Semantics Sensitive Sampling for Probabilistic Programs

Aditya V. Nori
Microsoft Research
adityan@microsoft.com

Chung-Kil Hur
Seoul National University
gil.hur@snu.ac.kr

Sriram K. Rajamani
Microsoft Research
sriram@microsoft.com

Selva Samuel
Microsoft Research
t-ssamue@microsoft.com

Abstract

We present a new semantics sensitive sampling algorithm for probabilistic programs, which are "usual" programs endowed with statements to sample from distributions, and condition executions based on observations. Since probabilistic programs are executable, sampling can be performed by repeatedly executing them. However, in the case of programs with a large number of random variables and observations, naive execution does not produce high quality samples, and it takes an intractable number of samples in order to perform reasonable inference. Our MCMC algorithm called $\mathtt{S}^3$ tackles these problems using ideas from program analysis. First, $\mathtt{S}^3$ propagates observations back through the program in order to obtain a semantically equivalent program with conditional sample statements – this has the effect of preventing rejections due to executions that fail to satisfy observations. Next, $\mathtt{S}^3$ decomposes the probabilistic program into a set of straight-line programs, one for every valid program path, and performing Metropolis-Hastings sampling over each straight-line program independently. Sampling over straight-line programs has the advantage that random choices from previous executions can be re-used merely using the program counter (or line number) associated with each random choice. Finally, it combines the results from sampling each straight-line program (using appropriate weighting) to produce a result for the whole program. We formalize the semantics of probabilistic programs and rigorously prove the correctness of $\mathtt{S}^3$. We also empirically demonstrate the effectiveness of $\mathtt{S}^3$, and compare it with an importance sampling based tool over various benchmarks.

1 Introduction

Probabilistic programs are "usual" programs (written in languages like C, Java, LISP or ML) with two added constructs: (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. *Probabilistic inference* 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 [3, 4, 7, 8, 10, 12].

Since probabilistic programs are executable, sampling can be performed by repeatedly executing such programs. However, in the case of programs with large number of random variables (representing a multivariate distribution), and observations (potentially representing low probability evidence), naive execution does not produce high quality samples and it takes an impractically large number of samples to infer the correct distribution. Consequently, efficient sampling techniques for probabilistic programs are a topic of active research [4, 13–15].

We present a new approach to perform Markov Chain Monte Carlo (MCMC) sampling for probabilistic programs. Our approach, called $\mathtt{S}^3$, consists of three steps:

1. Propagate observations back through the program using pre-image transformation techniques (from program semantics) to conditioning on sampling statements. This transformation preserves program semantics (formally defined in Section 3.1), and helps efficiently compute a set of valid executions of the program and perform efficient sampling (defined in the next step).
2. Decompose a probabilistic program into a set of straight-line programs, one for every valid program path. Perform Metropolis Hastings (MH) sampling on each straight-line program independently, by constraining proposals generated for MH using the conditioning associated with the sampling statements (as computed by Step 1). The pre-image transformation (from Step 1) helps avoid rejections due to observations.
3. Combine the estimates from each straight-line program with appropriate weighting to produce estimates for the whole program.

Our approach offers two main advantages. First, the backward propagation of observations to sampling statements prevents 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 [4, 13–15] have not specifically addressed rejections due to failing observations. Second, by decomposition of a program into paths and performing MH on each individual path, we are able to "reuse" random choices from previous executions using merely the program counter (or line number) associated with each random choice. Path splitting also allows our algorithm to report multi-modal distributions as answers (one mode for every path in the program). In fact, since a straight-line program has fewer modes than the original program, performing MCMC on straight-line programs separately is more efficient than performing MCMC on the original program. In contrast, previous approaches [14, 15] deal with the whole program in which each execution can follow different control paths, involving considerable book-keeping, and our approach avoids these complications and inefficiencies by decomposing the program into paths, and performing MCMC on each path separately.

In Section 2, we formalize the semantics of probabilistic programs and rigorously prove the correctness of $\mathtt{S}^3$ which is presented in Section 3. In Section 4, we empirically demonstrate the effectiveness of $\mathtt{S}^3$ over a set of benchmarks. In particular, we compare $\mathtt{S}^3$ with the QI algorithm [1] – our empirical results show that $\mathtt{S}^3$ outperforms QI on a set of benchmarks and real world applications.

Though our current implementation is sequential, our algorithm is massively parallelizable: each path can be sampled independently, and within each path, we can independently perform sampling in parallel. Our formulation shows how to appropriately weight and combine each of the results from such parallel computations to produce a result for the whole program.

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 [9] uses program structure to come up with good proposal distributions for MCMC sampling. Wingate et al. [14] use nonstandard interpretations during runtime execution to compute derivatives, track provenance, and use these computations to improve the efficiency of MCMC sampling. Wingate et al. [15] name random choices according to their "structural position" in the path in order to "reuse" random choices from a previous execution during a subsequent execution. However, calculating structural position for arbitrary program paths is difficult, and involves considerable book-keeping, and our approach avoids these complications and inefficiencies by decomposing the program into paths, and performing MCMC on each path separately.

