Implementing Inference Methods in Anglican

Jan-Willem van de Meent

Interface for Inference

```
(defquery one-flip [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Interface for Inference

```
(defquery one-flip [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Likelihood Weighting
(*implemented by hand*)

```
(defn importance-one-flip
  [outcome]
  (let [theta (sample* (beta 1 1))
        lp (observe* (flip theta) outcome)]
    {:log-weight lp
     :result theta
     :predicts []}))
```

Interface for Inference

```
(defquery one-flip [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Likelihood Weighting
(*implemented by hand*)

```
(defn importance-one-flip
  [outcome]
  (let [theta (sample* (beta 1 1))
        lp (observe* (flip theta) outcome)]
    {:log-weight lp
     :result theta
     :predicts []}))
```

Interface for Inference

```
(defquery one-flip [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Likelihood Weighting
(*implemented by hand*)

```
(defn importance-one-flip
  [outcome]
  (let [theta (sample* (beta 1 1))
        lp (observe* (flip theta) outcome)]
    {:log-weight lp
     :result theta
     :predicts []}))
```

Interface for Inference

```
(defquery one-flip [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Likelihood Weighting
(*implemented by hand*)

```
(defn importance-one-flip
  [outcome]
  (let [theta (sample* (beta 1 1))
        lp (observe* (flip theta) outcome)]
    {:log-weight lp
     :result theta
     :predicts []}))
```

Interface for Inference

```
(defquery one-flip [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

- *Language Runtime*
  All deterministic operations

- *Inference Back End*
  Implements sample and observe

Interface for Inference

```
(defquery one-flip [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Anglican Backend Implementation

- Repeat until finished:
    - Call exec to run program until next sample or observe
    - Perform algorithm-specific actions and continue

Interface for Inference

```
(defquery one-flip [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

```
(use '[anglican emit runtime inference state])
(exec :importance one-flip [true] initial-state)
```

Interface for Inference

```
(defquery one-flip [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))

(use '[anglican emit runtime inference state])
(exec :importance one-flip [true] initial-state)
```

Algorithm

Interface for Inference

```
(defquery one-flip [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

```
(use '[anglican emit runtime inference state])
(exec :importance one-flip [true] initial-state)
```

Query

Interface for Inference

```
(defquery one-flip [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

```
(use '[anglican emit runtime inference state])
(exec :importance one-flip [true] initial-state)
```

Argument values

Interface for Inference

```
(defquery one-flip [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

```
(use '[anglican emit runtime inference state])
(exec :importance one-flip [true] initial-state)
```

Execution state

Interface for Inference

```
(defquery one-flip [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

```
(use '[anglican emit runtime inference state])
(exec :importance one-flip [true] initial-state)
```

Program Execution

```
#anglican.trap.sample{:id S23882,
                      :dist (anglican.runtime/beta 1 1),
                      :cont #function[…],
                      :state {:log-weight 0.0,
                              :predicts [],
                              :result nil,
                              :anglican.state/mem {},
                              :anglican.state/store nil}}
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

Returns

```
{:id 'S24726
 :dist (beta 1 1)
 :cont (fn [theta $state]
            ...)
 :state $state}
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

Returns

```
{:id 'S24726
 :dist (beta 1 1)
 :cont (fn [theta $state]
           ...)
 :state $state}
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

Returns

```
{:id 'S24726
 :dist (beta 1 1)
 :cont (fn [theta $state]
          ...)
 :state $state}
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

Returns

```
{:id 'S24726
 :dist (beta 1 1)
 :cont (fn [theta $state]
         ...)
 :state $state}
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

Returns

```
{:id 'S24726
 :dist (beta 1 1)
 :cont (fn [theta $state]
         ...)
 :state $state}
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

```
(let [x (sample* dist)]
  (cont x $state))
```

Inference
Backend

Returns

```
{:id 'S24726
 :dist (beta 1 1)
 :cont (fn [theta $state]
         ...)
 :state $state}
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

```
(let [x (sample* dist)]
  (cont x $state))
```

Inference
Backend

Returns

```
{:id 'S24726
 :dist (beta 1 1)
 :cont (fn [theta $state]
         ...)
 :state $state}
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

Returns

```
{:id 'O24724
 :dist (flip theta)
 :value outcome
 :cont (fn [_ $state]
         (->result
           theta $state))
 :state $state}
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

Returns

```
{:id 'O24724
 :dist (flip theta)
 :value outcome
 :cont (fn [_ $state]
         (->result
           theta $state))
 :state $state}
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

Returns

```
{:id 'O24724
 :dist (flip theta)
 :value outcome
 :cont (fn [_ $state]
        (->result
          theta $state))
 :state $state}
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

Returns

```
{:id 'O24724
 :dist (flip theta)
 :value outcome
 :cont (fn [_ $state]
         (->result
           theta $state))
 :state $state}
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

Returns

```
{:id 'O24724
 :dist (flip theta)
 :value outcome
 :cont (fn [_ $state]
         (->result
           theta $state))
 :state $state}
```

Inference Backend

```
(let [lp (observe* dist value)]
  (cont nil (add-log-weight
              $state lp)))
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

Returns

```
{:id 'O24724
 :dist (flip theta)
 :value outcome
 :cont (fn [_ $state]
         (->result
          theta $state))
 :state $state}
```

Inference Backend

```
(let [lp (observe* dist value)]
  (cont nil (add-log-weight
              $state lp)))
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
      (->result theta $state))
    $state))
  $state))
```

```
(let [lp (observe* dist value)]
  (cont nil (add-log-weight
              $state lp)))
```

Inference
Backend

Returns

```
{:id 'O24724
 :dist (flip theta)
 :value outcome
 :cont (fn [_ $state]
         (->result
           theta $state))
 :state $state}
```

Interface for Inference

