R2: An Efficient MCMC Sampler for Probabilistic Programs

Aditya V. Nori
Microsoft Research
"Vigyan", #9, Lavelle Road
Bangalore 560 001, India
adityan@microsoft.com

Chung-Kil Hur*
Department of Computer Science and Engineering
Seoul National University
Seoul 151-744, Republic of Korea
gil.hur@cse.snu.ac.kr

Sriram K. Rajamani and **Selva Samuel**
Microsoft Research
"Vigyan", #9, Lavelle Road
Bangalore 560 001, India
{*sriram, t-ssamue*}*@microsoft.com*

Abstract

We present a new Markov Chain Monte Carlo (MCMC) sampling algorithm for probabilistic programs. Our approach and tool, called R2, has the unique feature of employing program analysis in order to improve the efficiency of MCMC sampling. Given an input program P, R2 propagates observations in P backwards to obtain a semantically equivalent program P' in which every probabilistic assignment is immediately followed by an observe statement. Inference is performed by a suitably modified version of the Metropolis-Hastings algorithm that exploits the structure of the program P'. This has the overall effect of preventing rejections due to program executions that fail to satisfy observations in P. We formalize the semantics of probabilistic programs and rigorously prove the correctness of R2. We also empirically demonstrate the effectiveness of R2—in particular, we show that R2 is able to produce results of similar quality as the CHURCH and STAN probabilistic programming tools with much shorter execution time.

1 Introduction

Probabilistic programs are "usual" programs (written in languages such as C, Java, LISP or ML) with two added features: (1) the ability to draw values at random from distributions, and (2) the ability to condition values of variables in a program via observations. Unlike usual programs that are run to produce outputs, the goal of a probabilistic program is to model probability distributions succinctly and implicitly. *Inference* for probabilistic programs is the problem of computing an explicit representation of the probability distribution implicitly specified by a probabilistic program. Over the past several years, a variety of probabilistic programming languages and inference systems have been proposed (Gilks, Thomas, and Spiegelhalter 1994; Koller, McAllester, and Pfeffer 1997; Minka et al. 2009; Pfeffer 2007a; Goodman et al. 2008; Kok et al. 2007).

*Supported by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning (MSIP) / National Research Foundation of Korea (NRF) (Grant NRF-2008-0062609).

Since probabilistic programs are executable, sampling can be performed by repeatedly executing such programs. However, in the case of programs with a large number of random variables (representing a multivariate distribution), and observations (potentially representing low probability evidence), naive execution results in frequent rejections (due to generated states not satisfying observations), and an impractically large number of samples to infer the correct distribution. Consequently, efficient sampling techniques for probabilistic programs are a topic of active research (Wingate et al. 2011; Wingate, Stuhlmüller, and Goodman 2011; Pfeffer 2007b; Goodman et al. 2008).

We present a new approach to perform Markov Chain Monte Carlo (MCMC) sampling for probabilistic programs. Our approach and tool, called R2, consists of the following steps:

1. Propagation of observations back through the program using the *pre-image operation* (Dijkstra 1976) and placing an observe statement immediately next to every probabilistic assignment. This transformation preserves program semantics (formally defined in Section 4), and helps perform efficient sampling (defined in the next step).
2. Perform a modified Metropolis-Hastings (MH) sampling (Chib and Greenberg 1995) over the transformed probabilistic program. The modifications exploit the structure in the transformed programs that observe statements immediately follow probabilistic assignments, and sample from sub-distributions in order to avoid rejections.

The above two steps prevent rejections due to executions that fail to satisfy observations, and significantly improves the number of accepted MH samples in a given time budget. In contrast, previous approaches (Wingate, Stuhlmüller, and Goodman 2011; Wingate et al. 2011; Pfeffer 2007b; Goodman et al. 2008; Hoffman and Gelman 2013) have not specifically addressed rejections due to failing observations.

In Section 3, we formalize the semantics of probabilistic programs and this is used to prove the correctness of R2 in Section 4. In Section 5, we empirically demonstrate the effectiveness of R2. In particular, we compare R2 with the CHURCH (Goodman et al. 2008) and STAN (Hoffman and Gelman 2013) probabilistic programming tools on a number

```
1:  bool earthquake, burglary, alarm, phoneWorking,
    maryWakes,called;
2:  earthquake = Bernoulli(0.001);
3:  burglary = Bernoulli(0.01);
4:  alarm = earthquake || burglary;
5:  if (earthquake)
6:    phoneWorking = Bernoulli(0.6);
7:  else
8:    phoneWorking = Bernoulli(0.99);
9:  if (alarm && earthquake)
10:   maryWakes = Bernoulli(0.8);
11: else if (alarm)
12:   maryWakes = Bernoulli(0.6);
13: else
14:   maryWakes = Bernoulli(0.2);
15: called = maryWakes && phoneWorking;
16: observe(called);
17: return burglary;
```

Figure 1: Pearl's burglar alarm example.

of probabilistic program benchmarks—our empirical results show that R2 is able to produce results of similar quality as CHURCH and STAN with much shorter execution time.

