme — (Church, Venture).
- Scala — Figaro.
- Haskell — Hakaru, Model-Bayes.
- ...

As well as Python, C#, and other languages.

Outline

Anglican and Clojure

One language on top of (or besides) another:

- Interpeter.
- Source-to-source compiler.
- Integrated language. ✓

Anglican and Clojure

One language on top of (or besides) another:

- Interpeter.
- Source-to-source compiler.
- Integrated language. ✓

Integrated language

- either extends the syntax of the host language;
- or changes the semantics of the host language constructs ✓.

Anglican and Clojure

One language on top of (or besides) another:

- Interpeter.
- Source-to-source compiler.
- Integrated language. ✓

Integrated language

- either extends the syntax of the host language;
- or changes the semantics of the host language constructs ✓.

Anglican is

- Integrated with Clojure.
- Shares syntax.
- Alters operational semantics.

Design challenges and choices

- Anglican syntax: Clojure + probabilistic constructs.
  - Subset of Clojure.
  - Special forms `sample` and `observe`.
  - Macros to delimit Anglican code within a Clojure program.

Design challenges and choices

- Anglican syntax: Clojure + probabilistic constructs.
  - Subset of Clojure.
  - Special forms `sample` and `observe`.
  - Macros to delimit Anglican code within a Clojure program.

- Source-to-source compilation of Anglican into Clojure.
  - CPS Transformation, with some tricks.
  - Transparent use of Clojure functions from Anglican.

Design challenges and choices

- Anglican syntax: Clojure + probabilistic constructs.
  - Subset of Clojure.
  - Special forms `sample` and `observe`.
  - Macros to delimit Anglican code within a Clojure program.
- Source-to-source compilation of Anglican into Clojure.
  - CPS Transformation, with some tricks.
  - Transparent use of Clojure functions from Anglican.
- Inference algorithms.
  - Required to run an Anglican program.
  - Step in at checkpoints (`sample` and `observe`).
  - Execute programs by calling continuations repeatedly.

The language

A subset of Clojure, wrapped inside `defquery`:

- `if`, `when`, `cond`, `case`, `let`, `and`, `or`, `fn`.
- Vector destructuring in bindings of `let` and `fn`.
- Compound literals for vectors, hash maps, and sets.
- `loop`/`recur` — a convenience.

Core library:

- All of Clojure core library, except for higher-order functions.
- `map`, `reduce`, `filter`, `some`, `repeatedly`, `comp`, `partial`.

Any Clojure function can be called from Anglican.

Macro-based compilation

Anglican code macro-compiled into Clojure:

```
(loop [data data]
  (if (seq data)
    (let [[x & data] data]
      (observe (gamma a b) x)
      (recur data))
    (predict :a a)))
```

```
(fn loop [C23151 state data]
   (if (seq data)
     (let [[x & data] data]
       (->observe 'O23153 (gamma a b) x
         (fn do23152 [_ state]
           (loop C23151 state data))
         state))
     (fn []
       (C23151
         nil
         (add-predict state :a a)))))
```

Outline

Probabilistic forms

Two **probabilistic** forms:

- `sample` — draws a value from a distribution;
- `observe` — conditions the *a posteriori* distribution by observing a value from a distribution.

This is where the inference algorithm steps in:

```
=> (cps-of-expression '(sample dist) 'cont)
(->sample dist cont $state)

=> (cps-of-expression '(observe dist val) 'cont)
(->observe dist val cont $state)
```

Memoization

Memoization of random choice:

- A person has a random eye color.
- But the *same* person has a fixed eye color.

```
(let [eye-color (mem (fn [person]
                       (sample
                         (categorical
                           ['brown 0.5]
                           ['green 0.5]))))]
  (if (not= (eye-color 'bill) (eye-color 'john))
    (predict (eye-color 'bill))
    (predict (eye-color 'john))))
```

Compiling a memoized function

```
(mem (fn [person] ...))
```

```
(fn []
 (cont
  ;; every memoization gets a unique key
  (let [M23145 (gensym "M")]
   (fn [C23144 $state & P23147]
    (if (in-mem? $state M23145 P23147)
     ;; previously memoized result
     (fn []
       (C23144
         (get-mem $state M23145 P23147)
         $state))
     ;; new computation
     (clojure.core/apply
      (fn [C23150 $state person]
        (fn [] (C23150 ... $state)))
      ;; memoize result in state
      (fn [V23146 $state]
       (fn []
        (C23144
          V23146
          (set-mem $state M23145 P23147 V23146))))
      $state
      P23147))))
  $state))
```

Managing stack size

- Clojure does not support tail-call optimization (TCO).
- Under CPS transformation the stack will explode.

Managing stack size

- Clojure does not support tail-call optimization (TCO).
- Under CPS transformation the stack will explode.
- Anglican uses *trampolining*: every continuation call is wrapped into a *thunk*.

The thunk is returned ...

```
(fn [x y] (+ x y))
```

```
(fn [cont $state x y]
  (fn []
    (cont (+ x y) $state)))
```

... and called by `trampoline`:

```
(defn exec
  [algorithm prog value state]
  (loop [step (trampoline prog value state)]
    (let [next (checkpoint algorithm step)]
      (if (fn? next)
        (recur (trampoline next))
        next))))
```

Outline

Importance sampling

Importance sampling is the simplest:

```
(defmethod infer :importance [_ prog value & {}]
  (letfn
    [(sample-seq []
       (lazy-seq
         (cons
           (:state (exec ::algorithm prog value
                         initial-state))
           (sample-seq))))]
    (sample-seq)))
```

Default checkpoint handlers are called:

```
(defmethod checkpoint [::algorithm anglican.trap.observe] [_ obs]
  #((:cont obs) nil (add-log-weight (:state obs)
                                    (observe* (:dist obs) (:value obs)))))

(defmethod checkpoint [::algorithm anglican.trap.sample] [_ smp]
  #((:cont smp) (sample* (:dist smp)) (:state smp)))
```

Other inference algorithms

- Implement `infer`.
- Redefine one or both checkpoints.
- An average implementation is ≈ 150 lines of source code.
- A dozen different inference algorithms are in the code base.
- Half of them are really useful.

Recap

- Anglican is integrated with Clojure.
- Shares syntax but alters semantics.
- Macro-compiled.
- Efficient.
- Makes implementing AND using inference easy.

Thank you!

Questions?