`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.