A Provably Correct Sampler for Probabilistic Programs

Chung-Kil Hur[1], Aditya V. Nori[2], Sriram K. Rajamani[2], and Selva Samuel[3]

1 **Seoul National University, South Korea**
gil.hur@sf.snu.ac.kr
2 **Microsoft Research, US**
{adityan,sriram}@microsoft.com
3 **Carnegie Mellon University, US**
ssamuel@cs.cmu.edu

Abstract

We consider the problem of inferring the implicit distribution specified by a probabilistic program. A popular inference technique for probabilistic programs called Markov Chain Monte Carlo or MCMC sampling involves running the program repeatedly and generating sample values by perturbing values produced in "previous runs". This simulates a Markov chain whose stationary distribution is the distribution specified by the probabilistic program.

However, it is non-trivial to implement MCMC sampling for probabilistic programs since each variable could be updated at multiple program points. In such cases, it is unclear which values from the "previous run" should be used to generate samples for the "current run".

We present an algorithm to solve this problem for the general case and formally prove that the algorithm is correct. Our algorithm handles variables that are updated multiple times along the same path, updated along different paths in a conditional statement, or repeatedly updated inside loops, We have implemented our algorithm in a tool called Infer$^{\checkmark}$. We empirically demonstrate that Infer$^{\checkmark}$ produces the correct result for various benchmarks, whereas existing tools such as R2 and Stan produce incorrect results on several of these benchmarks.

1998 ACM Subject Classification D.2.4 [Software Engineering] Software/Program Verification

Keywords and phrases Probabilistic Programming, Program Correctness, Probabilistic Inference, Markov Chain Monte Carlo Sampling

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2015.475

1 Introduction

Recent years have seen a wide variety of languages for writing probabilistic programs, as well as tools and techniques for performing inference over these programs [7, 16, 24, 21, 8, 14, 22]. One of the main advantages of probabilistic programming is that it allows developers who are familiar with programming language notation, but unfamiliar with machine learning, to focus on the specification of the probabilistic model, and not worry about how to implement inference algorithms over the model. Probabilistic programming tools are able to automatically generate inference code from specifications written as probabilistic programs, thus reducing the degree of expertise required to implement a machine learning algorithm.

We focus on sampling-based inference, in particular, Metropolis-Hastings (MH) based sampling algorithms [3] for probabilistic programming languages. MH based sampling involves execution of the input program with the characteristic that the sample generated at the "current run" depends on the sample generated during the "previous run".

35th IARCS Annual Conf. Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2015).
Editors: Prahladh Harsha and G. Ramalingam; pp. 475–488

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In a probabilistic program, a variable can be assigned values more than once in a single path or assigned values along different paths, and it is unclear what "previous value" to use in generating a "current value" of the variable. For existing probabilistic programming tools such as R2 [22] and Stan [12], the notion of "previous value" used is incorrect (we give examples and more details later), and therefore these tools can produce incorrect results.

In this paper, we precisely define a sample drawn from the distribution represented by the program by specifying a big-step operational semantics for probabilistic programs. We show that this operational semantics is equivalent to the widely accepted denotational semantics of probabilistic programs [17].

Based on this notion of a sample, we then propose a simple MH algorithm for probabilistic programs where a variable can be updated at multiple program points. Our main insight is to track the ordered list of values that a variable gets assigned during a run, together with the distributions that were used to generate these values. We present a procedure to make use of values from such a list generated during a "previous run" to generate samples for the "current run", as well as calculate the quantities (such as acceptance ratio) that are needed for MH sampling. Complications arise because each run of the program can follow different paths with potentially different number of probabilistic assignments to a variable along each path resulting in lists of different lengths across different runs. Our algorithm handles all such cases.

We have implemented our algorithm in a tool called Infer$^{\checkmark}$, and compare it with existing probabilistic programming tools such as R2 and Stan. We prove formally that our algorithm correctly implements MH for probabilistic programs. We also demonstrate cases where existing tools produce incorrect results, whereas our algorithm produces correct results in all cases.

2 Overview

In this section, we first introduce probabilistic programs with an example, and then motivate our algorithm by describing the complications that arise while performing *correct* MH sampling-based inference for probabilistic programs.

Consider the probabilistic program in Figure 1 that is defined over two Boolean variables `x` and `y`. The program tosses two fair coins (modeled by calls to `Bernoulli(0.5)`) in lines 2 and 3, and assigns the outcomes to the variables `x` and `y` respectively. The observe statement `observe(x || y)` in line 4 blocks all executions of the program that do not satisfy the Boolean expression `(x || y)`. The meaning of this program is the distribution over its return expression (which is the tuple `(x, y)` conditioned by permitted executions of the program). This distribution is: $\Pr(\texttt{x} = \texttt{false}, \texttt{y} = \texttt{false}) = 0$, and $\Pr(\texttt{x} = \texttt{false}, \texttt{y} = \texttt{true}) = \Pr(\texttt{x} = \texttt{true}, \texttt{y} = \texttt{false}) = \Pr(\texttt{x} = \texttt{true}, \texttt{y} = \texttt{true}) = 1/3$.

Inference. Probabilistic inference is the task of determining the distribution implicitly specified by a probabilistic program. Inference can be performed by executing the program several times and averaging over the resulting samples. To do this efficiently, many probabilistic programming tools [8, 12, 22] employ Markov Chain Monte Carlo (MCMC) sampling algorithms [19] and their variants.

```
1: bool x, y;
2: x ~ Bernoulli(0.5);
3: y ~ Bernoulli(0.5);
4: observe(x || y);
5: return(x, y);
```

Figure 1 A simple probabilistic program.

To make this paper self-contained, we give a brief overview of the most basic form of an MCMC algorithm which is the Metropolis-Hastings

(MH) algorithm [19]. The MH algorithm takes a *target probability distribution* $T(\bar{x})$ as input, and returns samples that are distributed according to this distribution by performing the following steps:

1. First, a *proposal distribution* $Q(v_{old} \to v_{new})$ is used to pick a new value v_{new} for the variable $\bar{x}$ by appropriately perturbing its old value v_{old}.
2. Next, a parameter β called the *acceptance ratio* is computed. It is used to decide whether to accept or reject the new sampled value v_{new} for $\bar{x}$, and is defined as follows:

$$\beta = \min\left\{1, \frac{T(v_{new}) \times Q(v_{new} \to v_{old})}{T(v_{old}) \times Q(v_{old} \to v_{new})}\right\}$$

3. The sample is accepted if a random draw from a Bernoulli distribution with mean β (1 occurs with probability β and 0 occurs with probability $1 - \beta$) results in a 1, otherwise it is rejected.