Pfeffer [13] 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. [1] generalize these

x	$\in$	Vars	
$\mathcal{T}$	$::=$	$\cdots$	C basic types
uop	$::=$	$\cdots$	C unary operators
bop	$::=$	$\cdots$	C binary operators
φ, ψ	$::=$	$\cdots$	logical formula
$\mathcal{D}$	$::=$	$\mid \mathcal{T} x_1, x_2, \ldots, x_n$	declaration
$\mathcal{E}$	$::=$		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
		$\mid x = \mathcal{E}$	deterministic assignment
		$\mid x \sim \texttt{Dist}(\bar{\theta})$	probabilistic assignment
		$\mid \texttt{observe}\ (\varphi)$	observe
		$\mid \texttt{skip}$	skip
		$\mid \mathcal{S}_1; \mathcal{S}_2$	sequential composition
		$\mid \texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2$	conditional composition
		$\mid \texttt{while}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}_1$	loop
$\mathcal{P}$	$::=$	$\mathcal{D}\ \mathcal{S}\ \texttt{return}\ (\mathcal{E})$	program

Figure 1: Syntax of PROB.

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 programs. Thus, we are able to first use pre-image on the whole program, split a program into straight line programs efficiently, and then use pre-image (again) on the straight-line programs to avoid rejections. 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. 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. Further, we also present a new method for combining the sub-expectations for straight-line programs in order to compute the expectation for the whole program.

2 Probabilistic Programs

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

1. The *sample 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 1. A program consists of variable declarations, a statement and a return expression. Variables have base types such as int, bool, float and double. Statements include primitive statements (deterministic assignment, probabilistic assignment, observe, skip) and composite statements (sequential composition, conditionals and loops). We omit the discussion of arrays, pointers, structures and function calls in the language. Our implementation, however, is able to handle all these features.

The meaning of a probabilistic program is the expected value of its return expression. A state σ of a program is a valuation to all its variables. The set of all states (which can be infinite) is denoted by Σ. The probabilistic semantics $(S, \sigma) \Downarrow^{\bar{v}} (p, \sigma')$ denotes that the outcome σ' of the program S with initial state σ is determined by a sequence of samples $\bar{v}$ with probability density p. The sequence of samples $\bar{v}$ belongs to the measure space $\mathbb{S}$ defined by: $\mathbb{S} = \biguplus_{n \geq 0} (\mathbb{N} \uplus \mathbb{R})^n$.

The operational semantics of PROB is completely specified using the following rules:

$$\begin{array}{rll}
(x = \mathcal{E}, \sigma) \Downarrow^{\epsilon} & (1, \sigma[x \leftarrow \sigma(\mathcal{E})]) & \\
(x \sim \texttt{Dist}(\bar\theta), \sigma) \Downarrow^{(v)} & (p, \sigma[x \leftarrow v]) & \text{if } v \in \mathbb{N} \uplus \mathbb{R} \wedge p = \texttt{Dist}(\sigma(\bar\theta))(v) > 0 \\
(\texttt{observe}(\varphi), \sigma) \Downarrow^{\epsilon} & (1, \sigma) & \text{if } \sigma(\varphi) = \texttt{true} \\
(\texttt{skip}, \sigma) \Downarrow^{\bar v_2} & (1, \sigma) & \\
& & \\
(\mathcal{S}_1; \mathcal{S}_2, \sigma) \Downarrow^{\bar v_1 \cdot \bar v_2} & (p_1 \times p_2, \sigma_2) & \text{if } (\mathcal{S}_1, \sigma) \Downarrow^{\bar v_1} (p_1, \sigma_1) \wedge (\mathcal{S}_2, \sigma_1) \Downarrow^{\bar v_2} (p_2, \sigma_2) \\
(\texttt{if}\, \mathcal{E}\, \texttt{then}\, \mathcal{S}_1\, \texttt{else}\, \mathcal{S}_2, \sigma) \Downarrow^{\bar v_1} & (p_1, \sigma_1) & \text{if } \sigma(\mathcal{E}) = \texttt{true} \wedge (\mathcal{S}_1, \sigma) \Downarrow^{\bar v_1} (p_1, \sigma_1) \\
(\texttt{if}\, \mathcal{E}\, \texttt{then}\, \mathcal{S}_1\, \texttt{else}\, \mathcal{S}_2, \sigma) \Downarrow^{\bar v_2} & (p_2, \sigma_2) & \text{if } \sigma(\mathcal{E}) = \texttt{false} \wedge (\mathcal{S}_2, \sigma) \Downarrow^{\bar v_2} (p_2, \sigma_2) \\
(\texttt{while}\, \mathcal{E}\, \texttt{do}\, \mathcal{S}, \sigma) \Downarrow^{\bar v_1} & (p_1, \sigma_1) & \text{if } \sigma(\mathcal{E}) = \texttt{true} \wedge (\mathcal{S}; \texttt{while}\, \mathcal{E}\, \texttt{do}\, \mathcal{S}, \sigma) \Downarrow^{\bar v_1} (p_1, \sigma_1) \\
(\texttt{while}\, \mathcal{E}\, \texttt{do}\, \mathcal{S}, \sigma) \Downarrow^{\bar v_2} & (1, \sigma) & \text{if } \sigma(\mathcal{E}) = \texttt{false}
\end{array}$$

