`churchscm`: Probabilistic Programming within MIT Scheme

Michael Behr

May 12, 2014

1 Introduction

`churchscm` is an implementation of the Church language within MIT Scheme. Church is a probabilistic programming language, allowing users to represent and sample from probability distribution with all the expresiveness of a programming language [2]. Most current implementations of Church are implemented as separate languages; by instead defining Scheme operators, `churchscm` is compatible with all existing features of MIT Scheme. `churchscm`'s supported sampling mechanisms are rejection sampling and Metropolis-Hastings sampling.

2 Usage

A Church program consists of two parts. First, the user must specify the distribution to be sampled from. The distribution should be represented as a function with no arguments that draws a value from the distribution. The `observe` and `constrain` operators can be used to condition the distribution on observations. All randomness should come from `churchscm`'s probabilistic operators, as described below; aside from these operators, the function should be purely functional. If side effects or other sources of randomness are used, the results of sampling are undefined.

Second, once the distribution is defined, a sampling operator can be called to draw samples from the distribution. See section 4 for information on the different samplers and their parameters.

Below is an example program in `churchscm`: it flips two coins, observes that exactly one of the coins is heads, and returns a list of numbers corresponding to the two coins. The following query draws 20 samples from this distribution:

```
(rejection-query
 20 200
 (lambda ()
   (let ((result1 (flip))
         (result2 (flip)))
     (observe (= (+ result1 result2) 1))
     (list result1 result2))))
```

More examples can be found in `test.scm`.

3 Probabilistic operators

Probabilistic programs should not use existing sources of randomness directly: in order for the sampler to properly control the flow of execution, randomness should come from *probabilistic operators* specific to this system. These probabilistic operators can be created and used by means of the procedures below.

`(make-operator sampler logmass-function)`

Creates a probabilistic operator.

`sampler` should be a procedure that takes any number of arguments and returns a sample from the distribution that the operator represents. Its arguments will correspond to the arguments of the probabilistic operator. It is not currently supported for a sampler to call any other probabilistic operators. Instead, either use a regular procedure to call many probabilistic operators, or have the sampler generate many random values.

`logmass-function` should be a procedure that takes the same set of arguments as `sampler` and returns another procedure. This procedure should take a single argument, corresponding to a point in the probability space of the distribution, and return the natural logarithm of the mass at that point. Exact continuous distributions are not supported, but they can be approximated using `logdensity->logmass`.

`sampler` and `logmass-function` must represent the same probability distribution, or inference may not be accurate.

`(named-operator operator name)`

Gives a name to a probabilistic operator. The resulting operator will have the same semantics as the original operator, except that samplers will be able to recognize that it is the same operator across alternate histories of the computation. This allows for more efficient sampling; see the section on Metropolis-Hasting sampling for details.

`operator` is the probabilistic operator to be named. It is not mutated; a new copy is returned. `operator` must not have been constrained with `constrain`, or inference will be inaccurate. (In the currently implemented samplers, there is never a need to have an operator with both a name and a constraint; if this is desired, see the `pspec->operator` procedure.)

`name` is the name of the new operator. Two operators are considered to be the same across alternate histories if their names are the same when compared using `eq?`. No two operators in the same history should have the same name, or behavior will be undefined. However, if two operators in different histories have the same name, it is not necessary that they share any other similarities.

`(mem operator)`

Creates an operator that returns the same value whenever it is called with the same arguments multiple times within the same history of a computation. Argument lists are compared with `equal?`.

`(observe observed)`

Condition the distribution on an observation. `observed` should be a boolean value; computations in which it is false are inconsistent, and they will not be sampled from.

`(constrain operator value)`

Condition the distribution on the observation that a certain probabilistic operator takes on a certain value. `operator` should be a probabilistic operator and `value` a value it might return. The constrained operator will always return `value` when sampled from, regardless of its arguments, but its mass at `value` will be the same as the original's mass. The sampler will use this mass to ensure that this constraint does not bias the samples drawn. Depending on how `operator`'s samples can be compared for equality, this will often specify the same distribution as sampling from `operator` and conditioning on its result being `value` using `observe`; however, it may be more efficient, because the sampler can entirely neglect the inconsistent histories where the result is not `value`.

`(mass->logmass mass)`

Converts probability mass into its natural logarithm. Identical to `log`, except that `(mass->logmass 0)` returns $-\infty$ instead of an error.

`(logdensity->logmass logdensity x)`

Approximates the density of a continuous distribution as the mass of a discrete distribution. `x` is the value at which the density is computed, and `logdensity` is the natural logarithm of the density at `x`. The mass of the discrete distribution depends on the density of the continuous distribution and the precision with which `x` is represented in memory.

A few sample probabilistic operators are defined.

`(flip)`

Simulates a fair coin flip: returns 1 with probability 1/2 and 0 with probability 1/2.

`(bernoulli p)`

The Bernoulli distribution with parameter `p`: returns 1 with probability `p` and 0 with probability $1 - \texttt{p}$.

`(cont-uniform a b)`

The continuous uniform distribution on $[\texttt{a}, \texttt{b})$. Returns a value between `a` inclusive and `b` exclusive. Requires $\texttt{a} < \texttt{b}$.

`(normal mean stdev)`

The normal distribution with mean `mean` and standard deviation `stdev`.

4 Sampling

`(rejection-query samples cutoff thunk)`

Draw samples from `thunk` using rejection sampling. `thunk` should be a procedure taking no arguments. `samples` determines the number of samples to be returned. At most `cutoff` samples will be considered: if `cutoff` is too low and the distribution yields too many infeasible samples, the query may terminate early and return fewer samples than requested.

Rejection sampling is a straightforward method of sampling. A sample is drawn by performing the supplied computation. If the computation's history is inconsistent with the supplied observations, the sample is discarded; otherwise, it is recorded. If the sampler evaluates an operator that is constrained to return a specific value, the sample is discarded depending on the mass of that operator at that value. Rejection sampling is not recommended except for simple models with few observations, because the time that it takes to draw a sample increases with the probability that a sample will be rejected.