The MH algorithm executes the above steps iteratively to generate samples.

A probabilistic program P denotes a target distribution T implicitly (see Section 3 for a formal definition of the target distribution denoted by a probabilistic program).

In order to perform MH sampling on a probabilistic program, we need to run the program P which results in the state being constructed incrementally, as P executes along a path. Along the path, the program encounters several probabilistic assignments to variables, and to generate a new value for each variable, we require the corresponding old value of the variable from the previous run of the program. Such an association between old and new values is easy if each variable is assigned only once, or if the program is a single path without branches or loops.

However, associating old values with new values is non-trivial for the general case. If the program has branches or loops, the previous value corresponding to a probabilistic assignment may come from an assignment in a different branch than the one currently being executed. Also, different branches may generate samples for the same variable from different distributions. Alternatively, a variable may have been assigned multiple times during execution of the previous run, and it is sometimes unclear which of these values is to be used to generate values for the current run. So, care must be taken to compute the MH acceptance ratio correctly. Due to these reasons, implementing the above 3 steps of the MH algorithm for a probabilistic program is non-trivial.

We illustrate the difficulties with implementing a correct sampling algorithm via the following examples.

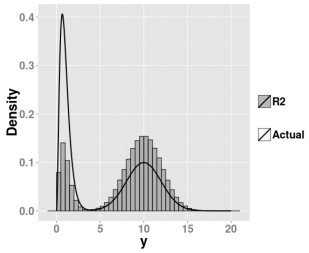
Example 1 (mixtures). Consider the probabilistic program shown in Figure 2(a). The program is defined over two variables `x` and `y`. The variable `y` is drawn from a mixture of a Gaussian distribution and a Gamma distribution (specified by the if-then-else statement in lines 3–6), whereas the variable `x` is drawn from another Gaussian distribution (line 2), and determines the mixture proportion.

Suppose we perform MH sampling for this example. Assume that during run n, the program follows the path 1, 2, 3, 4, 7. That is, `x` was assigned a value greater than 0, say 0.1, and the "then" branch of the conditional statement was taken, and `y` was assigned a value say 9.6 from the distribution `Gaussian(10, 2)`.

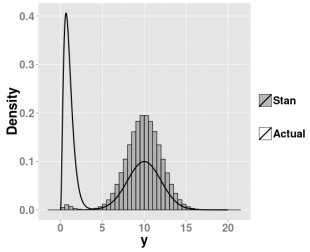
Next, we consider how to execute run $n + 1$ using MH sampling. During run $n + 1$, at line 2, in order to generate a current value for `x`, we need to propose a value for `x` using a proposal distribution Q_x centered around the old value of `x`, namely 0.1, and calculate the

```
1: double x, y;
2: x ~ Gaussian(0, 1);
3: if (x > 0) then
4:   y ~ Gaussian(10, 2);
5: else
6:   y ~ Gamma(3, 3);
7: return y;
```

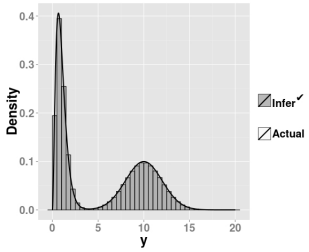
(a)



(b)



(c)



(d)

Figure 2 A probabilistic program for a mixture model together with inference results.

parameter β_x as the ratio described earlier. Now suppose the current value of `x` so chosen is -0.05 which is less than 0, then the "else" branch is taken, and we need to produce a current value for `y`.

How should we now generate the current value for `y`? Prior work such as [29] use the value of y from the most recent run which took the "else" branch, say run $n-3$ (assuming runs $n-2$ and $n-1$ took the "then" branch) and use that value to generate the current value of `y`. The probabilistic programming tools R2 [22] and Stan [12] follow the same algorithm.

However, if a value from a run other than the previously accepted run is used, then this would result in the algorithm converging to the incorrect distribution. This is shown in the plots in Figure 2. Plot 2(b) depicts the distribution inferred by R2 for the program in Figure 2(a), and plot 2(c) shows the distribution computed by Stan for the same program. The density function of the correct distribution for this example is shown by the line graph labelled `Actual` in each of the plots in Figure 2.

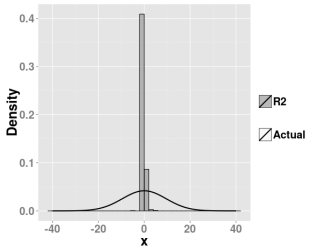
Example 2 (loop). Consider the probabilistic program for a hierarchical model shown in Figure 3(a). The loop in lines 4–7 constructs a series of Gaussian distributions with the final answer also being a Gaussian distribution (shown by the line graphs labelled `Actual` in the plots of Figure 3). The variable `x` is assigned 11 times during an execution of the program.

Tools like R2 and Stan use the last value that `x` was assigned in the previous run to generate each of the 11 values in the current run. So, these tools produce incorrect results as shown by the density histogram plots 3(b) and 3(c), respectively. To perform MH sampling correctly, it is necessary to record each of the 11 values generated for `x` and use the corresponding previous value to propose a new value in the current run.

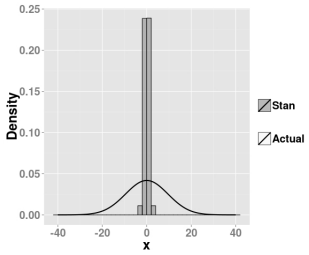
In addition to keeping track of previous values correctly, it is also important to record the distributions in the corresponding sampling statements whose execution generated those values. This is necessary so that the density of the previous value may be computed w.r.t. the correct target distribution while computing the acceptance ratio.

```
1: double x;
2: int i = 0;
3: x ~ Gaussian(0, 1);
4: while (i < 10) do {
5:   x ~ Gaussian(x, 3);
6:   i = i+1;
7: }
8: return x;
```

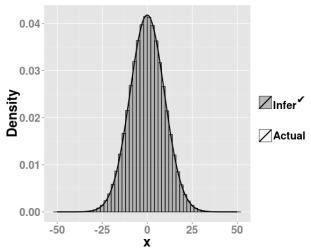
(a)



(b)



(c)



(d)

Figure 3 A probabilistic program for a hierarchical model together with inference results.