The operational semantics of a program $\mathcal{S}$ gives rise to an unnormalized PDF $\mathbf{P}_{\mathcal{S}}$ on $\mathbb{S}$, which can be seen as a distribution on program executions because program executions are identified with a subset of $\mathbb{S}$.

$$\mathbf{P}_{\mathcal{S}}(\bar v) = \begin{cases} p & \text{if } (\mathcal{S}, \sigma_{\text{INIT}}) \Downarrow^{\bar v} (p, \sigma) \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

In the above equation σ_{INIT} denotes the initial state of the program, where all variables are assigned default values from the domain (booleans are assigned false, integers assigned zero, etc.). The normalized PDF $\widetilde{\mathbf{P}}_{\mathcal{S}}$ is defined as follows:

$$\widetilde{\mathbf{P}}_{\mathcal{S}}(\bar v) = \frac{\mathbf{P}_{\mathcal{S}}(\bar v)}{Z_{\mathcal{S}}} \quad \text{with} \quad Z_{\mathcal{S}} = \int_{\bar v \in \mathbb{S}} \mathbf{P}_{\mathcal{S}}(\bar v)\, d\bar v \tag{2}$$

The return expression of a program is a function $f : \Sigma \to \mathbb{R}$ from program states to reals. From the PDF on program executions of $\mathcal{S}$, we can calculate the expectation $\mathbf{E}_{\mathcal{S}}(f)$ as follows:

$$\mathbf{E}_{\mathcal{S}}(f) = \int_{\bar v \in \mathbb{S}} \mathbf{R}_{\mathcal{S}}(f, \bar v) \times \widetilde{\mathbf{P}}_{\mathcal{S}}(\bar v)\, d\bar v \tag{3}$$

where

$$\mathbf{R}_{\mathcal{S}}(f, \bar v) = \begin{cases} f(\sigma) & \text{if } (\mathcal{S}, \sigma_{\text{INIT}}) \Downarrow^{\bar v} (p, \sigma) \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

We define the semantics $[\![\mathcal{S}]\!]$ of a program $\mathcal{S}$ as its expectation function:

$$[\![\mathcal{S}]\!] = \mathbf{E}_{\mathcal{S}} \tag{5}$$

We say two programs $\mathcal{S}_1$ and $\mathcal{S}_2$ are equivalent when their semantics coincide:

$$\mathcal{S}_1 \equiv \mathcal{S}_2 \iff [\![\mathcal{S}_1]\!] = [\![\mathcal{S}_2]\!] \tag{6}$$

Suppose a program $\mathcal{S}$ is split into a set of straight-line programs $\{\mathcal{S}_i\}_{i \in I}$ (one for every valid path of $\mathcal{S}$). Then we have:

$$\{(\bar v, p, \sigma) \mid (\mathcal{S}, \sigma_{\text{INIT}}) \Downarrow^{\bar v} (p, \sigma)\} = \biguplus_{i \in I} \{(\bar v, p, \sigma) \mid (\mathcal{S}_i, \sigma_{\text{INIT}}) \Downarrow^{\bar v} (p, \sigma)\} \tag{7}$$

Thus, we have $\mathbf{P}_{\mathcal{S}}(\bar v) = \sum_{i \in I} \mathbf{P}_{\mathcal{S}_i}(\bar v)$ and $Z_{\mathcal{S}} = \sum_{i \in I} Z_{\mathcal{S}_i}$. So, the expectation of any function f for the program $\mathcal{S}$ can be rewritten as follows:

