Implementing Inference Methods in Anglican

Jan-Willem van de Meent
(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 []}))
(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

(implimented 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

(\texttt{defquery one-flip [outcome]}
 (\texttt{let [theta (sample (beta 1 1))]}  
 (\texttt{observe (flip theta) outcome)}
 theta))

\textit{Anglican Backend Implementation}

- Repeat until finished:
  - Call \texttt{exec} to run program until next \texttt{sample} or \texttt{observe}
  - Perform algorithm-specific actions and continue
Interface for Inference

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

(use 'anglican-emitter-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)

#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}}
(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))

(fn [outcome] [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))

(fn [outcome] [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))

(fn [outcome] [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))

(fn [outcome] [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))

(fn [outcome] [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))

(fn [outcome] [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))

(fn [outcome] [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))

(fn [outcome] [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))

(fn [outcome] [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))

(fn [outcome] [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))

(fn [outcome] [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))

(fn [outcome] [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))

(fn [outcome] [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))

(fn [outcome] [theta (sample (beta 1 1))]
  (observe (flip theta) outcome)
  theta))

Continuation Passing Style
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}
(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))
  $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}
(query [outcome]
  (let [theta (sample (beta 1 1))]
    (observe (flip theta) outcome)
    theta))

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

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

{:
  :id 'S24726
  :dist (beta 1 1)
  :cont (fn [theta $state] ...
    :state $state}
### Interface for Inference

\[
\begin{align*}
\text{(query [outcome]}
\text{(let [theta (sample (beta 1 1))])}
\text{(observe (flip theta) outcome)}
\text{theta}))
\end{align*}
\]

**Continuation Passing Style**

\[
\begin{align*}
\text{(fn [outcome $state]}
\text{(->sample 'S24726}
\text{(beta 1 1)}
\text{(fn [theta $state]}
\text{(->observe 'O24724}
\text{(flip theta)}
\text{outcome}
\text{(fn [__ $state]}
\text{(->result theta $state)})
\text{$state))}
\text{$state))}
\end{align*}
\]

**Inference Backend**

\[
\begin{align*}
\text{(let [x (sample* dist)]}
\text{(cont x $state))}
\end{align*}
\]

**Returns**

\[
\begin{align*}
\{ :id 'S24726
\text{:dist (beta 1 1)}
\text{:cont (fn [theta $state]}
\text{...)}
\text{:state $state}\}
\end{align*}
\]
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 '024724
 :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 '024724 :
  :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 '024724
  :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))

Returns

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

{:id '024724
 :dist (flip theta)
 :value outcome
 :cont (fn [__ $state]
     (->result
       theta $state))
     :state $state}
(query [outcome]
  (let [theta (sample (beta 1 1))]
    (observe (flip theta) outcome theta))
)

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

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 '024724
 :dist (flip theta)
 :value outcome
 :cont (fn [_ $state]
              (->result
                theta $state))
 :state $state}
(query [outcome]
  (let [theta (sample (beta 1 1))]
    (observe (flip theta) outcome)
    theta))

Continuation Passing Style

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

Inference Backend

(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 '024724
 :dist (flip theta)
 :value outcome
 :cont (fn [__ $state]
            (-> result
                theta $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 '024724
        (flip theta)
        outcome
        (fn [__ $state]
          (->result theta $state))
        $state))
    $state))

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

(return)

{:id '024724
 :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 $state
 :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)))
Implementation of infer

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

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

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

**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 value initial-state)]
               (cons (:state result)
                     (sample-seq)))]
   (sample-seq)))
```
Implementation of `infer`

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

```lisp
(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)))
```
Implementation of `infer`

`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)))]
```
Implementation of \texttt{infer}

\texttt{infer}: \textit{calls exec to construct sample sequence}

\begin{verbatim}
(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))))
\end{verbatim}
Implementation of infer

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 value initial-state)]
                   (lazy-seq
                     (cons (:state result) (sample-seq)))))])
  (sample-seq)))
Implementation of \texttt{infer}

\texttt{infer}: \textit{calls exec to construct sample sequence}

\begin{verbatim}
(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))
\end{verbatim}
Implementation of `infer`

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

```clojure
(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)))
```
Implementation of **infer**

**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*
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))))
Implementation of checkpoint

(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 `checkpoint`

```
(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 `checkpoint`

```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)))))
```
Implementation of `checkpoint`

```
(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 `checkpoint`

```
(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 **checkpoint**

```
(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 `checkpoint`

```
(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 `checkpoint`

```lisp
(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 `checkpoint`

```lisp
(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 *checkpoint*

```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))))
```
Implementation of checkpoint

```
(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 `checkpoint`

```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)))))
Implementation of **checkpoint**

```lisp
(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 \texttt{checkpoint}
Implementation of **checkpoint**

```
(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 `checkpoint`

```
(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 `checkpoint`

```lisp
(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 `checkpoint`

```lisp
(defun multi checkpoint
  (fn [alg cpt] [alg (type cpt)]))

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

(defun method 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))))
```
# Algorithm Implementations

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

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