`(mh-query nsamples burn-in lag thunk)`

Draw samples from `thunk` using the Metropolis-Hastings algorithm. `thunk` should be a procedure taking no arguments. `nsamples` specifies the total number of samples to be returned. `burn-in` specifies the number of samples to be drawn before the first sample is recorded; the higher this value, the longer sampling will take, but the closer the algorithm will be to the correct distribution when samples are drawn. If the burn-in period is too short, early samples may be inconsistent with the computation's calls to `observe`, though observations from `constrain` will always be respected. `lag` is a positive integer specifying the number of samples that should be drawn for each sample recorded. If the lag is too low, nearby samples may be correlated.

The Metropolis-Hastings algorithm is a Markov chain Monte Carlo algorithm: it randomly walks through possible histories of the computation and draws samples as it walks. As long the computation obeys all the restrictions outlined in this document, the distribution over samples will eventually converge to the desired distribution. The `burn-in` parameter gives the distribution time to converge before samples are recorded, but it may not be easy to predict how many steps are needed. Furthermore, because the algorithm is a Markov chain, any given state will be highly correlated with the previous state even if the overall distribution has converged; the `lag` parameter can correct for this, but it may not be easy to predict how high it should be to make samples sufficiently uncorrelated.

Setting `burn-in` and `lag` higher than necessary will not decrease the fidelity of the samples drawn, but it will increase the number of samples that must be drawn and therefore the runtime of the algorithm. It may be necessary to try different values of `burn-in` and `lag` until acceptable results are given. Unlike rejection sampling, the sampler's runtime does not depend on the rate at which samples are accepted: the number of samples drawn will always be `burn-in` + `nsamples` ∗ `lag`.

The Metropolis-Hastings algorithm uses a proposal distribution $q(x \to x')$ in order to walk across the space of possible states. To choose the next state, a sample is drawn from this distribution, and the sample is accepted as the next state with probability determined by the acceptance ratio α:

$$\alpha = \min\left(1, \frac{p(x')q(x' \to x)}{p(x)q(x \to x')}\right)$$

In the current implementation, the proposal distribution is as follows: from the current history of the computation, uniformly choose a probabilistic operator that is not constrained to be a given value. Discard the chosen operator's old value and sample a new value (which may be the same as the old value), then resume the computation from that operator. When any other probabilistic operators are encountered, check to see if they share a name with any operators from the previous history. If so, it may be possible to reuse the operator's old value and thus decrease the variance of the proposal distribution, but note that the operator may be called with different arguments, and so it may no longer be possible for it to return the old value. Therefore, check to see whether the mass of the old value with the new arguments is zero before reusing the value. This choice of proposal distribution makes the acceptance ratio relatively easy to compute, but in future work, other distributions may allow for faster convergance: see section 6.

For more information on Church sampling using Metropolis-Hastings, see [1], Chapter 7.

No sampler

When there is no current sampler, calling a probabilistic operator will simply draw and return a sample from its sampling procedure.

5 Defining new samplers

It is possible to define other samplers. To do so, bind the following variables, which should have dynamic extent encompassing the computation to be sampled from:

`*sampler-state*`

A value representing the state of the sampler. There are no requirements for what this value should be; it is only interacted with by procedures specific to the sampler.

`(add-to-sampler operator-instance)`

A procedure that is called whenever a probabilistic operator is evaluated. `operator-instance` is a value of the structure type `prob-operator-instance`, containing information about the operator: its slots are described later in this section.

`(observe observed)`

Condition the distribution on an observation. `observed` should be a boolean value; computations in which it is false are inconsistent, and they will not be sampled from.

`(sampler-computation-state)`

Returns a value representing the current history of the computation. This value must give the proper result when passed to `sampler-in-computation?`, as described below; there are no other requirements for what the value must be.

`(sampler-in-computation? state)`

Tests whether a given state of the computation is in the history of the current state of the computation. `state` will be a value that was returned by a call to `sampler-computation-state`; this should return `#t` if the call to `sampler-computation-state` occurred in the same history as the current call to `sampler-in-computation`, and `#f` otherwise.

When a sample is to be drawn from a probabilistic operator, `add-to-sampler` is passed a value of the structure type `prob-operator-instance`. This structure has the following slots:

`pspec`

A value containing the sampling procedure and the log-mass procedure for the operator. It is of the structure type `pspec`, which has the slots `sampler`, giving the sampling procedure, and `logmass-function`, giving the log-mass procedure. These procedures have the same properties as described in the documentation for `make-operator`.

`continuation`

The contination of the probabilistic operator when it was called. Passing a value to this continuation will resume the computation, returning that value from the operator.

`args`

The argument list of the probabilistic operator. Calling the operator's sampler procedure with this list (as with `apply`) will draw a sample from its distribution. Calling its log-mass function with this list (again as with `apply`) will yield a function mapping a value from the distribution's probability space to the natural logarithm of the distribution's mass at that value.

`name`

The name of the operator, if it has one; the empty list if it does not. If two operators in different histories both have names, they should be considered the same operator for the purposes of sampling if they are equal when compared with `eq?`.

`constrained?`

A boolean denoting whether the operator has been constrained to return a specific value, regardless of its original distribution.

6 Future work

`churchscm` serves as a proof of concept to show that probabilistic programming is possible within an existing Scheme implementation. Before it can be suitable for general use, several improvements are needed.

Automatic naming To allow for efficient sampling with Metropolis-Hastings, it is necessary to manually supply names for distributions. This is a hassle, especially when recursion is needed, because distinct names are needed at each level of recursion. Supplying these names manually should be possible, but it would be better for the library to provide names itself. See [3] for more discussion on naming.

Nested sampling Support for nested sampling is currently incomplete: it is possible with rejection sampling, but not with Metropolis-Hastings. Nested sampling is desirable for programs representing inference about inference; for more, see [1], Chapter 6.

More powerful proposal distributions The mixing time of the Metropolis-Hastings algorithm depends largely on the variance of its proposal distribution. Too small a distribution, and the algorithm will become stuck at local maxima; too large, and it will not hit local maxima enough. To improve inference, future work should allow more fine-grained customization of the proposal distribution, including the option to choose an operator and sample a new value close to its previous value.