$$\begin{aligned}
\mathbf{E}_{\mathcal{S}}(f) &= \int_{\bar v \in \mathbb{S}} \mathbf{R}_{\mathcal{S}}(f, \bar v) \times \widetilde{\mathbf{P}}_{\mathcal{S}}(\bar v)\, d\bar v \;=\; \int_{\bar v \in \mathbb{S}} \mathbf{R}_{\mathcal{S}}(f, \bar v) \times \frac{\mathbf{P}_{\mathcal{S}}(\bar v)}{Z_{\mathcal{S}}}\, d\bar v \\
&= \frac{1}{Z_{\mathcal{S}}} \times \int_{\bar v \in \mathbb{S}} \mathbf{R}_{\mathcal{S}}(f, \bar v) \times \left(\textstyle\sum_{i \in I} \mathbf{P}_{\mathcal{S}_i}(\bar v)\right) d\bar v \;=\; \frac{1}{Z_{\mathcal{S}}} \times \sum_{i \in I} \int_{\bar v \in \mathbb{S}} \mathbf{R}_{\mathcal{S}}(f, \bar v) \times \mathbf{P}_{\mathcal{S}_i}(\bar v)\, d\bar v \\
&= \frac{1}{Z_{\mathcal{S}}} \times \sum_{i \in I} \int_{\bar v \in \mathbb{S}} \mathbf{R}_{\mathcal{S}_i}(f, \bar v) \times \mathbf{P}_{\mathcal{S}_i}(\bar v)\, d\bar v \qquad (\because \mathbf{R}_{\mathcal{S}_i}(f, \bar v) = \mathbf{R}_{\mathcal{S}}(f, \bar v) \text{ if } \mathbf{P}_{\mathcal{S}_i}(\bar v) \neq 0) \\
&= \frac{1}{Z_{\mathcal{S}}} \times \sum_{i \in I} \left(Z_{\mathcal{S}_i} \times \int_{\bar v \in \mathbb{S}} \mathbf{R}_{\mathcal{S}_i}(f, \bar v) \times \widetilde{\mathbf{P}}_{\mathcal{S}_i}(\bar v)\, d\bar v \right)
\end{aligned}$$

Therefore, we have

$$\mathbf{E}_{\mathcal{S}}(f) \quad = \quad \frac{\sum_{i \in I} Z_{\mathcal{S}_i} \times \mathbf{E}_{\mathcal{S}_i}(f)}{\sum_{i \in I} Z_{\mathcal{S}_i}} \tag{8}$$

We can transform a program P with while loops to a program P' without while loops by unrolling the loop a fixed number of times. If an exact bound can be determined for the maximum number of times a loop can execute, we can unroll that many times, and this transformation is semantics preserving. If not, P' approximates P (in the sense that the expectation of P' approximates the expectation of P), and the error between the expectation of P' and P can be made arbitrarily close by unrolling P enough number of times. In the remainder of the paper, we assume that such an unrolling has been done, and hence consider only loop-free programs.

3 The S^3 algorithm

Algorithm 1 describes the main procedure S^3 of our semantics sensitive sampler. This procedure takes a PROB program P and parameters κ_1, κ_2 as input and returns the expected value of the variable ret as output. Line 1 calls the procedure PRE (described in Section 3.1) that statically analyzes P and produces a program $\hat{P}$ that is semantically equivalent to P. The program $\hat{P}$ has the property that it samples from truncated distributions. Line 2 initializes a set Π that stores information about straight-line programs in $\hat{P}$. Every straight-line program in Π corresponds to a valid path of $\hat{P}$. Lines 3 – 6 iterate κ_1 number of times. In each iteration, $init(\hat{P}, P)$ runs the program $\hat{P}$ (line 4) and returns the resulting run in the form of a straight-line program Q that is a valid path in the original program P (which is accumulated into the set Π in line 5). For every run, $init$ executes every sample statement in $\hat{P}$ conditioned on the observe statement that immediately follows it. This ensures that every run of $\hat{P}$ is successful and results in a valid straight-line program. Next, for each straight-line program Q (lines 7 – 10), the PRE operation is performed and this results in semantically equivalent program $\hat{Q}$. Next, a Metropolis-Hastings sampler MH that operates on $\hat{Q}$ is called, and this results in a tuple $\Omega[i]$ (line 9, this is described in Section 3.2). Line 11 returns the expected value of ret for the program P, which is calculated according to Equation 8.

Algorithm 1 S$^3(P, \kappa_1, \kappa_2)$

Input: A PROB program P.
Output: The expected value of the variable ret returned by P.

```
1:  $\hat{P}, \_ :=$ PRE$(P, \texttt{true})$
2:  $\Pi := \emptyset$
3:  for $i = 1$ to $\kappa_1$ do
4:      $Q := init(\hat{P}, P)$
5:      $\Pi := \Pi \cup \{Q\}$
6:  end for
7:  for $Q \in \Pi$ do
8:      $\hat{Q}, \_ :=$ PRE$(Q, \texttt{true})$
9:      $\Omega[i] :=$ MH$(\hat{Q}, \kappa_2)$
10: end for
11: return $\frac{\sum_{1 \le i \le |\Pi|} fst(\Omega[i]) \times snd(\Omega[i])}{\sum_{1 \le i \le |\Pi|} fst(\Omega[i])}$
```