Related work. There has been prior work on exploiting program structure to perform efficient sampling, both in the context of importance sampling and MCMC sampling. BLOG (Milch and Russell 2006) uses program structure to come up with good proposal distributions for MCMC sampling. Wingate et al. (Wingate et al. 2011) use nonstandard interpretations during runtime execution to compute derivatives, track provenance, and use these computations to improve the efficiency of MCMC sampling. Moldovan et al. (Moldovan et al. 2013) introduce an MCMC algorithm for estimating conditional probabilities from an AND/OR tree for the probabilistic logic programming language ProbLog (Raedt, Kimmig, and Toivonen 2007).

Pfeffer (Pfeffer 2007b) presents several structural heuristics (such as conditional checking, delayed evaluation, evidence collection and targeted sampling) to help make choices during sampling that are less likely to get rejected by observations during importance sampling. Chaganty et al. (Chaganty, Nori, and Rajamani 2013) generalize these techniques uniformly using pre-image transformations on observations to perform efficient importance sampling for straight-line programs.

Our work differs from this work in two broad ways. First, we define pre-image as a separate transformation on whole programs (as opposed to simple straight-line programs or paths), and then use a suitably modified version of MH sampling for the transformed programs. In contrast, Chaganty et al. use a testing routine to collect straight-line programs (without using pre-image transformation) and rejections can happen during this process if testing is used, or expensive symbolic execution techniques need to be used to split a program into paths. Furthermore, splitting the program into individual paths can entail a huge compilation cost as well as inefficient sampling code. Second, our sampler is based on MH sampling that exploits knowledge from previous samples, whereas Chaganty et al. is based on importance sampling that is agnostic to previous samples or states.

```
1:  bool earthquake, burglary, alarm, phoneWorking,
    maryWakes,called;
2:  earthquake = Bernoulli(0.001);
3:  burglary = Bernoulli(0.01);
4:  alarm = earthquake || burglary;
5:  if (earthquake) {
6:    phoneWorking = Bernoulli(0.6);
7:    observe(phoneWorking);
8:  }
9:  else {
10:   phoneWorking = Bernoulli(0.99);
11:   observe(phoneWorking);
12: }
13: if (alarm && earthquake){
14:   maryWakes = Bernoulli(0.8);
15:   observe(maryWakes && phoneWorking);
16: }
17: else if (alarm){
19:   maryWakes = Bernoulli(0.6);
20:   observe(maryWakes && phoneWorking);
21: }
22: else {
23:   maryWakes = Bernoulli(0.2);
24:   observe(maryWakes && phoneWorking);
25: }
26: called = maryWakes && phoneWorking;
27: return burglary;
```

Figure 2: Example after the PRE analysis.

2 Overview

In this section, we informally explain the main ideas of R2.

Consider the probabilistic program shown in Figure 1, originally from Pearl's work (Pearl 1996). This program has probabilistic assignment statements which draw values from distributions. For instance, in line 2, the statement "`earthquake` = `Bernoulli`(`0.001`)" draws a value from a Bernoulli distribution with mean `0.001` and assigns it to the variable `x`. The program also has an observe statement "`observe`(`called`)" (line 16) that is used to condition the value of the variable `called`—this statement blocks all executions of the program that do not satisfy the boolean expression (`called` = `true`) at line 16.

The meaning of a probabilistic program is the probability distribution over output values returned by the program with respect to the implicit probability distribution that the program represents. In this example, the variable `burglary` is returned by the program and we are interested in estimating its probability distribution. Naive sampling, which corresponds to repeatedly running the program can result in rejected samples leading to wasteful computation and subsequently loss in efficiency.

R2 is a sampling algorithm that can exploit the structure that is present in probabilistic programs in order to improve the efficiency and convergence of sampling. R2 consists of two main steps: (1) First, a PRE analysis, which hoists all conditions in observe statements to probabilistic assignment statements (using pre-image analysis) is applied to transform the input probabilistic program, and (2) next, inference is performed on the transformed program using a modified MH sampling algorithm that avoids rejecting samples by using truncated proposal distributions.

x	$\in$	Vars	
$\mathbf{uop}$	::=	$\cdots$	C unary operators
$\mathbf{bop}$	::=	$\cdots$	C binary operators
φ, ψ	::=	$\cdots$	logical formula
$\mathcal{E}$	::=		expressions
		$\mid x$	variable
		$\mid c$	constant
		$\mid \mathcal{E}_1 \ \mathbf{bop} \ \mathcal{E}_2$	binary operation
		$\mid \mathbf{uop} \ \mathcal{E}_1$	unary operation

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

Figure 3: Syntax of PROB.

For the program shown in Figure 1, R2 first performs the PRE analysis in order to obtain the transformed program shown in Figure 2. Intuitively, the PRE analysis computes conditions under which the observe statement in line 16 can be satisfied. For instance, the PRE analysis determines that to satisfy `observe(called)` in line 16, the program state must satisfy `maryWakes&&phoneWorking` at line 15 (which we will call a pre-image).