Robust debugging and debugging tools The nature of probabilistic programming makes it easy for subtle bugs to arise, both in Church programs and in the implementation itself. Not only are the results of interesting

computations nondeterministic, but there may be no easy way to tell how long sampling should take to converge, leaving ambiguity about any errors come from bugs or simply from difficulties converging. Future work should introduce robust debugging tools to mitigate these difficulties as much as possible.

References

[1] N. D. Goodman and J. B. Tenenbaum. Probabilistic models of cognition. `https://probmods.org/`.

[2] Noah Goodman, Vikash Mansinghka, Daniel Roy, Keith Bonawitz, and Daniel Tarlow. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

[3] David Wingate, Andreas Stuhlmller, and Noah D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *International Conference on Artificial Intelligence and Statistics*, page 770778, 2011.

7 Appendix: Code listings

At the time of writing, these files are also available at `http://web.mit.edu/mkbehr/Public/churchscm`.

7.1 `load.scm`

```
(load "church-base.scm")
(load "distributions.scm")
(load "rejection.scm")
(load "metropolis-hastings.scm")
(load "util.scm")
```

7.2 `church-base.scm`

```
(define (mass->logmass mass)
  (if (= mass 0)
      -inf
      (log mass)))
(define logmass->mass exp)

(define *pspec-table* (make-weak-eq-hash-table))

(define-structure
  pspec
  sampler
  logmass-function)

;; Note: it is not currently supported to nest calls to other
```

```
;; probabilistic operators within an operator's sampler.

(define (pspec-sample spec #!optional args)
  (apply (pspec-sampler spec)
         (if (default-object? args)
             '()
             args)))
(define (pspec-logmass spec args x)
  ((apply (pspec-logmass-function spec) args) x))
(define (pspec-mass spec args x)
  (logmass->mass (pspec-logmass spec args x)))

;; Note: never make one of these without also populating *pspec-table*
(define-structure
  (prob-operator-instance
   (constructor make-prob-operator-instance
                (pspec continuation args
                       #!optional name constrained?)))
  pspec
  continuation
  args
  (name '())
  (constrained? #f))

(define (instance-sample instance)
  (pspec-sample (prob-operator-instance-pspec instance)
                (prob-operator-instance-args instance)))

(define (instance-logmass instance x)
  (pspec-logmass (prob-operator-instance-pspec instance)
                 (prob-operator-instance-args instance)
                 x))

(define (instance-mass instance x)
  (logmass->mass (instance-logmass instance x)))

(define (pspec->operator pspec #!optional name constrained?)
  (let ((out
         (lambda args
           (call-with-current-continuation
            (lambda (k)
              (add-to-sampler
               (make-prob-operator-instance
                ;; this does the right thing when some args aren't
                ;; supplied
                pspec k args name constrained?)))))))
    (hash-table/put! *pspec-table* out pspec)
    out))

(define (make-operator args)
  (pspec->operator (apply make-pspec args)))

;; Note: right now (named-operator (constrained foo)) doesn't do what
;; you might expect
(define (named-operator operator name)
  (pspec->operator
   (hash-table/get *pspec-table* operator '())
```

```
   name))

;; A value tied to a certain branch of the computation. Once the
;; sampler is no longer exploring the branch, the recorded value is no
;; longer meaningful.
(define-structure
  (local-computation-record
   (constructor make-local-computation-record (value)))
  value
  (computation-state (sampler-computation-state)))

(define (local-computation-record-stale? record)
  (not (sampler-in-computation?
        (local-computation-record-computation-state record))))

(define (mem f)
  (let ((mem-table (make-equal-hash-table)))
    (lambda args
      (let* ((default-value (list 'default))
             (recorded-out
              (hash-table/get mem-table args default-value)))
        (if (or
             (eq? recorded-out default-value)
             (local-computation-record-stale? recorded-out))
            (let ((value (apply f args)))
              (hash-table/put! mem-table args
                               (make-local-computation-record value))
              value)
            (local-computation-record-value recorded-out))))))


(define add-to-sampler
  (lambda (operator-instance)
    ((prob-operator-instance-continuation operator-instance)
     (instance-sample operator-instance))))

(define *sampler-state* #f)

(define (observe observed)
  (if observed
      'ok
      (error "Inconsistent observation outside of sampling")))

(define (constrain operator value)
  (pspec->operator
   (make-pspec
    (lambda args value)
    (pspec-logmass-function
     (hash-table/get *pspec-table* operator '())))
   '()
   #t))

(define *null-computation-state* (list '*null-computation-state*))

(define sampler-computation-state
  (lambda () *null-computation-state*))
```

```
(define sampler-in-computation?
  (lambda (state) (eq? state *null-computation-state*)))
```

7.3 util.scm

```
(define *pi* (acos -1))
(define +inf (/ 1.0 0.0))
(define -inf (/ -1.0 0.0))
(define NaN (+ +inf -inf))

(define (equal-histogram samples)
  (define hist-table (make-equal-hash-table))
  (define not-in-table (list 'not-in-table))
  (for-each
   (lambda (sample)
     (if (eq? (hash-table/get hist-table sample not-in-table)
              not-in-table)
         (hash-table/put! hist-table sample 1)
         (hash-table/put!
          hist-table sample
          (+ 1 (hash-table/get hist-table sample not-in-table)))))
   samples)
  (hash-table->alist hist-table))
```