Algorithm. Motivated by the above examples, we desire to come up with an algorithm which correctly chooses the appropriate value from the previous run to be used to propose values for the current run, and computes the correct acceptance ratio in all possible scenarios.

Our sampling algorithm, called INFER$^{\checkmark}$, is based on a sampling-based operational semantics for probabilistic programs that specifies the meaning of a sample by clearly setting out all the values generated in an execution that would need to be tracked. The operational semantics also specifies how to compute the probability density for the sample generated in an execution so that the MH acceptance ratio may be computed correctly. In addition, INFER$^{\checkmark}$ uses samples from only the previously accepted run to propose new values in the current run. We also prove that by doing this, INFER$^{\checkmark}$ computes the correct distribution specified by any input probabilistic program. We informally describe the main ideas behind the algorithm next. A more complete and formal description is given in Sections 3 and 4.

INFER$^{\checkmark}$ uses a list for each variable in the program to keep track of the probabilistic assignments to that variable during a single run of the program (as opposed to the standard variable to value mapping). Each element of the list is a pair whose first element is the value generated for the corresponding variable when a sampling statement was encountered during the execution of the program. The second element of the pair is used to store information about the distribution used in the sampling statement and its parameters. This information about distributions is stored in order to compute the MH acceptance ratio correctly.

Specifically, INFER$^{\checkmark}$ maintains two lists for each sampled variable x in the program:

1. The first list l_{PRE} is used to store the samples generated for x together with the distribution information at the corresponding sampling statement executed in the most recent run which produced a sample that was accepted. The values in this list are used to propose new values for x. The distribution information is used to update the MH acceptance ratio.
2. The second list l_{CUR} is used to store the samples produced for x paired with the distribution information at the corresponding sampling statement executed in the current run of the

program. When a sample is accepted, l_{PRE} is assigned the value of l_{CUR}. Otherwise, l_{CUR} is discarded, and constructed again as the program is executed and new values are proposed for x.

We now informally show how INFER$^{\checkmark}$ correctly works for the previous examples.

Example 1 (mixtures). For the program in Figure 2(a), the variables `x` and `y` are sampled only once in every run of the program. Therefore, the l_{PRE} and l_{CUR} lists for both `x` and `y` will each contain one element at the end of every run. The sample value in the l_{PRE} and l_{CUR} lists for `x` is produced by the execution of the probabilistic assignment statement in line 2. The distribution information for `x` would note that the distribution at line 2 is a standard Gaussian distribution.

On the other hand, the sample value and distribution information in the l_{PRE} and l_{CUR} lists for `y` can be generated either by executing line 4 (if the "then" branch is taken) or line 6 (if the "else" branch is taken). This ensures that correct previous values are used to propose new values of `y` even when different branches are taken in different runs of the program.

Storing the distribution information also enables the computation of the density of the previous sample w.r.t. to the correct target distribution which is needed for calculating the MH acceptance ratio. By doing this, INFER$^{\checkmark}$ is able to produce the correct answer as shown by the plot in Figure 2(d).

Example 2 (loop). The program in Figure 3(a) contains one random variable `x` which is sampled 11 times during an execution (one at line 3 and ten at line 5). Thus, the l_{PRE} and l_{CUR} lists for `x` will each contain 11 elements at the end of every run.

The first value in l_{PRE} is generated by the probabilistic assignment statement in line 3 in the previous run. This value is used to propose a new value for `x` when line 3 is executed in the current run. This new proposed value is added as the first element of the first pair in the list l_{CUR} for `x`. The corresponding distribution information in both the lists specifies that the target is the standard Gaussian distribution.

Similarly, the other 10 values in l_{PRE} come from the execution of the probabilistic assignment statement in line 5 during the iterations of the while loop in the previous run, and are used to propose a new value for `x` when line 5 is executed in the corresponding iteration in the current run. Each proposed value for `x` is added to the list l_{CUR} as the first element of the corresponding pair. The distribution information for each of these pairs in both lists specifies that the target distribution is a Gaussian distribution whose mean is the value of x before line 5 is executed in that iteration and whose standard deviation is 3.

As noted earlier, the distribution information is used to compute the density of the previous samples w.r.t. to the correct target distribution to enable the correct computation of the MH acceptance ratio. INFER$^{\checkmark}$ computes the correct distribution for this example also as can be seen in the plot 3(d).

Notice that we only maintain the sequence of values generated for a variable during an execution. It is not necessary to keep track of the program points which produced these values. Also, note that different runs of the program may produce lists of different lengths for a particular variable, since different runs can follow different paths in the program.

If the list l_{PRE} for a given variable `x` has *fewer* elements than those needed to produce values for the current run, then, for the extra samples that are produced in the current run, the algorithm resorts to *Metropolis independent sampling* [10, 27, 18]. It is a modification to the MH algorithm in which the proposal distribution is independent of the previous sample value. The MH acceptance ratio is also updated appropriately.

$x \in \text{Vars}$
$\mathbf{uop} ::= \cdots$ C unary operators
$\mathbf{bop} ::= \cdots$ C binary operators
$\varphi, \psi ::= \cdots$ logical formula

$\mathcal{E} ::=$ expressions
$\mid x$ variable
$\mid c$ constant
$\mid \mathcal{E}_1 \ \mathbf{bop} \ \mathcal{E}_2$ binary operation
$\mid \mathbf{uop} \ \mathcal{E}$ unary operation

$\mathcal{S} ::=$ statements
$\mid \texttt{skip}$ skip
$\mid x = \mathcal{E}$ deterministic assignment
$\mid x \sim \texttt{Dist}(\bar{\theta})$ probabilistic assignment
$\mid \texttt{observe}(\varphi)$ observe
$\mid \mathcal{S}_1; \mathcal{S}_2$ sequential composition
$\mid \texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2$ conditional composition
$\mid \texttt{while } \mathcal{E} \texttt{ do } \mathcal{S}$ while−do loop

$\mathcal{P} ::= \mathcal{S} \texttt{ return } (\mathcal{E}_1, \ldots, \mathcal{E}_n)$ program

Figure 4 Syntax of PROB.