The PRE analysis is essentially a *backward* analysis, and it places, next to each probabilistic assignment, the propagated pre-image as an observe statement. For instance, right after each probabilistic assignment for `maryWakes`, the PRE analysis introduces the observe statement with the predicate `maryWakes&&phoneWorking`. This can be seen in Figure 2, which is the program obtained by applying the PRE analysis to Pearl's burglar alarm example. An interesting consequence is that if every probabilistic assignment in Figure 2 is executed such that the resulting sample satisfies the observe statement immediately following it, then the original observe statement in line 16 of Figure 1 is guaranteed to be satisfied.

In Section 5, we show that the PRE analysis is semantics preserving (in other words, the probability distributions of `burglary` in the programs shown in Figures 1 and 2 are equal)—this is crucial for proving the correctness of R2.

Next, R2 performs sampling over the transformed program in order to compute the probability distribution of `burglary`. This is done by using a modified version of the MH sampling algorithm that truncates the distribution in each probabilistic assignment statement with the condition in the observe statement following it. More precisely, in the modified MH algorithm (described in Section 4), both the proposal distribution and the target density function at each probabilistic assignment statement are modified to use truncated distributions. This ensures that the values drawn from the truncated distributions are such that every program execution is a valid one (that is, the execution satisfies all observe statements), thus avoiding wasteful rejected samples.

Though the example used in this section is a loop-free program and uses only discrete distributions, our techniques work for programs with loops, and programs that use continuous distributions. Loops with an a priori fixed iteration count can be trivially handled by unrolling the loop. For general loops with unknown iteration counts, the PRE analysis can still be done if the user supplies a loop invariant. More details can be found in Section 4.

3 Probabilistic Programs

Our probabilistic programming language PROB is a C-like imperative programming language with two additional constructs:

1. The *probabilistic assignment statement* "$\texttt{x} \sim \texttt{Dist}(\bar{\theta})$" draws a sample from a distribution `Dist` with a vector of parameters $\bar{\theta}$, and assigns it to the variable x. For instance, "$\texttt{x} \sim \texttt{Gaussian}(\mu, \sigma^2)$" draws a value from a Gaussian distribution with mean μ and variance σ^2, and assigns it to the variable x.
2. The *observe statement* "$\texttt{observe}(\varphi)$" conditions a distribution with respect to a predicate or condition φ that is defined over the variables in the program. In particular, every valid execution of the program must satisfy all conditions in observe statements that occur along the execution.

The syntax of PROB is formally described in Figure 3. A program consists of statements and a return expression. Variables have base types such as int, bool, float and double. Statements include primitive statements (skip, deterministic assignment, probabilistic assignment, observe) and composite statements (sequential composition, conditionals and loops). We omit the discussion of arrays, pointers, structures and function calls in the language since their treatment does not introduce any additional challenges to the definition of the semantics of PROB.

The semantics of PROB is described in Figure 4. A state σ of a program is a (partial) valuation to all its variables. The set of all states (which can be infinite) is denoted by Σ. We also consider the natural lifting of $\sigma : \text{Vars} \rightharpoonup \text{Val}$ to expressions $\sigma : \text{Exprs} \to \text{Val}$. We make this lifting a total function by assuming default values for uninitialized variables. The definition of the lifting σ for constants, unary and binary operations is standard.

The meaning of a probabilistic statement $\mathcal{S}$ is the probability distribution over all possible output states of $\mathcal{S}$ for any given initial state σ. It is standard to represent a probability distribution on a set X as a function calculating the expected value of f w.r.t. the distribution on X for any given return function $f \in X \to [0, 1]$, where $[0, 1]$ is the unit interval (*i.e.,* the set of real numbers between 0 and 1, inclusive). Thus, the denotational semantics $[\![\mathcal{S}]\!](f)(\sigma)$ gives the expected value of return function $f \in \Sigma \to [0, 1]$ when S is executed with initial state σ. The semantics is completely specified using the rules in Figure 4.

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

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

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
$[\![\mathcal{S}\ \texttt{return}\ \mathcal{E}]\!] \in (\mathbb{R} \to [0,1]) \to [0,1]$

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

where $\bot$ denotes the empty state.

Figure 4: Denotational Semantics of PROB.

The `skip` statement merely applies the return function f to the input state σ, since the statement does not change the input state. The deterministic assignment statement first transforms the state σ by executing the assignment and then applies f. The meaning of the probabilistic assignment is the expected value obtained by sampling v from the distribution `Dist`, executing the assignment with v as the RHS value, and applying f on the resulting state (the expectation is the integral over all possible values v). The `observe` statement functions like a `skip` statement if the expression φ evaluates to `true` in the initial state σ, and returns the value 0 otherwise. The sequential and conditional statements behave as expected and the while-do loop has a standard fixpoint semantics.

Due to the presence of non-termination and observe statements, the semantics of statements shown in Figure 4 is unnormalized. The normalized semantics for programs is obtained by appropriately performing the normalization operation as shown in the second part of Figure 4. The $\bot$ in the figure is a default initial state. Note that the exact initialization does not matter as all programs that we consider are closed programs with no inputs.

4 The R2 Algorithm

In this section, we describe the two main steps of R2: (1) The pre-image analysis, and (2) the MH sampling algorithm.