7.4 distributions.scm

```
;;; Discrete distributions

;; Simulates a fair coin flip: returns 1 with probability 1/2 and 0
;; with probability 1/2.
(define flip
  (make-operator
   (lambda () (random 2))
   (lambda ()
     (lambda (x)
       (mass->logmass
        (cond ((= x 1) 1/2)
              ((= x 0) 1/2)
              (else 0)))))))

;; Bernoulli distribution: returns 1 with probability p and 0 with
;; probability 1-p.
(define bernoulli
  (make-operator
   (lambda (p)
     (if (<= (random 1.0) p)
         1
         0))
   (lambda (p)
     (lambda (x)
       (mass->logmass
        (cond ((= x 1) p)
              ((= x 0) (- 1 p))
              (else 0)))))))

;;; "Continuous" distributions
```

```
;; These distributions are in fact implemented as discrete
;; distributions, based on the machine's precision.


(define (logdensity->logmass logdensity x)
  (let ((epsilon
         ;; This calculation adapted from gjs's scmutils: find the
         ;; smallest number that makes a difference when added to x
         (if (= x (+ 1. x))
             (let loop ((e 1.0))
               (if (= x (+ e x))
                   (loop (* e 2))
                   e))
             (let loop ((e 1.0))
               (if (= x (+ (/ e 2) x))
                   e
                   (loop (/ e 2)))))))
    (+ logdensity (log epsilon))))
(define density->logdensity mass->logmass)

;; Uniform distribution: returns a value between a inclusive and b
;; exclusive. Assumes a < b.
(define cont-uniform
  (make-operator
   (lambda (a b)
     (+ (* (random 1.0)
           (- b a))
        a))
   (lambda (a b)
     (lambda (x)
       (if (and (>= x a) (< x b))
           (logdensity->logmass
            (- (log (- b a)))
            x)
           -inf)))))

;; Normal distribution with mean of mean and standard deviation of
;; stdev.
(define normal
  (make-operator
   (lambda (mean stdev)
     ;; draw from standard normal distribution using marsaglia polar
     ;; method
     ;; note: if it is desired to draw as few random numbers as
     ;; possible, this can be rewritten to generate two standard
     ;; normal deviates instead of one, the other being
     ;; (* v (sqrt (/ (* -2 (log s)) s)))
     (let lp ((u (- 1 (random 2.0)))
              (v (- 1 (random 2.0))))
       (let ((s (+ (square u) (square v))))
         (if (>= s 1.)
             (lp (- 1 (random 2.0)) (- 1 (random 2.0)))
             (let ((z (* u (sqrt (/ (* -2 (log s)) s)))))
               (+ mean (* z stdev)))))))
   (lambda (mean stdev)
     (lambda (x)
       (logdensity->logmass
```

```
        (density->logdensity
         (/
          (exp (-
                (/ (square (- x mean))
                   (* 2 (square stdev)))))
          (* stdev (sqrt (* 2 *pi*)))))
        x)))))
```

7.5 rejection.scm

```
(define-structure
  (rejection-state
   (constructor make-rejection-state
                (nsamples cutoff output-continuation)))
  (top-level '())
  (samples '())
  nsamples
  cutoff
  (computation-state (generate-uninterned-symbol))
  output-continuation)

(define (rejection-add-to-sampler operator-instance)
  (if (null? (rejection-state-top-level *sampler-state*))
      (set-rejection-state-top-level!
       *sampler-state*
       operator-instance))
  (let ((value (instance-sample operator-instance)))
    (if (and (prob-operator-instance-constrained? operator-instance)
             (< (instance-mass instance value)
                (random 1.0)))
        (rejection-reject)
        ((prob-operator-instance-continuation operator-instance)
         value))))

(define (rejection-return)
  ((rejection-state-output-continuation *sampler-state*)
   ;; samples are collected in reverse order, so put them back before
   ;; returning
   (reverse (rejection-state-samples *sampler-state*))))

(define (rejection-state-add-sample! state sample)
  (set-rejection-state-samples!
   state
   (cons sample
         (rejection-state-samples state)))
  (set-rejection-state-nsamples!
   state
   (- (rejection-state-nsamples state) 1))
  (set-rejection-state-cutoff!
   state
   (- (rejection-state-cutoff state) 1)))

(define (rejection-reject)
  (set-rejection-state-cutoff!
   *sampler-state*
   (- (rejection-state-cutoff *sampler-state*) 1))
  (if (<= ;; have we tried too many times?
```

```
       (rejection-state-cutoff *sampler-state*)
       0)
      (rejection-return)
      (rejection-restart)))

(define (rejection-observe observed)
  (if observed
      'ok
      (rejection-reject)))

(define (rejection-computation-state)
  (rejection-state-computation-state *sampler-state*))

(define (rejection-in-computation? state)
  (if (eq? state (sampler-computation-state))
      #t
      ;; If e.g. a mem table entry wasn't defined in our current
      ;; branch, we may still be doing nested queries, so check to see
      ;; if it would have been in our current branch on the next
      ;; sampling level up. The output continuation should contain all
      ;; the information we need in order to do that.
      (call-with-current-continuation
       (lambda (k)
         (within-continuation
          (rejection-state-output-continuation *sampler-state*)
          (lambda () (k (sampler-in-computation? state))))))))

(define (rejection-restart)
  (let ((top-level (rejection-state-top-level *sampler-state*)))
    (set-rejection-state-computation-state!
     *sampler-state* (generate-uninterned-symbol))
    ((prob-operator-instance-continuation top-level)
     (instance-sample top-level))))

(define (rejection-query samples cutoff thunk)
  (if (or (<= samples 0) (<= cutoff 0))
      '()
      (call-with-current-continuation
         (lambda (out)
           (fluid-let ((*sampler-state*
                        (make-rejection-state samples cutoff out))
                       (add-to-sampler
                        rejection-add-to-sampler)
                       (observe
                        rejection-observe)
                       (sampler-computation-state
                        rejection-computation-state)
                       (sampler-in-computation?
                        rejection-in-computation?))
             (let lp ((sample (thunk)))
               (rejection-state-add-sample! *sampler-state* sample)
               (if (or
                    (<= (rejection-state-nsamples *sampler-state*) 0)
                    (<= (rejection-state-cutoff *sampler-state*) 0))
                   (rejection-return)
                   (if (null?
                        (rejection-state-top-level *sampler-state*))
```

```
                    (lp (thunk))
                    (rejection-restart))))))))))
```