Algorithm 2 PRE$(\mathcal{S}, \varphi)$

Input: A predicate φ defined over program variables and a statement $\mathcal{S}$.
Output: A program $\hat{\mathcal{S}}$ that maps every sample statement with a pre-image predicate (via an observe statement immediately following the sample statement), and a pre-image predicate over $\mathcal{S}$.

```
switch ($\mathcal{S}$)
case $x = \mathcal{E}$:
    return $((x = \mathcal{E}), \varphi[\mathcal{E}/x])$
case $x \sim \texttt{Dist}(\bar{\theta})$:
    return $((x \sim \texttt{Dist}(\bar{\theta}); \texttt{observe}(\varphi)), \exists x.\varphi)$
case $\texttt{observe}\ (\psi)$:
    return $(\texttt{skip}, \varphi \wedge \psi)$
case $\texttt{skip}$:
    return $(\texttt{skip}, \varphi)$
case $\mathcal{S}_1; \mathcal{S}_2$:
    $(\mathcal{S}_2', \varphi') :=$ PRE$(\mathcal{S}_2, \varphi)$
    $(\mathcal{S}_1', \varphi'') :=$ PRE$(\mathcal{S}_1, \varphi')$
    return $((\mathcal{S}_1'; \mathcal{S}_2'), \varphi'')$
case $\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2$:
    $(\mathcal{S}_1', \varphi_1) :=$ PRE$(\mathcal{S}_1, \varphi)$
    $(\mathcal{S}_2', \varphi_2) :=$ PRE$(\mathcal{S}_2, \varphi)$
    return $((\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1'\ \texttt{else}\ \mathcal{S}_2'), (\mathcal{E} \wedge \varphi_1) \vee (\neg\mathcal{E} \wedge \varphi_2))$
case $\texttt{return}\ ret$:
    return $((\texttt{return}\ ret), \varphi)$
end switch
```

Theorem 1 *As $\kappa_1 \to \infty$, $\kappa_2 \to \infty$, S$^3(P, \kappa_1, \kappa_2)$ converges to the expected value of ret for the program P.*

Proof: This follows from Theorem 2, Theorem 3 and Equation 8. ■

3.1 Pre-image program analysis

Algorithm 2 describes the analysis where all the observe statements in the input program P are *hoisted* to all sample statements. The output of this phase is a new program $\hat{P}$, semantically equivalent to P, such that every sample statement in $\hat{P}$ is immediately followed by an observe statement with a corresponding pre-image predicate. A property of the pre-image predicate is that a program state σ satisfies it if and only if there exists an execution starting from σ satisfying all subsequent observations in the program. In other words, sampling from distributions truncated by their corresponding pre-image predicate will always produce acceptable samples and also all acceptable samples can be generated in such a way. The algorithm operates over the syntactic structure of the program in order to perform the pre-image analysis. In particular, the algorithm starts with the predicate `true` and uses the rules encoded in each of the case statements in Algorithm 2 to push the predicate backward all the way to the start of the program. For instance, as seen in lines 2 and 3, the pre-image 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 $\mathcal{E}$). As a result of the backward propagation of pre-image predicates, we have pre-image predicates associated with every sample statement via an observe statement (line 5). The following theorem proves the correctness of this transformation.

Theorem 2 *For any probabilistic program P with at least one successful execution, the program $\hat{P}$ with $(\hat{P}, \_) = \textsc{Pre}(P, \texttt{true})$ is semantically equivalent to P (denoted as $P \equiv \hat{P}$).*

Proof: Proof in the appendix of supplementary material. ■

It is important to note that the S^3 algorithm in Algorithm 1 invokes PRE twice, once in line 1, and also for every straight-line program in line 8. The first call is a whole program transformation which allows us to obtain valid executions (or straight-line programs) efficiently via the call to $init$ in line 4. The second call to PRE for every straight-line program propagates not only the observe statement, but also predicates due to conditionals and loops along the path. As a result, every sample statement is truncated or conditioned by a stronger pre-image predicate, so that every execution satisfies not only all observations, but also the conditions needed to execute the specific path.

3.2 The MH sampler

In this section, we describe a new Metropolis-Hastings algorithm for sampling from probabilistic straight-line programs. This is called in line 9 in Algorithm 1. The procedure MH (shown in Algorithm 3) takes a straight-line program P together with κ as inputs. The parameter κ represents the number of times we would like to run the program P (lines 2 – 26). The output of the algorithm is the expected value of ret for P. The procedure MH maintains two variables: (1) α which is useful for computing the probability of executing the straight-line program in the context of the original program, and (2) β which is used to decide whether the sample generated is to be accepted or rejected. Both α and β are initialized in line 3 together with a program state σ to be σ_{INIT}. This state gets updated as the program is executed. We also initialize α_{PREV} and Θ_{PREV} to be undefined values (denoted by $\perp$). Note that this initialization is reasonable as the samples generated from these undefined values can be included in the *burn-in* period of the MH routine. Lines 4 – 18 execute the program P one statement at a time. In particular, if the current statement is a sample statement "$(x \sim \texttt{Dist}(\bar{\theta}); \texttt{observe}(\varphi))$" (line 6), then a value v is sampled from a proposal distribution $\textsc{Prop}(\texttt{Dist}(\sigma(\bar{\theta})), \Theta_{\text{PREV}}[l](x)) \mid \{u \mid \sigma[x \leftarrow u](\varphi)\}$ (line 7), where $\sigma[x \leftarrow u]$ is the result of updating σ with the value of x set to u, and $\sigma(\bar{\theta})$ and $\sigma[x \leftarrow u](\varphi)$ are the values of the parameters $\bar{\theta}$ and the predicate φ evaluated with respect to the current program state σ and the updated program state $\sigma[x \leftarrow u]$. The expression $\Theta_{\text{PREV}}[l](x)$ represents the value of x at l during the previous run (*i.e.,* the previously sampled value). By DIST $\mid X$, we denote the truncated distrbution that results from restricting the domain of the distrbution DIST to the set X. Thus, the samples generated from the above truncated proposal distributions always satisfy the observe statement $\texttt{observe}(\varphi)$. Specifically, our implementation uses the following underlying proposal distributions:

Algorithm 3 $\textsc{Mh}(P, \kappa)$

Input: A straight-line program P, and κ, the number of samples to be generated.
Output: A tuple with the first component being the probability of executing P, and second component the expected value of the variable ret returned by P.

```
1: $\Omega := \emptyset$, $\alpha_{\text{PREV}} := \bot$, $\Theta_{\text{PREV}} := \bot$
2: **for** $i = 1$ to $\kappa$ **do**
3:     $\alpha := 1.0$, $\beta := 1.0$, $\sigma := \sigma_{\text{INIT}}$
4:     **for** $l = 1$ to $lines(P)$ **do**
5:         **switch** $(stmt(l))$
6:         **case** $(x \sim \texttt{Dist}(\bar{\theta}); \texttt{observe}(\varphi))$**:**
7:             $v \sim \textsc{Prop}(\texttt{Dist}(\sigma(\bar{\theta})), \Theta_{\text{PREV}}[l](x)) \mid \{u \mid \sigma[x \leftarrow u](\varphi)\}$
8:             $\sigma_{\text{PREV}} := \Theta_{\text{PREV}}[l-1]$, $v_{\text{PREV}} := \Theta_{\text{PREV}}[l](x)$
9:             $w := \int_{u \in \{u \mid \sigma[x \leftarrow u](\varphi)\}} \textsc{Density}(\texttt{Dist}(\sigma(\bar{\theta})))(u)\ du$
10:             $\alpha := \alpha \times w$
11:             $\beta := \beta \times \frac{\textsc{Density}(\texttt{Dist}(\sigma(\bar{\theta})))(v)\ \times\ \textsc{Density}(\textsc{Prop}(\texttt{Dist}(\sigma_{\text{PREV}}(\bar{\theta})), v) \mid \{u \mid \sigma_{\text{PREV}}[x \leftarrow u](\varphi)\})(v_{\text{PREV}})}{\textsc{Density}(\texttt{Dist}(\sigma_{\text{PREV}}(\bar{\theta})))(v_{\text{PREV}})\ \times\ \textsc{Density}(\textsc{Prop}(\texttt{Dist}(\sigma(\bar{\theta})), v_{\text{PREV}}) \mid \{u \mid \sigma[x \leftarrow u](\varphi)\})(v)}$
12:             $\sigma := \sigma[x \leftarrow v]$
13:             **break**
14:         **default:**
15:             $\sigma := eval(\sigma, l, P)$
16:         **end switch**
17:         $\Theta[l] := \sigma$
18:     **end for**
19:     **if** $i = 1 \lor \beta \geq 1 \lor \textsc{Bernoulli}(\beta)$ **then**
20:         $\Omega[i] := (\alpha, \sigma(ret))$
21:         $\Theta_{\text{PREV}} := \Theta$
22:         $\alpha_{\text{PREV}} := \alpha$
23:     **else**
24:         $\Omega[i] := (\alpha_{\text{PREV}}, \Theta_{\text{PREV}}[l](ret))$
25:     **end if**
26: **end for**
27: **return** $\left(\frac{\kappa}{\sum_{i=1}^{\kappa} 1/fst(\Omega[i])}, \frac{1}{\kappa}\sum_{i=1}^{\kappa} snd(\Omega[i])\right)$
```

- For any discrete distribution, the underlying proposal distribution is the same distribution, being independent of the previous sample v_{PREV}:

$$\textsc{Prop}(\texttt{Discrete}(\bar{\theta}), v_{\text{PREV}}) := \texttt{Discrete}(\bar{\theta})$$

- For any continuous distribution, the underlying proposal distribution is always a Gaussian distribution with mean v_{PREV} and a suitably chosen variance ν^2:

$$\textsc{Prop}(\texttt{Continuous}(\bar{\theta}), v_{\text{PREV}}) := \textsc{Gaussian}(v_{\text{PREV}}, \nu^2)$$

In line 9, w is assigned the probability of satisfying φ when the value of x is sampled from the non-truncated distribution $\texttt{Dist}(\sigma(\bar{\theta}))$, and this is accumulated across sample statements in the variable α (line 10).