Pre-Image Analysis

The pre-image analysis propagates predicates from `observe` statements backward through the program using a particular variant of the pre-image operator PRE (Dijkstra 1976) for deterministic programs. We describe the transformation for various statement types in Figure 5. The operator is carefully designed to have the property that the transformed program is equivalent to the input program, and it helps identify samples that are going to be rejected by observe statements early in the execution.

$$\begin{aligned}
&\textsc{Pre}(x = \mathcal{E}, \varphi) && = (x = \mathcal{E}, \varphi[\mathcal{E}/x]) \\
&\textsc{Pre}(\texttt{skip}, \varphi) && = (\texttt{skip}, \varphi) \\
&\textsc{Pre}(\mathcal{S}_1; \mathcal{S}_2, \varphi) && = \textbf{let}\ (\mathcal{S}_2', \varphi') = \textsc{Pre}(\mathcal{S}_2, \varphi)\ \textbf{and} \\
& && \quad \textbf{let}\ (\mathcal{S}_1', \varphi'') = \textsc{Pre}(\mathcal{S}_1, \varphi')\ \textbf{in} \\
& && \quad (\mathcal{S}_1'; \mathcal{S}_2', \varphi'') \\
&\textsc{Pre}(\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1 && = \textbf{let}\ (\mathcal{S}_1', \varphi_1) = \textsc{Pre}(\mathcal{S}_1, \varphi)\ \textbf{and} \\
&\quad \texttt{else}\ \mathcal{S}_2, \varphi) && \quad \textbf{let}\ (\mathcal{S}_2', \varphi_2) = \textsc{Pre}(\mathcal{S}_2, \varphi)\ \textbf{in} \\
& && \quad (\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1'\ \texttt{else}\ \mathcal{S}_2', \\
& && \quad (\mathcal{E} \wedge \varphi_1) \vee (\neg\mathcal{E} \wedge \varphi_2)) \\
&\textsc{Pre}(\texttt{observe}\ (\psi), \varphi) && = (\texttt{observe}(\psi), \varphi \wedge \psi) \\
&\textsc{Pre}(x \sim \texttt{Dist}(\bar{\theta}), \varphi) && = (x \sim \texttt{Dist}(\bar{\theta}); \texttt{observe}(\varphi), \exists x.\varphi) \\
&\textsc{Pre}(\texttt{while}\ \{\psi\}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}, \varphi) && = \textbf{let}\ (\mathcal{S}', \psi') = \textsc{Pre}(\mathcal{S}, \psi)\ \textbf{in} \\
& && \quad \textbf{assert}(\mathcal{E} \wedge \psi' \implies \psi); \\
& && \quad \textbf{assert}(\neg\mathcal{E} \wedge \varphi \implies \psi); \\
& && \quad (\texttt{while}\ \{\psi\}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}', \psi)
\end{aligned}$$

Figure 5: Given a statement $\mathcal{S}$ and a predicate φ defined over program variables, $\textsc{Pre}(S, \varphi)$ is a pair $(\hat{\mathcal{S}}, \hat{\varphi})$ where $\hat{\mathcal{S}}$ maps every sample statement with a pre-image predicate (via an observe statement immediately following the sample statement), and $\hat{\varphi}$ is a pre-image of φ over $\mathcal{S}$. We assume that every while statement is annotated with a loop invariant ψ.

We formally state the definition of the PRE operator, and show that the PRE operator does not change the semantics of the input program. More precisely, the operator $\textsc{Pre}(\mathcal{S}, \varphi)$ returns a pair $(\hat{\mathcal{S}}, \hat{\varphi})$, where $\hat{\mathcal{S}}$ is a program transformation of $\mathcal{S}$ such that every sample statement in $\hat{S}$ is immediately followed by an observe statement with a corresponding pre-image predicate. Suppose we have that $\textsc{Pre}(\mathcal{S}, \texttt{true}) = (\hat{\mathcal{S}}, \hat{\varphi})$. We show that the statements $\mathcal{S}$ and $\hat{\mathcal{S}}$ are semantically equivalent.

The PRE of a predicate φ with respect to an assignment statement $x = \mathcal{E}$ is defined to be $\varphi[\mathcal{E}/x]$) (this denotes the predicate obtained by replacing all occurrence of x in φ with $\mathcal{E}$). The PRE operator does nothing with `skip` statements (and it propagates the input predicate), and it treats sequential and conditional composition in the usual way, as one would expect any pre-image operator to work.

The PRE of a predicate φ with respect to the statement $\texttt{observe}(\psi)$ is the conjunction φ and ψ. The PRE operator treats probabilistic assignments like nondeterministic assignments, using existential quantification of the assigned variable. However, in addition, the PRE operator adds an extra observe statement $\texttt{observe}(\varphi)$ which is placed immediately after the probabilistic assignment. Intuitively, the condition φ is propagated by the PRE operator to another equivalent observe statement right next to each probabilistic assignment. The goal of this `observe` statement is to precisely reject those states that will be rejected by the original observe statements in the program.

The PRE operator on a while-do loop uses an user-provided annotation ψ, which is a special kind of loop invariant that is guaranteed not to block any runs that will be accepted by observe statements after the loop. As long as the conditions $\mathcal{E} \wedge \psi' \implies \psi$ and $\neg\mathcal{E} \wedge \varphi \implies \psi$ are satisfied by the annotated loop invariant ψ (where $(\mathcal{S}', \psi') = \textsc{Pre}(\mathcal{S}, \psi)$), we can show (see below) that the program pro-

duced by the PRE operator is equivalent to the input program.