7.6 metropolis-hastings.scm

```
(define *rejected-samples* 0)
(define *bad-proposals* 0)

(define-structure
  (mh-computation-node
   (constructor make-mh-computation-node
                (operator-instance
                 operator-value
                 log-likelihood
                 infeasible?
                 becomes-infeasible?
                 #!optional reused-value)))
  operator-instance
  operator-value
  log-likelihood
  infeasible?
  becomes-infeasible?
  (reused-value #f))

(define (make-empty-mh-computation-node)
  (make-mh-computation-node
   '() '() '() '() '() '()))

(define-structure
  (mh-state
   (constructor make-mh-state%
                (nsamples burn-in lag output-continuation
                 computation-root computation-path)))
  nsamples
  burn-in
  lag
  output-continuation
  computation-root
  (samples '())
  (samples-drawn 0)
  (samples-recorded 0)
  (common-ancestor)
  ;; mc-value-state and mc-computation-state store the last value
  ;; accepted in the markov chain and its corresponding computation
  ;; path.
  (mc-value-state)
  (mc-computation-state)
  (mc-becomes-infeasible? #f)
  ;; computation-path is a stack of mh-computation-nodes, with the
  ;; special computation-root node at the bottom. Note: this stack
  ;; should /not/ be destructively modified, but rebinding
  ;; computation-path in order to push and pop is fine.
  computation-path
  (becomes-infeasible? #f)
  (named-operator-values (make-weak-eq-hash-table)))

(define (make-mh-state nsamples burn-in lag output-continuation)
```

```
  (let ((computation-root
         (make-mh-computation-node
          ;; operator-instance
          #f
          ;; operator-value
          #f
          ;; log-likelihood
          0.0
          ;; infeasible?
          #f
          ;; becomes-infeasible?
          #f)))
    (make-mh-state% nsamples burn-in lag output-continuation
                    computation-root (list computation-root))))

(define (mh-push-computation-node! mh-state node)
  (set-mh-state-computation-path!
   mh-state
   (cons node
         (mh-state-computation-path mh-state))))

(define (instance-resample? instance x)
  (= (instance-mass instance x) 0))

(define (mh-add-to-sampler operator-instance)
  (let* ((name (prob-operator-instance-name operator-instance))
         (default (list 'default))
         (lcrecord (hash-table/get
                    (mh-state-named-operator-values *sampler-state*)
                    name default))
         (resample?
          (cond
           ((null? name) #t) ;; no name means no stored value
           ((eq? lcrecord default) #t) ;; no entry in table
           ;; If the operator instance wasn't part of the previous
           ;; state's computation, then the recorded value is
           ;; stale. Note that mh-resume-with-value-thunk will update
           ;; the record if we don't resample, so we only have to
           ;; check the previous state.
           ((not (mh-in-computation?
                  (local-computation-record-computation-state
                   lcrecord)
                  (mh-state-mc-computation-state *sampler-state*)))
            #t) ;; stored value is from an old branch
           (else (instance-resample?
                  operator-instance
                  (local-computation-record-value lcrecord)))))
         (new-value-thunk
          (if resample?
              (lambda ()
                (instance-sample operator-instance))
              (begin
                ;; (pp "not resampling")
                (let ((new-value
                       (local-computation-record-value lcrecord)))
                  (lambda () new-value))))))
    (mh-resume-with-value-thunk
```

```
     operator-instance new-value-thunk (not resample?))))

(define (mh-resume-with-value-thunk
         operator-instance value-thunk #!optional reused?)
  (let ((node-parent
         (car (mh-state-computation-path *sampler-state*)))
        (computation-node (make-empty-mh-computation-node)))
    ;; Push the computation node before evaluating the value thunk,
    ;; because something like mem might want a reference to it
    (mh-push-computation-node! *sampler-state* computation-node)
    (let* ((operator-value (value-thunk))
           (log-likelihood
            (instance-logmass operator-instance operator-value))
           (becomes-infeasible?
            (mh-state-becomes-infeasible? *sampler-state*))
           (infeasible?
            (or becomes-infeasible?
                (mh-computation-node-infeasible? node-parent))))
      (set-mh-computation-node-operator-instance!
       computation-node operator-instance)
      (set-mh-computation-node-operator-value!
       computation-node operator-value)
      (set-mh-computation-node-log-likelihood!
       computation-node log-likelihood)
      (set-mh-computation-node-infeasible?!
       computation-node infeasible?)
      (set-mh-computation-node-becomes-infeasible?!
       computation-node becomes-infeasible?)
      (if (not (default-object? reused?))
          (set-mh-computation-node-reused-value!
           computation-node reused?))
      (set-mh-state-becomes-infeasible?!
       *sampler-state* #f)
      (let ((name (prob-operator-instance-name operator-instance)))
        (if (not (null? name))
            (hash-table/put!
             (mh-state-named-operator-values *sampler-state*)
             name
             (make-local-computation-record operator-value))))
      ((prob-operator-instance-continuation operator-instance)
       operator-value))))

(define (mh-return)
  ((mh-state-output-continuation *sampler-state*)
   ;; samples are collected in reverse order, so put them back before
   ;; returning
   (reverse (mh-state-samples *sampler-state*))))

(define (mh-state-add-sample! state sample)
  (set-mh-state-samples!
   state
   (cons sample
         (mh-state-samples state)))
  (set-mh-state-samples-recorded!
         *sampler-state*
         (+ 1 (mh-state-samples-recorded *sampler-state*))))
```

```
(define (mh-maybe-record! sample)
  ;; check burnout, lag, and sample index; add sample if match. Update
  ;; count of samples drawn. Do not resume; handle that elsewhere.
  ;; This also does not update the markov chain's state.
  (if (and (>= (mh-state-samples-drawn *sampler-state*)
               (mh-state-burn-in *sampler-state*))
           (= (modulo (- (mh-state-nsamples *sampler-state*)
                         (mh-state-burn-in *sampler-state*))
                      (mh-state-lag *sampler-state*)))
           (mh-state-add-sample! *sampler-state* sample))
  (set-mh-state-samples-drawn!
   *sampler-state*
   (+ 1 (mh-state-samples-drawn *sampler-state*))))

(define (mh-state-last-sample state)
  (if (null? (mh-state-samples state))
      (error "MH: Tried to get nonexistent last sample")
      (car (mh-state-samples state))))

(define (mh-observe observed)
  (if observed
      'ok
      (set-mh-state-becomes-infeasible?!
       *sampler-state* #t)))

(define (mh-computation-state)
  (mh-state-computation-path *sampler-state*))

(define (mh-in-computation? state computation)
  (let lp ((path computation))
    (cond
     ((eq? path state) #t)
     ((null? path)
      (call-with-current-continuation
       (lambda (k)
         (within-continuation
          (mh-state-output-continuation *sampler-state*)
          (lambda () (k (sampler-in-computation? state)))))))
     (else (lp (cdr path))))))

(define (mh-in-this-computation? state)
  (mh-in-computation?
   state
   (mh-state-computation-path *sampler-state*)))

(define (n-unconstrained-nodes path)
  (cond ((null? path) 0)
        ((and (mh-computation-node-operator-instance (car path))
              (not (prob-operator-instance-constrained?
                    (mh-computation-node-operator-instance
                     (car path)))))
         (+ 1 (n-unconstrained-nodes (cdr path))))
        (else (n-unconstrained-nodes (cdr path)))))

(define (mh-resample)
  (let ((n-resampling-choices
         (n-unconstrained-nodes
```

```
        (mh-state-computation-path *sampler-state*))))
    (cond