- Unnormalized Semantics for Statements
  $[\![\mathcal{S}]\!] \in (\Sigma \to [0,1]) \to \Sigma \to [0,1]$

$$
\begin{aligned}
[\![\texttt{skip}]\!](f)(\sigma) &:= f(\sigma) \\
[\![x = \mathcal{E}]\!](f)(\sigma) &:= f(\sigma[x \leftarrow \sigma(\mathcal{E})]) \\
[\![x \sim \texttt{Dist}(\bar{\theta})]\!](f)(\sigma) &:= \int_{v \in \text{Val}} \texttt{Dist}(\sigma(\bar{\theta}))(v) \times f(\sigma[x \leftarrow v])\, dv \\
[\![\texttt{observe}(\varphi)]\!](f)(\sigma) &:= \begin{cases} f(\sigma) & \text{if } \sigma(\varphi) = \texttt{true} \\ 0 & \text{otherwise} \end{cases} \\
[\![\mathcal{S}_1; \mathcal{S}_2]\!](f)(\sigma) &:= [\![\mathcal{S}_1]\!]([\![\mathcal{S}_2]\!](f))(\sigma) \\
[\![\texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2]\!](f)(\sigma) &:= \begin{cases} [\![\mathcal{S}_1]\!](f)(\sigma) & \text{if } \sigma(\mathcal{E}) = \texttt{true} \\ [\![\mathcal{S}_2]\!](f)(\sigma) & \text{otherwise} \end{cases} \\
[\![\texttt{while } \mathcal{E} \texttt{ do } \mathcal{S}]\!](f)(\sigma) &:= \sup_{n \geq 0} [\![\texttt{while } \mathcal{E} \texttt{ do}_n \mathcal{S}]\!](f)(\sigma)
\end{aligned}
$$

  where
  $\texttt{while } \mathcal{E} \texttt{ do}_0 \mathcal{S} = \texttt{observe}(\texttt{false})$
  $\texttt{while } \mathcal{E} \texttt{ do}_{n+1} \mathcal{S} = \texttt{if } \mathcal{E} \texttt{ then } (\mathcal{S}; \texttt{while } \mathcal{E} \texttt{ do}_n \mathcal{S}) \texttt{ else } (\texttt{skip})$

- Normalized Semantics for Programs
  $[\![\mathcal{S} \texttt{ return } (\mathcal{E}_1, \ldots, \mathcal{E}_n)]\!] \in (\mathbb{R}^n \to [0,1]) \to [0,1]$

$$
[\![\mathcal{S} \texttt{ return } (\mathcal{E}_1, \ldots, \mathcal{E}_n)]\!](f) := \frac{[\![\mathcal{S}]\!](\lambda\sigma.\ f(\sigma(\mathcal{E}_1), \ldots, \sigma(\mathcal{E}_n)))(\bot)}{[\![\mathcal{S}]\!](\lambda\sigma.\ 1)(\bot)}
$$

  where $\bot$ denotes the default initial state.

Figure 5 Denotational Semantics of PROB.

On the other hand, if the list l_{PRE} for a given variable `x` has *more* elements than those needed to produce values for the current run, then, the extra values in l_{PRE} are used to produce some adjustments in the MH acceptance ratio β. These details are explained in Section 4.

3 Probabilistic Programs

We consider a probabilistic programming language called PROB [13] whose syntax is formally described in Figure 4. A PROB program consists of statements and a return expression. Variables have base types such as bool, int, float and double, and expressions include variables, constants, binary and unary operations. Statements include primitive statements (skip, deterministic assignment, probabilistic assignment, observe) and composite statements (sequential composition, conditionals and loops). Features such as arrays, pointers, structures and function calls can be incorporated in the language, but for the sake of brevity, we omit these features from the core language.

A popular choice of the specification of formal semantics of probabilistic programs is the denotational semantics introduced by Kozen in [17]. This is summarized in Figure 5. The denotational semantics specifies the meaning of a probabilistic program by defining the joint distribution over the output state of the program, where a state σ of a program is a partial

$$\frac{}{(\texttt{skip}, \sigma) \Downarrow^{\epsilon} (1, \sigma)} \qquad \frac{}{(x = \mathcal{E}, \sigma) \Downarrow^{\epsilon} (1, \sigma[x \leftarrow \sigma(\mathcal{E})])} \qquad \frac{\sigma(\varphi) = \texttt{true}}{(\texttt{observe}(\varphi), \sigma) \Downarrow^{\epsilon} (1, \sigma)}$$

$$\frac{v \in \text{Val} \qquad p = \texttt{Dist}(\sigma(\bar{\theta}))(v) > 0}{(x \sim \texttt{Dist}(\bar{\theta}), \sigma) \Downarrow^{x \mapsto [v]} (p, \sigma[x \leftarrow v])} \qquad \frac{(\mathcal{S}_1, \sigma) \Downarrow^{s_1} (p_1, \sigma_1) \qquad (\mathcal{S}_2, \sigma_1) \Downarrow^{s_2} (p_2, \sigma_2)}{(\mathcal{S}_1; \mathcal{S}_2, \sigma) \Downarrow^{s_1 \cdot s_2} (p_1 \times p_2, \sigma_2)}$$

$$\frac{\sigma(\mathcal{E}) = \texttt{true} \qquad (\mathcal{S}_1, \sigma) \Downarrow^{s_1} (p_1, \sigma_1)}{(\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2, \sigma) \Downarrow^{s_1} (p_1, \sigma_1)} \qquad \frac{\sigma(\mathcal{E}) = \texttt{false} \qquad (\mathcal{S}_2, \sigma) \Downarrow^{s_2} (p_2, \sigma_2)}{(\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2, \sigma) \Downarrow^{s_2} (p_2, \sigma_2)}$$

$$\frac{\sigma(\mathcal{E}) = \texttt{false}}{(\texttt{while}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}, \sigma) \Downarrow^{\epsilon} (1, \sigma)} \qquad \frac{\sigma(\mathcal{E}) = \texttt{true} \qquad (\mathcal{S}; \texttt{while}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}, \sigma) \Downarrow^{s} (p, \sigma)}{(\texttt{while}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}, \sigma) \Downarrow^{s} (p, \sigma)}$$

where

$$\begin{aligned} \epsilon &= \lambda u.\, [\,] \\ x \mapsto [v] &= \lambda u.\, \text{if } u = x \text{ then } [v] \text{ else } [\,] \\ s_1 \cdot s_2 &= \lambda u.\, s_1(u) +\!\!+ s_2(u), \text{ and } +\!\!+ \text{ denotes list concatenation} \end{aligned}$$

Figure 6 Sampling-based operational semantics of Prob.

valuation of all its variables. The set of all states (possibly infinite) is denoted by Σ.