In practice (and for our benchmarks), most loops in probabilistic programs have fixed deterministic iterations and can be unrolled into non-loopy code—this is precisely what the R2 implementation does. For unbounded loops, R2 bounds the loops (if the loop invariant is not supplied) and computes the PRE transformation. Automatically inferring the loop invariant ψ for a probabilistic loop is interesting future work, and for the rest of the paper, we will assume the availability of such invariants.

Definition 1 *The function $f : \Sigma \to [0,1]$ satisfies φ, if $\forall \sigma. f(\sigma) \neq 0 \implies \sigma(\varphi) = \texttt{true}$.*

The following lemma is used to prove the correctness of the algorithm.

Lemma 1 *Let $(\hat{\mathcal{S}}, \hat{\varphi}) = \mathrm{Pre}(\mathcal{S}, \varphi)$. Then for any f satisfying φ, we have $[\![\mathcal{S}]\!](f) = [\![\hat{\mathcal{S}}]\!](f)$, and $[\![\mathcal{S}]\!](f)$ satisfies $\hat{\varphi}$.*

Proof: We prove the lemma by induction on the structure of $\mathcal{S}$.

• When $\mathcal{S}$ is `skip`:
The lemma holds trivially.

• When $\mathcal{S}$ is $x = \mathcal{E}$:
We have $\hat{\mathcal{S}} = \mathcal{S}$ and $\hat{\varphi} = \varphi[\mathcal{E}/x]$. If $[\![\mathcal{S}]\!](f)(\sigma) = f(\sigma[x \leftarrow \sigma(\mathcal{E})]) \neq 0$, then we have $\sigma[x \leftarrow \sigma(\mathcal{E})](\varphi) = \texttt{true}$. Thus we have

$$\sigma(\hat{\varphi}) = \sigma(\varphi[\mathcal{E}/x]) = \sigma[x \leftarrow \sigma(\mathcal{E})](\varphi) = \texttt{true}\ .$$

• When $\mathcal{S}$ is $x \sim \texttt{Dist}(\bar{\theta})$:
We have $\hat{\mathcal{S}} = (x \sim \texttt{Dist}(\bar{\theta}); \texttt{observe}(\varphi))$ and $\hat{\varphi} = \exists x.\, \varphi$. Since f satisfies φ, one can easily show $[\![\mathcal{S}]\!](f) = [\![\hat{\mathcal{S}}]\!](f)$. If $[\![\mathcal{S}]\!](f)(\sigma) = \int_{v \in \mathrm{Val}} \texttt{Dist}(\sigma(\bar{\theta}))(v) \times f(\sigma[x \leftarrow v])\, dv \neq 0$, then we have v such that $f(\sigma[x \leftarrow v]) \neq 0$ and thus $\sigma[x \leftarrow v](\varphi) = \texttt{true}$. Thus we have

$$\sigma[\hat{\varphi}] = \sigma[\exists x.\, \varphi] = \texttt{true}\ .$$

• When $\mathcal{S}$ is $\texttt{observe}(\psi)$:
We have $\hat{\mathcal{S}} = \mathcal{S}$ and $\hat{\varphi} = \varphi \wedge \psi$. One can easily see that $[\![\texttt{observe}(\psi)]\!](f)$ satisfies $\varphi \wedge \psi$.

• When $\mathcal{S}$ is $\mathcal{S}_1; \mathcal{S}_2$:
We have $\hat{\mathcal{S}} = \hat{\mathcal{S}}_1; \hat{\mathcal{S}}_2$ with $(\hat{\mathcal{S}}_2, \varphi') = \mathrm{Pre}(\mathcal{S}_2, \varphi)$ and $(\hat{\mathcal{S}}_1, \hat{\varphi}) = \mathrm{Pre}(\mathcal{S}_1, \varphi')$. By the induction hypothesis, we have $[\![\mathcal{S}_2]\!](f) = [\![\hat{\mathcal{S}}_2]\!](f)$ and $[\![\mathcal{S}_2]\!](f)$ satisfies φ'. Again by the induction hypothesis, we have $[\![\mathcal{S}_1]\!]([\![\mathcal{S}_2]\!](f)) = [\![\hat{\mathcal{S}}_1]\!]([\![\mathcal{S}_2]\!](f))$ and $[\![\mathcal{S}_1]\!]([\![\mathcal{S}_2]\!](f))$ satisfies $\hat{\varphi}$. Thus we have $[\![\mathcal{S}]\!](f) = [\![\hat{\mathcal{S}}]\!](f)$ and $[\![\mathcal{S}]\!](f)$ satisfies $\hat{\varphi}$.