In line 11, β is computed and is used to decide whether the sample is to be accepted or rejected (this is similar to the standard Metropolis-Hastings procedure). In line 12, the state σ is updated with the value of x set to v.

If line l is not a sample statement, then the state σ is updated by executing that statement (line 15). The map Θ is also updated to contain the state σ at index l. If $i = 1$ (first run) or β is greater than 1, then the sample $\sigma(ret)$ is accepted, otherwise it is accepted with probability β (lines 19 – 25).

Finally, in line 27, $\textsc{Mh}$ returns a tuple – the first component is the harmonic mean of the α values from each run, and the second component is the arithmetic mean over the return values from each run. Theorem 3 (below) states that the first component is the probability of executing the straight-line program P in the context of the whole program, and the second component is the expected value of ret with respect to the probability distribution defined by P.

Theorem 3 *Let $(\hat{P}, \_) = \textsc{Pre}(P, true)$ for a straight-line probabilistic program P. As $\kappa \to \infty$, $\textsc{Mh}(P, \kappa)$ computes the probability of executing P as well as the expected value of ret with respect to the distribution defined by P.*

Proof: Proof in the appendix of supplementary material. ■

NAME	DESCRIPTION	REFERENCE
Grass Model	Small model relating the probability of rain, having observed a wet lawn.	[4, 6].
Burglar Alarm	Estimate the probability of a burglary, having observed an alarm, earthquake	Adapted from Pearl
Noisy OR	Given a DAG, each node is a noisy-or of its parents. Find posterior marginal probability of a node, given observations.	[6]
Chess	Skill rating system for a Chess tournament consisting of 77 players and 2926 games.	[5]
Halo	Skill rating system for a tournament consisting of 31 teams, at most 4 players per team and 465 games played between teams.	Adapted from Figure 1 in [5]

Table 1: Benchmark programs.

NAME	ALGORITHM	SAMPLES	TIME TAKEN(s)
Burglar Alarm	S^3	90	0.012
	Qi	90	2.36
	S^3-SP	90	0.021
Noisy OR	S^3	1600	1.68
	Qi	3200	154.24
	S^3-SP	1600	17.68
Grass Model	S^3	1600	0.52
	Qi	2000	81.12
	S^3-SP	1600	349.36
Chess	S^3	100000	2910
	Qi	$\perp$	$\perp$
Halo	S^3	500000	3989.6
	Qi	$\perp$	$\perp$

Table 2: Evaluation results.

4 Empirical Evaluation

We evaluated S^3 on three popular benchmarks (Burglar Alarm, Noisy OR and Grass model) as well as two real world applications (Chess and Halo which are two different variations of the TrueSkill skill rating system in Xbox live [5]). We compared S^3 with the importance sampling algorithm Qi in [1] (note that this algorithm is already an improvement over the Church algorithm [4]). These benchmarks are described in Table 1 and the results are reported in Table 2. We have implemented S^3 in C++ and use the Z3 theorem prover [2] in order to represent and manage pre-image predicates.

The results in Table 2 show that S^3 significantly outperforms Qi when the objective is to obtain the same degree of precision with both algorithms – this difference is clearly visible for the Chess and Halo benchmarks where Qi does not produce an answer with precision comparable to S^3 even after 3 hours (denoted by $\perp$ in the table). Note that the **SAMPLES** column denotes the total number of samples generated by each algorithm to obtain answers with comparable precision. One of the reasons for this improvement is due to the fact that S^3 is able to improvise based on its current state in order to move into regions of high density. On the other hand, Qi is agnostic to previous samples, and therefore can produce samples with very low quality or weight. It is interesting to note that for the Burglar alarm benchmark, S^3 is faster than Qi even when both algorithms generate the same number of samples. This is due to extra computation performed by Qi to compute weighted average (over importance weights and samples) to compute the expectation. On the other hand, S^3 computes the arithmetic mean for expectations (second component of the tuple in line 27 of Algorithm 3).

In order to make a comparison with structural position based variable indexing [15], we implemented such an indexing for S^3. We denote these results with the label S^3-SP. For all benchmarks, structural position based indexing is more expensive than the PC (or line number) based indexing implemented in S^3. In the case of the Grass model benchmark, S^3 with strucural-position based indexing is also more expensive than Qi. We note that the PC based indexing is possible in S^3 because the first phase of S^3 splits a program into paths.

5 Conclusion

We have presented a new MCMC algorithm S^3 for efficiently sampling from probabilistic programs. Our algorithm splits the program into straghtline paths, computes estimates separately for these paths, and combines them to produce an estimate for the program. A unique feature of our algorithm is that it uses ideas from program analysis such as pre-image computation in order to avoid rejection due to conditioning in the program.