However, in order to design an MCMC algorithm, we need to have a distribution over program executions. To this end, we introduce a big-step operational semantics for Prob. The operational semantics defines the probability density of a sample. It is interesting to note here that each sample (a sequence of program states) uniquely determines a program execution. We will also assume that all program executions terminate with probability 1.

The operational semantics thus gives rise to a distribution over these program executions. From this distribution, the distribution over the output state can be derived by marginalizing out the intermediate values. We also show that this distribution is the same as that defined by the denotational semantics.

Sampling-based Operational Semantics. We now inductively define the sampling-based operational semantics of Prob in Figure 6, which will form the basis of our MH-based sampling algorithm presented in Section 4.

The relation $(\mathcal{S}, \sigma) \Downarrow^{s} (p, \sigma')$ intuitively means that if we run the program statement $\mathcal{S}$ with initial state σ, it may internally draw a sample $s \in \mathbb{S}$ from the associated distributions and terminate with output state σ', with probability density p. Here the sample space $\mathbb{S}$ is defined as $\Gamma \to \text{List}(\text{Val})$ with Γ the set of variables and $\text{Val} = \mathbb{Z} \uplus \mathbb{R}$. Notice that, as explained earlier, a sample consists of a *list* of values associated with each random variable.

All rules of the sampling-based operational semantics are standard except for probabilistic assignment, observe, and sequential composition statements. The probabilistic assignment statement draws a sample v from the given distribution with the associated density p and sets the variable x to v. The observe statement proceeds only when the given condition φ is met. The sequential composition statement executes the sub-statements in order, and then multiplies the associated densities and concatenates the generated samples.

4 Algorithm

Given a program $\mathcal{P}$ written in Prob, and κ (the number of samples to be generated) as input, Infer$^{\checkmark}$ (shown in Algorithm 1) returns a sequence of samples from the distribution specified by $\mathcal{P}$. Note that the parameter κ controls the accuracy of the algorithm—the greater the number of samples generated by the algorithm, the better is the approximation to the actual distribution specified by $\mathcal{P}$. Infer$^{\checkmark}$ uses standard ideas from MH sampling, which were reviewed in Section 2. The program $\mathcal{P}$ is executed for κ times, and every execution produces

Algorithm 1 $\textsc{Infer}^{\checkmark}(\mathcal{P}, \kappa)$

Input: A Prob program $\mathcal{P}$, and κ, the number of samples to be generated.
Output: Samples from the distribution specified by $\mathcal{P}$.

```
1: Ω := [], Θ_ACC := ϵ
2: for i = 1 to κ do
3:    β := 1.0, Θ_PRE := Θ_ACC, Θ := ϵ
4:    σ := ⊥, ret := ()
5:    S return(E_1, ..., E_n) := P
6:    (σ, β, Θ_PRE, Θ) := EVAL(S, σ, β, Θ_PRE, Θ)
7:    ret := (σ(E_1), ..., σ(E_n))
8:    for x in Γ do                          (* Γ is the set of variables in P *)
9:       while Θ_PRE(x) ≠ [] do
10:         (v_PRE, Δ_PRE) := HEAD(Θ_PRE(x))
11:         Θ_PRE(x) := TAIL(Θ_PRE(x))
12:         β := β × PROP(Δ_PRE)(v_PRE) / Δ_PRE(v_PRE)
13:      end while
14:   end for
15:   if ret ≠ () ∧ (Ω = [] ∨ β ≥ 1 ∨ BERNOULLI(β)) then
16:      Ω := ret :: Ω
17:      Θ_ACC := Θ
18:   else
19:      if Ω ≠ [] then Ω := HEAD(Ω) :: Ω endif
20:   end if
21: end for
22: return Ω
```

a sample (as defined by the operational semantics), and a value β which determines whether the sample will be accepted or rejected.

In line 1 of Algorithm 1, $\textsc{Infer}^{\checkmark}$ initializes variables Ω and Θ_{ACC}. The list Ω is initialized to an empty list, and is used to store the values of the return expression at the end of each accepted run of the program. The map Θ_{ACC} is used to store the most recently accepted sample and is initialized to the empty map. Θ_{ACC} maps each sampled variable to a list of pairs generated for that variable in the last accepted execution of the program. The first element of each pair in the list is the sample value v that is assigned to the variable. The second element is the probability distribution from which the value v of the variable is drawn and is used to compute the MH acceptance ratio β. Thus, the samples generated by Algorithm 1 conform to the definition of the sample specified by the operational semantics.

Lines 2–21 generate κ samples—each sample is either accepted or rejected based on the value of the parameter β (lines 15–20 which encode the standard MH acceptance criterion). Note that the first sample (represented by the test for empty Ω in line 15) is always accepted. If a sample is rejected, then the program sample generated from the previous execution of $\mathcal{P}$ is added to Ω (line 19). The maps Θ_{PRE} and Θ (initialized in line 3) have the same type as Θ_{ACC}. The map Θ_{PRE} is initialized to the previously accepted sample and is used to determine the proposal distribution to use at each sampling statement and to compute the acceptance ratio β correctly. The map Θ is used to build up the sample for the current execution of the program.

Note that $\textsc{Infer}^{\checkmark}$ has the same high level structure as the standard MH procedure. The recursive procedure Eval (called in line 6) operates over the syntactic structure of the program $\mathcal{P}$. It defines the *transitions* of the Markov chain constructed by Algorithm 1.