     ((= n-resampling-choices 0) ;; computation is deterministic
      (error
       "Can't resample from a deterministic computation"))
     (else
      (let* ((resample-index
              (random n-resampling-choices))
             (new-path-and-resample-node
              (let lp ((nodes-to-remove (+ resample-index 1))
                       (path (mh-state-mc-computation-state
                              *sampler-state*))
                       (last-resample-node '()))
                (cond
                 ((= nodes-to-remove 0)
                  (cons path last-resample-node))
                 ((prob-operator-instance-constrained?
                   (mh-computation-node-operator-instance (car path)))
                  (lp nodes-to-remove (cdr path) last-resample-node))
                 (else
                  (lp (- nodes-to-remove 1) (cdr path) (car path))))))
             (new-path (car new-path-and-resample-node))
             (resample-node (cdr new-path-and-resample-node))
             (resample-operator
              (mh-computation-node-operator-instance resample-node)))
        (set-mh-state-computation-path!
         *sampler-state* new-path)
        (set-mh-state-common-ancestor!
         *sampler-state* (car new-path))
        (set-mh-state-becomes-infeasible?!
         *sampler-state* #f)
        (mh-resume-with-value-thunk
         resample-operator
         (lambda ()
           (instance-sample resample-operator)))))))


(define (acceptance-ratio state)
  (cond ((null? (mh-state-samples state))
         ;; This is the initial sample; definitely accept it
         1)
        ((= (n-unconstrained-nodes (mh-state-computation-path state))
            0)
         ;; The computation state doesn't have any probabilistic
         ;; operators to resample from. It doesn't actually matter
         ;; whether we accept or reject here.
         1)
        ((or (mh-computation-node-infeasible?
              (car (mh-state-mc-computation-state state)))
             (mh-state-mc-becomes-infeasible? state))
         ;; Our previous state was infeasible. Just accept all
         ;; proposals until we find a feasible state. (Note that this
         ;; means that an infeasible sample may be recorded if the
         ;; burn-in period is too low and it is too hard to find an
         ;; initial sample)
         1)
        ((or (mh-computation-node-infeasible?
```

```
     (car (mh-state-computation-path state)))
    (mh-state-becomes-infeasible? state))
 ;; Our previous state was feasible, but our new state isn't,
 ;; so the transition probability is 0.
 0)
(else
 ;; Proceed as normal. The acceptance ratio depends on all the
 ;; nodes for which the proposal distribution differs from the
 ;; distribution to be sampled from, as well as the number of
 ;; nodes in each state's computational history.
 (let ((reused-names (make-eq-hash-table))
       (running-log-ratio 0))
   ;; reused-names will act as a hash set, containing the name
   ;; of every operator for which samples were reused. For the
   ;; current purposes, assume that if a sample is reused
   ;; while proposing x -> x', then that sample would also be
   ;; reused while proposing x' -> x.
   (let lp ((path
             (mh-state-computation-path state)))
     (if (eq? (car path)
              (mh-state-common-ancestor state))
         'done
         (begin
           (cond
            ((prob-operator-instance-constrained?
              (mh-computation-node-operator-instance
               (car path)))
             (let ((delta-logratio
                    (mh-computation-node-log-likelihood
                     (car path))))
               (set! running-log-ratio
                     (+ running-log-ratio delta-logratio))))
            ((mh-computation-node-reused-value (car path))
             (let ((name
                    (prob-operator-instance-name
                     (mh-computation-node-operator-instance
                      (car path))))
                   (delta-logratio
                    (mh-computation-node-log-likelihood
                     (car path))))
               (hash-table/put! reused-names name #t)
               (set! running-log-ratio
                     (+ running-log-ratio delta-logratio)))))
           (lp (cdr path)))))
   ;; now that reused-names is populated, find the
   ;; contributions from the original sample.
   (let lp ((path
             (mh-state-mc-computation-state state)))
     (if (eq? (car path)
              (mh-state-common-ancestor state))
         'done
         (begin
           (let ((name (prob-operator-instance-name
                        (mh-computation-node-operator-instance
                         (car path))))
                 (delta-logratio
                  (mh-computation-node-log-likelihood
```

```
                        (car path))))
                  (if (or (hash-table/get reused-names name #f)
                          (prob-operator-instance-constrained?
                           (mh-computation-node-operator-instance
                            (car path))))
                      (set! running-log-ratio
                            (- running-log-ratio delta-lograti o))))
                (lp (cdr path)))))
          (let ((length-contribution
                 (/ (n-unconstrained-nodes
                     (mh-state-mc-computation-state state))
                    (n-unconstrained-nodes
                     (mh-state-computation-path state)))))
            (* (logmass->mass running-log-ratio)
               length-contribution))))))

(define (mh-query nsamples burn-in lag thunk)
  (if (<= nsamples 0)
      '()
      (call-with-current-continuation
       (lambda (out)
         (fluid-let ((*sampler-state*
                      (make-mh-state nsamples burn-in lag out))
                     (add-to-sampler
                      mh-add-to-sampler)
                     (observe
                      mh-observe)
                     (sampler-computation-state
                      mh-computation-state)
                     (sampler-in-computation?
                      mh-in-this-computation?))
           ;; Draw a sample from the beginning. Then, repeatedly:
           ;; - choose a place to resample from
           ;; - resample from that place
           ;; - compute acceptance ratio
           ;; - accept or reject
           ;; - update state accordingly
           (let lp ((sample (thunk)))
             (let* ((alpha (acceptance-ratio *sampler-state*))
                    ;; note: (random 1.0) returns a number between
                    ;; zero inclusive and one exclusive. if alpha is 0
                    ;; and we draw a 0, we need to make sure to
                    ;; reject. So we accept if the acceptance ratio is
                    ;; strictly greater than the number drawn.
                    (accepted (< (random 1.0) alpha)))
               (if (= alpha 0)
                   (set! *bad-proposals* (+ *bad-proposals* 1)))
               (if accepted
                   (begin
                     (mh-maybe-record! sample)
                     (set-mh-state-mc-value-state!
                      *sampler-state* sample)
                     (set-mh-state-mc-computation-state!
                      *sampler-state*
                      (mh-state-computation-path *sampler-state*))
                     (set-mh-state-mc-becomes-infeasible?!
                      *sampler-state*
```

```
                (mh-state-becomes-infeasible? *sampler-state*)))
           (begin (mh-maybe-record! (mh-state-mc-value-state
                                     *sampler-state*))
                  (set! *rejected-samples*
                        (+ *rejected-samples* 1))))
      (cond
       ((>= (mh-state-samples-recorded *sampler-state*)
            (mh-state-nsamples *sampler-state*))
        (mh-return)) ; we're finished
       ((<=
         (length (mh-state-computation-path *sampler-state*))
         1)
        ;; no probabilistic operators, so no need to resample
        (lp sample))
       (else (mh-resample)))))))))
```