```
(query [outcome]
 (let [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))
```

Continuation Passing Style

```
(fn [outcome $state]
 (->sample 'S24726
  (beta 1 1)
  (fn [theta $state]
   (->observe 'O24724
    (flip theta)
    outcome
    (fn [_ $state]
     (->result theta $state))
    $state))
  $state))
```

Returns

```
{:result theta
 :log-weight
   (:log-weight $state)
 :predicts
   (:predicts $state)}
```

Likelihood Weighting

Implementation for **sample**

```
(let [x (sample* dist)]
  (cont x $state))
```

Implementation for **observe**

```
(let [lp (observe* dist value)]
  (cont nil (add-log-weight
              $state lp)))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))
```

(->sample …)

```
(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))
```

```
(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp)))))
```

Likelihood Weighting

```clojure
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))
```

```
(->observe …)
```

```
(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Likelihood Weighting

```
(defmulti checkpoint
 (fn [alg cpt] [alg (type cpt)]))

(defmethod checkpoint
  [:importance anglican.trap.sample] [alg smp]
  (let [cont (:cont smp)
        x (sample* (:dist smp))
        state (:state smp)]
    (fn [] (cont x state))))

(defmethod checkpoint
  [:importance anglican.trap.observe] [alg obs]
  (let [cont (:cont obs)
        lp (observe* (:dist obs) (:value obs))
        state (:state obs)])
    (fn [] (cont nil (add-log-weight state lp))))
```

Implementation of exec

exec: *calls* checkpoint *to handle interrupts*

```
(defn exec
  "executes the program, calling checkpoint handlers
  at the checkpoints and stopping when the handler
  returns a non-callable value"
  [algorithm prog value state]
  (loop [step (trampoline prog value state)]
    (let [next (checkpoint algorithm step)]
      (if (fn? next)
        (recur (trampoline next))
        next))))
```

Likelihood Weighting

infer: *calls* exec *to construct sample sequence*

```
(defmulti infer
  (fn [alg prog value & _] alg))

(defmethod infer :importance
  [alg prog value & opts]
  (letfn [(sample-seq []
            (let [result (exec ::algorithm
                               prog
                               value
                               initial-state)]
                (cons (:state result)
                      (sample-seq))))]
    (sample-seq)))
```

Likelihood Weighting

infer: *calls* exec *to construct sample sequence*

```
(defmulti infer
  (fn [alg prog value & _] alg))

(defmethod infer :importance
  [alg prog value & opts]
  (letfn [(sample-seq []
            (let [result (exec ::algorithm
                               prog
                               value
                               initial-state)]
                (cons (:state result)
                      (sample-seq))))]
    (sample-seq)))
```

Likelihood Weighting

infer: *calls* exec *to construct sample sequence*

```
(defmulti infer
  (fn [alg prog value & _] alg))

(defmethod infer :importance
  [alg prog value & opts]
  (letfn [(sample-seq []
            (let [result (exec ::algorithm
                               prog
                               value
                               initial-state)]
                (cons (:state result)
                      (sample-seq))))]
    (sample-seq)))
```

Likelihood Weighting

infer: *calls* exec *to construct sample sequence*

```
(defmulti infer
  (fn [alg prog value & _] alg))

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

Likelihood Weighting

infer: *calls* exec *to construct sample sequence*

```
(defmulti infer
  (fn [alg prog value & _] alg))

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

Likelihood Weighting

infer: *calls* exec *to construct sample sequence*

```
(defmulti infer
  (fn [alg prog value & _] alg))

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

Likelihood Weighting

infer: *calls* exec *to construct sample sequence*

```
(defmulti infer
  (fn [alg prog value & _] alg))

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

Likelihood Weighting

infer: *calls* exec *to construct sample sequence*

```
(defmulti infer
  (fn [alg prog value & _] alg))

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

Likelihood Weighting

infer: *calls* exec *to construct sample sequence*

```
(defmulti infer
  (fn [alg prog value & _] alg))

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

doquery: *wrapper around* infer

Algorithm Implementations

15+ algorithms, ~180 lines of code per algorithm on average

Algorithm	Type	Lines	Citation	Description
smc	IS	127	Wood et al. AISTATS, 2014	Sequential Monte Carlo
importance	IS	21		Likelihood weighting
pcascade	IS	176	Paige et al., NIPS, 2014	Particle cascade
bbvb	IS	480	van de Meent et al., AISTATS, 2016	Black Box Variational Inference
ipmcmc	PMCMC	198	Rainforth et al., ICML, 2016	Interacting Particle Markov Chain Monte Carlo
pgibbs	PMCMC	121	Wood et al. AISTATS, 2014	Particle Gibbs (iterated conditional SMC)
pimh	PMCMC	68	Wood et al. AISTATS, 2014	Particle independent Metropolis-Hastings
pgas	PMCMC	179	van de Meent et al., AISTATS, 2015	Particle Gibbs with ancestor sampling
lmh	**MCMC**	**177**	**Wingate et al., AISTATS, 2011**	**Lightweight Metropolis-Hastings**
almh	MCMC	320	Tolpin et al., ECML PKDD, 2015	Adaptive scheduling lightweight Metropolis-Hastings
rmh	MCMC	377	-	Random-walk Metropolis-Hastings
palmh	MCMC	66	-	Parallelised adaptive scheduling lightweight MH
plmh	MCMC	62	-	Parallelised lightweight Metropolis-Hastings
bamc	MAP	318	Tolpin et al., SoCS, 2015	Bayesian Ascent Monte Carlo
siman	MAP	193	Tolpin et al., SoCS, 2015	MAP estimation via simulated annealing