The procedure Eval is described by the rules in Figure 7. Given a statement $\mathcal{S}$, a program state σ (which is a partial map from variables to values), a parameter β (which is used to decide whether the sample generated is to be accepted or rejected), the map Θ_{PRE} and the map Θ, the procedure $\textsc{Eval}(\mathcal{S}, \sigma, \beta, \Theta_{\text{PRE}}, \Theta)$ computes new values for σ, β, Θ_{PRE} and Θ obtained after executing statement $\mathcal{S}$.

$$
\begin{array}{lcl}
\textsc{Eval}(x = \mathcal{E}, \sigma, \beta, \Theta_{\text{PRE}}, \Theta) & = & (\sigma[x \leftarrow \sigma(\mathcal{E})], \beta, \Theta_{\text{PRE}}, \Theta) \\
\textsc{Eval}(\texttt{skip}, \sigma, \beta, \Theta_{\text{PRE}}, \Theta) & = & (\sigma, \beta, \Theta_{\text{PRE}}, \Theta) \\
\textsc{Eval}(\mathcal{S}_1; \mathcal{S}_2, \sigma, \beta, \Theta_{\text{PRE}}, \Theta) & = & \textbf{let } (\sigma', \beta', \Theta'_{\text{PRE}}, \Theta') = \textsc{Eval}(\mathcal{S}_1, \sigma, \beta, \Theta_{\text{PRE}}, \Theta) \textbf{ and} \\
 & & \textbf{let } (\sigma'', \beta'', \Theta''_{\text{PRE}}, \Theta'') = \textsc{Eval}(\mathcal{S}_2, \sigma', \beta', \Theta'_{\text{PRE}}, \Theta') \textbf{ in} \\
 & & (\sigma'', \beta'', \Theta''_{\text{PRE}}, \Theta'') \\
\textsc{Eval}(\texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2, & = & \textbf{if } \sigma(\mathcal{E}) \textbf{ then } \textsc{Eval}(\mathcal{S}_1, \sigma, \beta, \Theta_{\text{PRE}}, \Theta) \\
\quad \sigma, \beta, \Theta_{\text{PRE}}, \Theta) & & \textbf{else } \textsc{Eval}(\mathcal{S}_2, \sigma, \beta, \Theta_{\text{PRE}}, \Theta) \\
\textsc{Eval}(x \sim \texttt{Dist}(\bar{\theta}), \sigma, \beta, \Theta_{\text{PRE}}, \Theta) & = & \textbf{let } \Delta = \texttt{Dist}(\sigma(\bar{\theta})) \textbf{ and} \\
 & & \textbf{let } (v, \beta', \Theta'_{\text{PRE}}) = \\
 & & \textbf{if } \Theta_{\text{PRE}}(x) = [\,] \textbf{ then} \\
 & & \quad \textbf{let } v \sim \textsc{Prop}(\Delta) \textbf{ and} \\
 & & \quad \textbf{let } \beta' = \beta \times \frac{\Delta(v)}{\textsc{Prop}(\Delta)(v)} \textbf{ in } (v, \beta', \Theta_{\text{PRE}}) \\
 & & \textbf{else} \\
 & & \quad \textbf{let } (v_{\text{PRE}}, \Delta_{\text{PRE}}) = \textsc{Head}(\Theta_{\text{PRE}}(x)) \textbf{ and} \\
 & & \quad \textbf{let } v \sim \textsc{Prop}(\Delta, v_{\text{PRE}}) \textbf{ and} \\
 & & \quad \textbf{let } \beta' = \beta \times \frac{\Delta(v) \times \textsc{Prop}(\Delta_{\text{PRE}}, v)(v_{\text{PRE}})}{\Delta_{\text{PRE}}(v_{\text{PRE}}) \times \textsc{Prop}(\Delta, v_{\text{PRE}})(v)} \textbf{ in} \\
 & & \quad (v, \beta', \textsc{Tail}(\Theta_{\text{PRE}})) \\
 & & \textbf{in} \\
 & & (\sigma[x \leftarrow v], \beta', \Theta'_{\text{PRE}}, \Theta(x) +\!\!+ [(v, \Delta)]) \\
\textsc{Eval}(\texttt{observe } (\varphi), \sigma, \beta, \Theta_{\text{PRE}}, \Theta) & = & \textbf{if } \sigma(\varphi) \textbf{ then } (\sigma, \beta, \Theta_{\text{PRE}}, \Theta) \textbf{ else } (\sigma, 0, \Theta_{\text{PRE}}, \Theta) \\
\textsc{Eval}(\texttt{while } \mathcal{E} \texttt{ do } \mathcal{S}, \sigma, \beta, \Theta_{\text{PRE}}, \Theta) & = & \textbf{let } (\sigma', \beta', \Theta'_{\text{PRE}}, \Theta') = \textsc{Eval}(\mathcal{S}, \sigma, \beta, \Theta_{\text{PRE}}, \Theta) \textbf{ in} \\
 & & \textbf{if } \sigma(\neg\mathcal{E}) \textbf{ then} \\
 & & \quad (\sigma, \beta, \Theta_{\text{PRE}}, \Theta) \\
 & & \textbf{else} \\
 & & \quad \textsc{Eval}(\textbf{while } \mathcal{E} \texttt{ do } \mathcal{S}, \sigma', \beta', \Theta'_{\text{PRE}}, \Theta')
\end{array}
$$

Figure 7 Given a statement $\mathcal{S}$ and parameters σ, β, Θ_{PRE}, and Θ, EVAL computes a tuple of parameters evaluated over $\mathcal{S}$ which are used by Algorithm 1 .

For instance, consider the assignment statement $x = \mathcal{E}$. Upon executing this from a state σ, we obtain the state $\sigma[x \leftarrow \sigma(\mathcal{E})]$, i.e., the value corresponding to the variable x in σ is set to the evaluation of $\mathcal{E}$ over σ. The values of β, Θ_{PRE} and Θ remain unchanged. The rules for the other statements also proceed in a similar manner.