7.7 test.scm

```
;;; Tests for system without any inference running

(flip)
;; expected output: 1 or 0
(observe #t)
;; expected output: not an error
(observe #f)
;; expected output: error

;;; Continuous distributions

(cont-uniform 0 10)
;; Expected value: inexact number between 0 and 10
(pspec-logmass cont-uniform-spec (list 0 10) 8)
(pspec-logmass cont-uniform-spec (list 0 10) 9)
;; Expected value: the same small value
(pspec-logmass cont-uniform-spec (list 0 10) 20)
;; Expected value: -inf
(normal 0 1)
;; Expected value: sample from the standard normal distribution
(pspec-logmass normal-spec (list 0 1) 0)
;; Expected value: something small
(pspec-logmass normal-spec (list 0 1) 100)
;; Expected value: something extremely small (this will be -inf
;; barring a very precise computer)

;;; Tests for rejection sampling

(rejection-query
 400 4000
 (lambda ()
   (pp "This should only print once")
   (flip)))
;; expected output: print "This should only print once", return list
;; of 400 1s and 0s, evenly distributed

(equal-histogram
 (rejection-query
  400 4000
```

```
  (lambda ()
    (flip))))

(rejection-query
 10 100
 (lambda () 1))
;; expected output: list of 10 1

;;; Tests for rejection sampling with observation

(rejection-query
 20 200
 (lambda ()
   (pp "This should only print once")
   (let ((result1 (flip))
         (result2 (flip)))
     (observe (= (+ result1 result2) 1))
     (list result1 result2))))
;; expected output: print "This should only print once", return list
;; of 20 (1 0) and (0 1), evenly distributed

(rejection-query
 10 100
 (lambda ()
   (let ((result (flip)))
     (observe #f)
     result)))
;; expected output: empty list

(rejection-query
 20 200
 (lambda ()
   (let ((result1 (bernoulli 2/3))
         (result2 (bernoulli 1/3)))
     (observe (= (+ result1 result2) 1))
     (list result1 result2))))
;; expected output: list of 20 (1 0) and (0 1), with a ~4:1 ratio

(let ((n 1000))
  (/ (fold-left
      + 0
      (rejection-query
       n n
       (lambda () (normal 0 1))))
     n))
;; Mean of n standard normal variables. Expected output: a small
;; number.

;;; Tests for rejection sampling with memoization

(rejection-query
 20 200
 (lambda ()
   (let ((f (mem flip)))
     (list (f) (f)))))
;; expected output: list of 20 (1 1) and (0 0), ~evenly distributed
```

```
(let ((f (mem flip)))
  (rejection-query
   20 200
   (lambda ()
     (list (f) (f)))))
;; expected output: list of 20 (1 1) and (0 0), ~evenly distributed

(let ((f (mem flip)))
  (f)
  (rejection-query
   20 200
   (lambda ()
     (list (f) (f)))))
;; expected output: list of 20 (1 1) or 20 (0 0)

;;; Test for nested rejection sampling and mem

(pp
 (rejection-query
  5 200
  (lambda ()
    (let* ((f (mem bernoulli))
           (f-result (f 49/100)))
      (rejection-query
       5 200
       (lambda ()
         (list
          f-result (f 49/100)
          (f 51/100) (f 51/100)))))))))
;; expected output: list of five lists of five lists of four numbers:
;; (((w x y z))), each 1 or 0
;;
;; in each bottom-level list, w = x and y = z
;;
;; in each middle level list, every w should be the same as every
;; other w, but y should be approximately evenly distributed
;;
;; in the top-level list, w should be approximately evenly distributed
;; (actually 51% 0 for w and 51% 1 for y, but that won't be visible on
;; this level)


;;; Tests for metropolis-hastings

(let ((f (mem flip)))
  (mh-query
   20 10 10
   (lambda ()
     (list (f) (f)))))

(define (simple-bayes-net observation)
  (let* ((cloudy (bernoulli 0.5))
         (sprinkler-prob
          (if (= cloudy 1)
              0.1
              0.5))
         (sprinkler (bernoulli sprinkler-prob))
```

```
         (rain-prob
          (if (= cloudy 1)
              0.8
              0.2))
         (rain (bernoulli rain-prob))
         (wet-grass-prob
          (cond
           ((and (= sprinkler 1) (= rain 1))
            0.99)
           ((and (= sprinkler 1) (= rain 0))
            0.9)
           ((and (= sprinkler 0) (= rain 1))
            0.9)
           (else
            0.01)))
         (wet-grass (bernoulli wet-grass-prob)))
    (if observation
        (for-each
         (lambda (val ob)
           (if ob
               (observe (= val ob))))
         (list cloudy sprinkler rain wet-grass)
         observation))
    (list cloudy sprinkler rain wet-grass)))

(equal-histogram
 (rejection-query
  10000 100000
  (lambda ()
    (simple-bayes-net '(#f #f #f 1)))))

(equal-histogram
 (mh-query
  10000 1000 10
  (lambda ()
    (simple-bayes-net '(#f #f #f 1)))))
;; these two should return similar values



(with-timings
 (lambda ()
   (equal-histogram
    (rejection-query
     10000 100000
     (lambda ()
       (simple-bayes-net '(#f #f #f 1))))))
 (lambda (run-time gc-time real-time)
   (write (internal-time/ticks->seconds run-time))
   (write-char #\space)
   (write (internal-time/ticks->seconds gc-time))
   (write-char #\space)
   (write (internal-time/ticks->seconds real-time))