• When $\mathcal{S}$ is $\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2$:
We have

$$\hat{\mathcal{S}} = \texttt{if}\ \mathcal{E}\ \texttt{then}\ \hat{\mathcal{S}}_1\ \texttt{else}\ \hat{\mathcal{S}}_2 \text{ and } \hat{\varphi} = (\mathcal{E} \wedge \varphi_1) \vee (\neg\mathcal{E} \wedge \varphi_2)$$

with $(\hat{\mathcal{S}}_1, \varphi_1) = \mathrm{Pre}(\mathcal{S}_1, \varphi)$ and $(\hat{\mathcal{S}}_2, \varphi_2) = \mathrm{Pre}(\mathcal{S}_2, \varphi)$. By the induction hypothesis, we have $[\![\mathcal{S}_1]\!](f) = [\![\hat{\mathcal{S}}_1]\!](f)$ and $[\![\mathcal{S}_1]\!](f)$ satisfies φ_1. Again by the induction hypothesis, we have $[\![\mathcal{S}_2]\!](f) = [\![\hat{\mathcal{S}}_2]\!](f)$ and $[\![\mathcal{S}_2]\!](f)$ satisfies φ_2. Thus we have $[\![\mathcal{S}]\!](f) = [\![\hat{\mathcal{S}}]\!](f)$. Suppose $[\![\mathcal{S}]\!](f)(\sigma) \neq 0$. If $\sigma(\mathcal{E}) = \texttt{true}$, then we have $[\![\mathcal{S}_1]\!](f)(\sigma) \neq 0$ and thus $\sigma(\varphi_1) = \texttt{true}$. If $\sigma(\mathcal{E}) \neq \texttt{true}$, then we have $[\![\mathcal{S}_2]\!](f)(\sigma) \neq 0$ and thus $\sigma(\varphi_2) = \texttt{true}$. So, we have $\sigma(\hat{\varphi}) = \sigma((\mathcal{E} \wedge \varphi_1) \vee (\neg\mathcal{E} \wedge \varphi_2)) = \texttt{true}$.

• When $\mathcal{S}$ is $\texttt{while}\ \{\psi\}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}_0$:
We have $\hat{\mathcal{S}} = \texttt{while}\ \{\psi\}\ \mathcal{E}\ \texttt{do}\ \hat{\mathcal{S}}_0$ and $\hat{\varphi} = \psi$ with $(\hat{\mathcal{S}}_0, \psi') = \mathrm{Pre}(\mathcal{S}_0, \psi)$. To show the lemma, it suffices to show that for any n,

- $[\![\texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \mathcal{S}_0]\!](f) = [\![\texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \hat{\mathcal{S}}_0]\!](f)$; and
- $[\![\texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \mathcal{S}_0]\!](f)$ satisfies ψ.

We prove this by induction on n. It holds trivially for the base case. For the step case, we need to show:

(1) $[\![\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_0; \texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \mathcal{S}_0\ \texttt{else skip}]\!](f)$
$= [\![\texttt{if}\ \mathcal{E}\ \texttt{then}\ \hat{\mathcal{S}}_0; \texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \hat{\mathcal{S}}_0\ \texttt{else skip}]\!](f)$; and

(2) $[\![\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_0; \texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \mathcal{S}_0\ \texttt{else skip}]\!](f)$ satisfies ψ.

By the induction hypotheses we have

$$\begin{aligned}[\![\mathcal{S}_0]\!]([\![\texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \mathcal{S}_0]\!](f)) &= [\![\hat{\mathcal{S}}_0]\!]([\![\texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \mathcal{S}_0]\!](f)) \\ &= [\![\hat{\mathcal{S}}_0]\!]([\![\texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \hat{\mathcal{S}}_0]\!](f)),\end{aligned}$$

from which (1) follows. Again by the induction hypotheses, we have that $[\![\mathcal{S}_0]\!]([\![\texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \mathcal{S}_0]\!](f))$ satisfies ψ'. To show (2), suppose

$$[\![\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_0; \texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \mathcal{S}_0\ \texttt{else skip}]\!](f)(\sigma) \neq 0\ .$$

If $\sigma(\mathcal{E}) = \texttt{true}$, we have $[\![\mathcal{S}_0]\!]([\![\texttt{while}\ \mathcal{E}\ \texttt{do}_n\ \mathcal{S}_0]\!](f))(\sigma) \neq 0$ and thus $\sigma(\psi') = \texttt{true}$. Since $\mathcal{E} \wedge \psi' \Rightarrow \psi$, we have $\sigma(\psi) = \texttt{true}$. If $\sigma(\mathcal{E}) \neq \texttt{true}$, we have $f(\sigma) \neq 0$ and thus $\sigma(\varphi) = \texttt{true}$. Since $\neg\mathcal{E} \wedge \varphi \Rightarrow \psi$, we have $\sigma(\psi) = \texttt{true}$. ■

The following theorem proves that the PRE operation is correct.

Theorem 1 *For any probabilistic program $\mathcal{S}$ $\texttt{return}\ \mathcal{E}$, we have*

$$[\![\mathcal{S}\ \texttt{return}\ \mathcal{E}]\!] = [\![\hat{\mathcal{S}}\ \texttt{return}\ \mathcal{E}]\!]$$

for $(\hat{\mathcal{S}}, \_) = \mathrm{PRE}(\mathcal{S}, \texttt{true})$.