The rule for the probabilistic assignment statement "$x \sim \texttt{Dist}(\bar{\theta})$" in EVAL specifies how to generate a sample and update β appropriately. As seen in Figure 7, in this case, first the variable Δ is set to the probability distribution function $\texttt{Dist}(\bar{\theta})$ with respect to the current state σ (*i.e.,* the parameters of the distribution $\bar{\theta}$ are evaluated at the state σ). Next, a value v is sampled, and an update to β is performed based on the following conditions:

Previous value for x is not available. This condition is represented by the predicate $\Theta_{\text{PRE}}(x) = [\,]$, which says that there are no values associated with the variable x from the previous execution of $\mathcal{P}$. A new value v for x is drawn from a proposal distribution $\textsc{Prop}(\Delta)$ that only depends on the target distribution Δ. In essence, this is the *Metropolized independent sampling algorithm* [18] which is a modification to the MH algorithm in which the proposal distribution is independent of the previous sample value. There are a number of choices for the proposal distribution, and in our implementation we set $\textsc{Prop}(\Delta)$ to be the target distribution Δ.

The updated value β' of β is the usual update for an MH algorithm. Intuitively, it is helpful to think of the previous sample value as a special value that is drawn from a distribution which produces this value with probability 1. If the proposal for this distribution is taken to be the distribution itself, then the MH acceptance ratio for x reduces to $\frac{\Delta(v)}{\textsc{Prop}(\Delta)(v)}$.

Previous value for x is available. This means that the list associated with the variable x in the map Θ_{PRE} is not empty. In particular, the previous value v_{PRE} for x and the probability density function Δ_{PRE} from which v_{PRE} is drawn are at the head of the list for x in the map Θ_{PRE}. Therefore, a new value for x is drawn from a proposal distribution $\textsc{Prop}(\Delta, v_{\text{PRE}})$ that depends on the previous value v_{PRE} for x.

The updated value β' of β is the usual update for an MH algorithm. As noted earlier, $\Delta_{\text{PRE}}(v_{\text{PRE}})$ is used for updating β and so, it is important to also keep track of the distribution from which a sample is drawn.

Finally, the state σ is updated to $\sigma[x \leftarrow v]$, the entry for x in the map Θ is appended with the value (v, Δ), and the tuple $(\sigma[x \leftarrow v], \beta', \textsc{Tail}(\Theta_{\text{PRE}}), \Theta(x)\text{++}[(v, \Delta)])$ is returned (where ++ denotes list concatenation).

It is also important to note that lines 8–14 in Algorithm 1 update β to take care of cases where the current value for a variable is undefined and the previous value is defined. This can be intuitively understood in a similar way as the update of β when a previous value is not available.

The following theorem states that for a probabilistic program $\mathcal{P}$, $\textsc{Infer}^{\checkmark}(\mathcal{P}, \kappa)$, computes answers that are consistent with $[\![\mathcal{P}]\!]$.

▶ **Theorem 4.1.** *For a program $\mathcal{P}$, the expectation of its return expression computed with respect to the samples generated by $\textsc{Infer}^{\checkmark}(\mathcal{P}, \kappa)$ approaches its denotational semantics $[\![\mathcal{P}]\!]$, as $\kappa \to \infty$.*

5 Evaluation

First, we show that $\textsc{Infer}^{\checkmark}$ computes correct answers for a set of micro-benchmark probabilistic programs. For these programs, other sampling based tools such as R2 and Stan compute answers that deviate significantly from the actual or analytically computed answers. Next, to demonstrate the practical applicability of $\textsc{Infer}^{\checkmark}$, we also show that it is able to work effectively for three real-world benchmark programs. All experiments were performed on a PC with a 2.00 GHz Intel 3rd Generation Core i7 processor and 8 GB RAM and running Microsoft Windows 8.1.

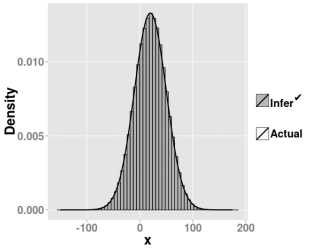
The first two micro-benchmarks that we consider are the programs in Figures 2(a) and 3(a). As can be seen in the plots 2(d) and 3(d), $\textsc{Infer}^{\checkmark}$ is able to estimate the correct answers. As discussed in Section 2, both R2 and Stan produce incorrect answers for these programs.

The other micro-benchmarks that we consider are shown in Figure 8. The benchmark (a) includes two assignments to the same variable `x`. The benchmark (b) is interesting because the variable `x` can be assigned different number of times across different runs of the program. If line 2 produces a value greater than 0.5, `x` is sampled again at line 4. On the other hand, if a sample value less than or equal to 0.5 is generated at line 2, then line 4 is not executed and `x` gets assigned only once. The benchmark (c) is a hierarchical model in which the prior distribution is estimated by means of a mixture of Gaussian distributions.

As seen from these figures, $\textsc{Infer}^{\checkmark}$ estimates distributions that coincide with the actual or analytically computed distributions. On the other hand, R2 and Stan produce incorrect results for all of the micro-benchmarks due to their lack of properly handling multiple sampling for the same variable and sampling from different execution paths.

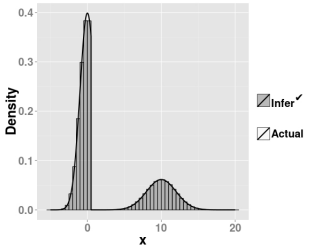
To demonstrate that $\textsc{Infer}^{\checkmark}$ is a practical algorithm, we also evaluate it on the following real-world benchmarks that are frequently used to test the robustness and scalability of Bayesian inference algorithms. As seen from the times reported below, $\textsc{Infer}^{\checkmark}$ is quite efficient on these benchmarks and therefore a practical solution.

```
1: double x;
2: x ~ Gaussian(10, 20);
3: x ~ Gaussian(20, 30);
4: return x;
```



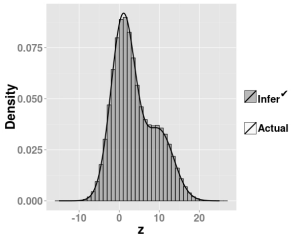
(a) Multiple assignments.

```
1: double x;
2: x ~ Gaussian(0, 1);
3: if (x > 0.5) then
4:   x ~ Gaussian(10, 2);
5: return x;
```



(b) Mixture model 1.

```
1: double x, y, z;
2: x ~ Gaussian(0, 1);
3: if (x > 0.5) then
4:   y ~ Gaussian(10, 2);
5: else
6:   y ~ Gamma(3, 3);
7: z ~ Gaussian(y, 3);
8: return z;
```



(c) Mixture model 2.