   (newline)))

(with-timings
 (lambda ()
```

```
   (equal-histogram
    (mh-query
     10000 100 10
     (lambda ()
       (simple-bayes-net '(#f #f #f 1))))))
 (lambda (run-time gc-time real-time)
   (write (internal-time/ticks->seconds run-time))
   (write-char #\space)
   (write (internal-time/ticks->seconds gc-time))
   (write-char #\space)
   (write (internal-time/ticks->seconds real-time))
   (newline)))
;; MCMC is hopefully faster! Though note that the speed of MCMC is
;; determined entirely by the sampler's parameters and the time it
;; takes to draw a sample, while the speed of rejection sampling also
;; depends on the probability of rejecting a sample.

(let ((n 1000))
  (/ (fold-left
      + 0
      (mh-query
       n 100 10
       (lambda () (normal 0 1))))
     n))
;; Mean of standard normal variables again. Expected output: a small
;; number.

(define (named-bayes-net observation)
  (let* ((cloudy ((named-operator bernoulli 'cloudy) 0.5))
         (sprinkler-prob
          (if (= cloudy 1)
              0.1
              0.5))
         (sprinkler ((named-operator bernoulli 'sprinkler)
                     sprinkler-prob))
         (rain-prob
          (if (= cloudy 1)
              0.8
              0.2))
         (rain ((named-operator bernoulli 'rain)
                rain-prob))
         (wet-grass-prob
          (cond
           ((and (= sprinkler 1) (= rain 1))
            0.99)
           ((and (= sprinkler 1) (= rain 0))
            0.9)
           ((and (= sprinkler 0) (= rain 1))
            0.9)
           (else
            0.01)))
         (wet-grass ((named-operator bernoulli 'wet-grass)
                     wet-grass-prob)))
    (if observation
        (for-each
         (lambda (val ob)
           (if ob
```

```
              (observe (= val ob))))
         (list cloudy sprinkler rain wet-grass)
         observation))
    (list cloudy sprinkler rain wet-grass)))

(equal-histogram
 (rejection-query
  10000 100000
  (lambda ()
    (simple-bayes-net '(#f #f 1 1)))))

(equal-histogram
 (rejection-query
  10000 100000
  (lambda ()
    (named-bayes-net '(#f #f 1 1)))))

(define *rejected-samples* 0)
(define *bad-proposals* 0)
(equal-histogram
 (mh-query
  10000 10000 100
  (lambda ()
    (simple-bayes-net '(#f #f 1 1)))))
*rejected-samples*
*bad-proposals*

(define *rejected-samples* 0)
(define *bad-proposals* 0)
(equal-histogram
 (mh-query
  10000 10000 100
  (lambda ()
    (named-bayes-net '(#f #f 1 1)))))
*rejected-samples*
*bad-proposals*
;; Sampling from named-bayes-net should give fewer rejected proposals
;; and bad proposals than sampling from simple-bayes-net. Also,
;; simple-bayes-net should give equal numbers of rejected proposals
;; and bad proposals, but named-bayes-net should reject some feasible
;; proposals.

(define (constrained-bayes-net observation)
  (let ((cloudy-constraint (list-ref observation 0))
        (sprinkler-constraint (list-ref observation 1))
        (rain-constraint (list-ref observation 2))
        (wet-grass-constraint (list-ref observation 3)))
    (let* ((cloudy-operator
            (if cloudy-constraint
                (constrain bernoulli cloudy-constraint)
                (named-operator bernoulli 'cloudy)))
           (cloudy (cloudy-operator 0.5))
           (sprinkler-operator
            (if sprinkler-constraint
                (constrain bernoulli sprinkler-constraint)
                (named-operator bernoulli 'sprinkler)))
           (sprinkler-prob
```

```
             (if (= cloudy 1)
                 0.1
                 0.5))
            (sprinkler (sprinkler-operator
                        sprinkler-prob))
            (rain-operator
             (if rain-constraint
                 (constrain bernoulli rain-constraint)
                 (named-operator bernoulli 'rain)))
            (rain-prob
             (if (= cloudy 1)
                 0.8
                 0.2))
            (rain (rain-operator
                   rain-prob))
            (wet-grass-operator
             (if wet-grass-constraint
                 (constrain bernoulli wet-grass-constraint)
                 (named-operator bernoulli 'wet-grass)))
            (wet-grass-prob
             (cond
              ((and (= sprinkler 1) (= rain 1))
               0.99)
              ((and (= sprinkler 1) (= rain 0))
               0.9)
              ((and (= sprinkler 0) (= rain 1))
               0.9)
              (else
               0.01)))
            (wet-grass (wet-grass-operator
                        wet-grass-prob)))
      (list cloudy sprinkler rain wet-grass))))

(equal-histogram
 (rejection-query
  10000 100000
  (lambda ()
    (constrained-bayes-net '(#f #f 1 1)))))

(define *rejected-samples* 0)
(define *bad-proposals* 0)
(equal-histogram
 (mh-query
  10000 10000 10
  (lambda ()
    (constrained-bayes-net '(#f #f 1 1)))))
*rejected-samples*
*bad-proposals*
;; This should tend to have fewer rejected samples than the named
;; bayes net, and it should have no bad proposals at all.
```