Proof: For any return function f, we have $[\![\mathcal{S}]\!](f) = [\![\hat{\mathcal{S}}]\!](f)$ by Lemma 1 since f satisfies `true`. Thus by definition we have $[\![\mathcal{S}\ \texttt{return}\ \mathcal{E}]\!] = [\![\hat{\mathcal{S}}\ \texttt{return}\ \mathcal{E}]\!]$. ■

Metropolis-Hastings Sampling

We show that the PRE transformed probabilistic program can be profitably used for efficient MCMC sampling. The PRE transformation places an observe statement next to every probabilistic assignment statement, and in this section we show how to use this structure to improve the efficiency of MCMC sampling.

The Metropolis-Hastings or MH algorithm (Chib and Greenberg 1995) takes a probability or target density $P(\bar{x})$ as input and returns samples that are distributed according to this density. These samples can be further used to compute any estimator such as expectation of a function with respect to the target density $P(\bar{x})$. Two key components of the MH algorithm are:

1. For every probabilistic assignment statement "$x \sim P$", where x is a variable which takes its values from a distribution P, the *proposal distribution* $Q(x_{old} \to x_{new})$ which is used to pick a new value x_{new} for the variable x by appropriately perturbing its old value x_{old}.
2. The parameter β that is used to decide whether to accept or reject a new sampled value for x, and is defined as follows:

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

The sample is accepted if a random value drawn from the uniform distribution over $[0, 1]$ is less than β, otherwise it is rejected.

Let $\mathrm{P}_{|\varphi}$ denote the truncated distribution that results from restricting the domain of the distribution P to a predicate φ. That is, if $\varphi(x)$ is false, then $\mathrm{P}_{|\varphi}(x) = 0$. Otherwise, if $\varphi(x)$ is true, then $\mathrm{P}_{|\varphi}(x) = \mathrm{P}(x)/Z$, where Z is an appropriate normalization factor. R2 employs the following modified MH steps:

1. For every sample statement and observation "$x \sim P; observe(\varphi)$", the proposal distribution $Q_{|\varphi}(x_{old} \to x_{new})$ is truncated according to the condition φ.
2. The parameter β is defined as follows (note that both the target and proposal distributions are truncated):

$$\beta = \min\left\{1, \frac{P_{|\varphi}(x_{new}) \times Q_{|\varphi}(x_{new} \to x_{old})}{P_{|\varphi}(x_{old}) \times Q_{|\varphi}(x_{old} \to x_{new})}\right\}$$

The modified MH algorithm on the transformed program is easily shown to be equivalent to the standard MH algorithm, since the probabilistic assignment statement $x \sim P$ followed by the statement $observe(\varphi)$ is equivalent to the statement $x \sim P_{|\varphi}$. Since the proposal distribution is truncated according to the condition φ, we have that samples produced by the modified MH algorithm *always* satisfy the observe statement `observe`(φ) immediately following the probabilistic assignment, and hence there is no possibility of rejection.

It is interesting to note the above modification to the MH algorithm can also be applied to other variants such as Hamiltonian Monte Carlo (Neal 2010) which suggests that the PRE transformation can be used profitably by other probabilistic programming tools such as CHURCH (Goodman et al. 2008) and STAN (Hoffman and Gelman 2013).

5 Empirical Evaluation

In this section, we empirically evaluate R2 and compare it with two state-of-the-art probabilistic programming tools CHURCH and STAN over a number of benchmarks. All experiments were performed on a 2.5 GHz Intel system with 8 GB RAM running Microsoft Windows 8. R2 is implemented in C++ and uses the `Z3` theorem prover (de Moura and Bjorner 2008) in order to represent and manage preimage predicates. We evaluated R2 on the following benchmarks:

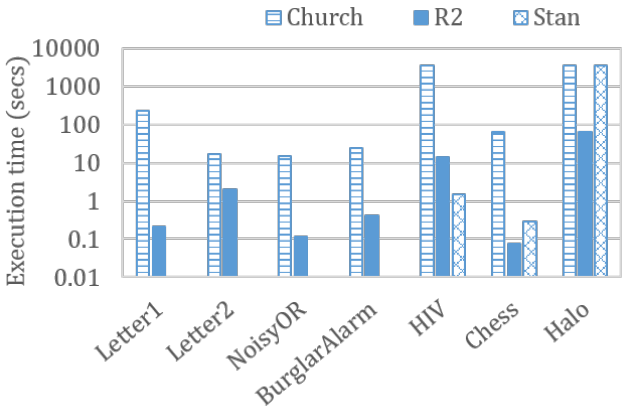

Figure 6: Evaluation results.

- `Letter1`, `Letter2`: These are programs representing the probabilistic model for a student getting a good reference letter (Koller and Friedman 2009).
- `NoisyOR`: Given a directed acyclic graph, each node is a noisy-or of its parents. Find the posterior probability of a node, given observations (Kiselyov and Shan 2009).
- `BurglarAlarm`: This is an adaptation of Pearl's example (Pearl 1996) where we want to estimate the probability of a burglary having observed a phone call (see Figure 1).
- `HIV`: This is a multi-level linear model with interaction and varying slope and intercept (Hoffman and Gelman 2013).
- `Chess`: This is skill rating system for a chess tournament consisting of 77 players and 2926 games (Herbrich, Minka, and Graepel 2006).
- `Halo`: This is a skill rating system for a tournament consisting of 31 teams, with at most 4 players per team, and 465 games played between teams (Herbrich, Minka, and Graepel 2006).