Figure 8 Micro-benchmarks.

- `Linear regression`: This is the standard Bayesian formulation of the linear regression model for fitting 1000 points [28] (*time taken by* Infer✓*: 12.55 seconds*).
- `HIV`: This is a multi-level or hierarchical linear model with varying slope and intercept. This model is for inferring the immunity levels in HIV-positive patients. The data comprises of 369 measurements taken over a two-year period on 84 patients [6] (*time taken by* Infer✓*: 4.36 seconds*).
- `Halo`: This is a skill rating system for a tournament of the Halo video game among 35 teams, with 2 players per team, and 500 games played between the teams [11] (*time taken by* Infer✓*: 5809 seconds*).

6 Related work

There has been significant progress in the development of probabilistic programming languages and tools in recent years [7, 21, 24, 8, 12, 22, 1, 9]. There are several approaches to performing inference for programs written in these languages: (1) by using static analysis techniques [20, 26, 4] such as abstract interpretation and data flow analysis, (2) by using dynamic analysis techniques [8, 12, 22, 2] such as MCMC sampling algorithms [19], or (3) by using Bayesian techniques where the program is compiled to a probabilistic model such as a Bayesian network [15] and inference is performed using probabilistic inference techniques such as belief propagation [23] over the probabilistic model [21].

We have observed that for techniques based on dynamic analysis, MH sampling based approaches in particular, tracking an ordered list of values that a variable gets assigned during a run of a program together with the distributions that were used to generate these values results in a correct sampling procedure for probabilistic programs. It might be tempting to consider a variable renaming scheme such as the static single assignment form (SSA) [5], or a variable indexing scheme [29] based on line numbers, distribution types, etc. (implemented in Stochastic Matlab). However, such schemes are inadequate to determine all values in the previous run that must be kept track of in order to propose a new value in the current run. This is clearly seen in Example 2 in Section 2.

For other inference techniques such as those based on static analysis [26] or Bayesian inference [21], it would be interesting to study analogous techniques for implementing provably correct inference algorithms.

7 Summary

We have highlighted the difficulties encountered in implementing a correct sampling-based inference engine for imperative probabilistic programs via several examples. We have designed an algorithm Infer$^{\checkmark}$ that overcomes these challenges, and generates samples from the correct distribution specified by the corresponding input probabilistic program.

Our algorithm is general and works for all probabilistic programs. We have also formally proved the correctness of our algorithm. We have implemented it in a tool called Infer$^{\checkmark}$, and have shown empirically that it works in all cases by comparing it with existing tools such as R2 and Stan. We have also shown that Infer$^{\checkmark}$ is a practical solution by evaluating it on real-world benchmarks.

Acknowledgements. We are grateful to Johannes Borgström for his valuable feedback on this paper.

References

1 Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. Measure transformer semantics for Bayesian machine learning. In *European Symposium on Programming (ESOP)*, pages 77–96, 2011.

2 Arun T. Chaganty, Aditya V. Nori, and Sriram K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *Artificial Intelligence and Statistics (AISTATS)*, 2013.

3 Siddhartha Chib and Edward Greenberg. Understanding the Metropolis-Hastings algorithm. *American Statistician*, 49(4):327–335, 1995.

4 Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. Bayesian inference for probabilistic programs via symbolic execution. In *Foundations of Software Engineering (FSE)*, 2013.

5 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Principles of Programming Languages (POPL)*, pages 25–35, 1989.

6 Andrew Gelman and Jennifer Hill. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge University Press, 2006.

7 W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. A language and program for complex Bayesian modelling. *The Statistician*, 43(1):169–177, 1994.

8 Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence (UAI)*, pages 220–229, 2008.

9 Andrew D. Gordon, Mihhail Aizatulin, Johannes Borgström, Guillaume Claret, Thore Graepel, Aditya V. Nori, Sriram K. Rajamani, and Claudio Russo. A model-learner pattern for Bayesian reasoning. In *Principles of Programming Languages (POPL)*, pages 403–416, 2013.

10 W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

11 Ralf Herbrich, Tom Minka, and Thore Graepel. TrueSkill: A Bayesian skill rating system. In *Neural Information Processing Systems (NIPS)*, pages 569–576, 2006.

12 Matthew D. Hoffman and Andrew Gelman. The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, in press, 2013.

13 Chung-Kil Hur, Aditya V. Nori, Sriram K. Rajamani, and Selva Samuel. Slicing probabilistic programs. In *Programming Languages Design and Implementation (PLDI)*, 2014.

14 S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, and P. Domingos. The Alchemy system for statistical relational AI, 2007. `http://alchemy.cs.washington.edu`.

15 D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

16 D. Koller, D. A. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *National Conference on Artificial Intelligence (AAAI)*, pages 740–747, 1997.

17 Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Science (JCSS)*, 22:328–350, 1981.

18 Jun S. Liu. Metropolized independent sampling with comparisons to rejection sampling and importance sampling. *Statistics and Computing*, 6(2):113–119, 1996.

19 David J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.

20 P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa. Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. *Journal of Computer Security*, 2013.

21 Tom Minka, John Winn, John Guiver, and Anitha Kannan. Infer.NET 2.3, November 2009. Software available from `http://research.microsoft.com/infernet`.

22 Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. R2: An efficient MCMC sampler for probabilistic programs. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2014.

23 Judea Pearl. *Probabilistic Reasoning in Intelligence Systems*. Morgan Kaufmann, 1996.

24 Avi Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*. MIT Press, 2007.

25 Walter Rudin. *Principles of mathematical analysis*. McGraw-Hill, 1976.

26 S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis of probabilistic programs: Inferring whole program properties from finitely many executions. In *Programming Languages Design and Implementation (PLDI)*, 2013.

27 Luke Tierney. Markov chains for exploring posterior distributions. *Annals of Statistics*, 22(4):1701–1728, 1994.

28 Gero Walter and Thomas Augustine. Bayesian linear regression - different conjugate models and their (insensitivity) to prior-data conflict. Technical report, University of Munich, TR-069, 2009.

29 David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Artificial Intelligence and Statistics (AISTATS)*, 2011.