The benchmarks `Letter1`, `Letter2`, `NoisyOR` and `BurglarAlarm` are toy models, whereas the benchmarks `HIV`, `Chess` and `Halo` are larger models used in real-word applications.

Figure 6 summarizes the results obtained by running CHURCH, R2 and STAN over the seven benchmarks. The execution time is in seconds and shown in the figure using a logarithmic scale. R2 produces results with shorter execution time than CHURCH and STAN (all tools were run so that they produced the correct answer up to three significant digits). This difference can be attributed primarily to the PRE transformation described in Section 4. For the `HIV` and `Halo` benchmarks, CHURCH does not produce an answer and times out after one hour. STAN times out after one hour on the `Halo` benchmark. The STAN results for the benchmarks `Letter1`, `Letter2`, `NoisyOR` and `BurglarAlarm` are missing as it does not support models with discrete latent variables (Stan Development Team 2013).

It is important to note that R2 is at least 50 times faster than QI (Chaganty, Nori, and Rajamani 2013) on these benchmarks (with QI timing out on the `Chess` and `Halo`

benchmarks). This improvement in performance is due to the fact that R2 is able to improvise based on its current state in order to move into regions of high density.

It is also interesting to note that for the HIV benchmark, STAN performs better than R2. One of the reasons for this is that STAN is based on Hamiltonian Monte Carlo sampling (Neal 2010) which enables it to generate samples from areas of high probability. In contrast, R2 exhibits the diffusive behavior of random walks and moves slowly from one high probability region to another. As future work, we would like to extend the sampling technique in R2 to Hamiltonian Monte Carlo sampling (as described in Section 4).

6 Conclusion

We have presented a tool R2 that is based on a new MCMC algorithm for efficiently sampling from probabilistic programs. A unique feature of our algorithm is that it uses ideas from program analysis such as pre-image computation to avoid rejection due to conditioning in the program.

We have formalized the semantics of probabilistic programs and rigorously proved the correctness of R2. Our experimental results are also encouraging—we show that R2 is able to produce results of similar quality as state-of-the-art probabilistic programming tools such as CHURCH and STAN with much shorter execution time.

Acknowledgments

We are grateful to Deepak Vijaykeerthy for feedback on a draft of this paper.

References

Chaganty, A. T.; Nori, A. V.; and Rajamani, S. K. 2013. Efficiently sampling probabilistic programs via program analysis. *International Conference on Artificial Intelligence and Statistics (AISTATS)*.

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

de Moura, L., and Bjorner, N. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 337–340.

Dijkstra, E. W. 1976. *A Discipline of Programming*. Prentice Hall.

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

Goodman, N. D.; Mansinghka, V. K.; Roy, D. M.; Bonawitz, K.; and Tenenbaum, J. B. 2008. Church: a language for generative models. In *Uncertainty in Artificial Intelligence (UAI)*, 220–229.

Herbrich, R.; Minka, T.; and Graepel, T. 2006. TrueSkill: A Bayesian skill rating system. In *Neural Information Processing Systems (NIPS)*, 569–576.

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

Kiselyov, O., and Shan, C. 2009. Monolingual probabilistic programming using generalized coroutines. In *Uncertainty in Artificial Intelligence (UAI)*, 285–292.

Kok, S.; Sumner, M.; Richardson, M.; Singla, P.; Poon, H.; Lowd, D.; and Domingos, P. 2007. The Alchemy system for Statistical Relational AI. Technical report, University of Washington.

Koller, D., and Friedman, N. 2009. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press.

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

Milch, B., and Russell, S. J. 2006. General-purpose MCMC inference over relational structures. In *Uncertainty in Artificial Intelligence(UAI)*.

Minka, T.; Winn, J.; Guiver, J.; and Kannan, A. 2009. Infer.NET 2.3.

Moldovan, B.; Thon, I.; Davis, J.; and Raedt, L. D. 2013. MCMC estimation of conditional probabilities in probabilistic programming languages. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU)*, 436–448.

Neal, R. M. 2010. MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*.

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

Pfeffer, A. 2007a. The design and implementation of IBAL: A general-purpose probabilistic language. In *Statistical Relational Learning*, 399–432.

Pfeffer, A. 2007b. A general importance sampling algorithm for probabilistic programs. Technical report, Harvard University TR-12-07.

Raedt, L. D.; Kimmig, A.; and Toivonen, H. 2007. ProbLog: A probabilistic prolog and its application in link discovery. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2462–2467.

Stan Development Team. 2013. *Stan Modeling Language User's Guide and Reference Manual, Version 1.3*.

Wingate, D.; Goodman, N. D.; Stuhlmüller, A.; and Siskind, J. M. 2011. Nonstandard interpretations of probabilistic programs for efficient inference. In *Neural Information Processing Systems (NIPS)*, 1152–1160.

Wingate, D.; Stuhlmüller, A.; and Goodman, N. D. 2011. Lightweight implementations of probabilistic programming languages via transformational compilation. *International Conference on Artificial Intelligence and Statistics (AISTATS)* 15:770–778.