We have formalized the semantics of probabilistic programs and rigorously proved the correctness of $\mathtt{S}^3$. Our experimental results are also encouraging —we show that $\mathtt{S}^3$ significantly outperforms the importance sampling algorithm QI on a set of benchmarks and two real world applications.

As future work, we would like to extend Algorithm 3 to Hamiltonian Monte Carlo sampling [11]. Note that this would entail replacing line 7 in Algorithm 3 with leapfrog computation steps. In addition, we would like to parallelize $\mathtt{S}^3$. Though our current implementation of $\mathtt{S}^3$ is sequential, our algorithm is massively parallelizable: each path can be sampled independently, and within each pth, we can independently perform sampling in parallel. Using Equation 8, we can appropriately combine the results from such parallel computations to produce a result for the whole program. An implementation of this approach is also interesting future work.

References

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

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

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

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

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

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

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

[8] 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.

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

[10] T. Minka, J. Winn, J. Guiver, and A. Kannan. Infer.NET 2.3, 2009.

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

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

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

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

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

A Proof of Theorem 2

In order to prove the theorem, we first prove that the following lemma holds.

Lemma 1 *For any S, φ and $(\hat{\mathcal{S}}, \hat{\varphi}) = \mathrm{Pre}(\mathcal{S}, \varphi)$, we have:* $\forall \sigma_0, \bar{v}, p, \sigma.$

$$(\sigma(\varphi) = \texttt{true} \wedge (\mathcal{S}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)) \iff (\sigma_0(\hat{\varphi}) = \texttt{true} \wedge (\hat{\mathcal{S}}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma))$$

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

- When $\mathcal{S}$ is `skip`: we have $\hat{\mathcal{S}} = \mathcal{S}$ and $\hat{\varphi} = \varphi$.
  The lemma holds trivially.
- When $\mathcal{S}$ is $x = \mathcal{E}$: we have $\hat{\mathcal{S}} = \mathcal{S}$ and $\hat{\varphi} = \varphi[\mathcal{E}/x]$.

$\Rightarrow$: We have $\sigma_0[x \leftarrow \sigma_0(\mathcal{E})] = \sigma$ by definition of $\Downarrow$ and thus we have $\sigma_0[\hat{\varphi}] = \sigma_0[\varphi[\mathcal{E}/x]] = \sigma_0[x \leftarrow \sigma_0(\mathcal{E})](\varphi) = \sigma(\varphi) = \texttt{true}$.

$\Leftarrow$: Similarly, we have $\sigma(\varphi) = \sigma_0(\hat{\varphi}) = \texttt{true}$.

- 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 $\hat{\varphi} = \exists x.\, \varphi$.

$\Rightarrow$: We have $\sigma_0[x \leftarrow \bar{v}_{(1)}] = \sigma$ by definition of $\Downarrow$. Thus $\sigma_0(\hat{\varphi}) = \sigma_0(\exists x.\, \varphi) = \texttt{true}$ follows from $\sigma_0[x \leftarrow \bar{v}_{(1)}](\varphi) = \sigma(\varphi) = \texttt{true}$. Also, due to the assumption $\sigma(\varphi) = \texttt{true}$, we have $(\hat{\mathcal{S}}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$.

$\Leftarrow$: We have $\sigma(\varphi) = \texttt{true}$ due to $\texttt{observe}(\varphi)$ statement. $(\mathcal{S}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$ holds trivally.

- When $\mathcal{S}$ is $\texttt{observe}(\psi)$: we have $\hat{\mathcal{S}} = \texttt{skip}$ and $\hat{\varphi} = \varphi \wedge \psi$.

$\Rightarrow$: We have $\sigma(\varphi) = \texttt{true}$ by assumption and $\sigma = \sigma_0$ and $\sigma(\psi) = \texttt{true}$ by definition of $\Downarrow$. Thus we have $\sigma_0(\hat{\varphi}) = \sigma(\varphi \wedge \psi) = \texttt{true}$. And $(\hat{\mathcal{S}}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$ holds trivally.

$\Leftarrow$: We have $\sigma_0(\hat{\varphi}) = \texttt{true}$ by assumption and $\sigma = \sigma_0$ by definition of $\Downarrow$. Since $\sigma_0(\hat{\varphi}) = \sigma(\varphi \wedge \psi) = \texttt{true}$, we have $\sigma(\varphi) = \texttt{true}$ and $\sigma(\psi) = \texttt{true}$, from which $(\mathcal{S}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$ follows.

- 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 $(\hat{\mathcal{S}}_1, \hat{\varphi}) = \mathrm{Pre}(\mathcal{S}_1, \varphi')$.

$\Rightarrow$: From $(\mathcal{S}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$ we have $\bar{v}_1, \bar{v}_2, p_1, p_2, \sigma'$ such that $(\mathcal{S}_1, \sigma_0) \Downarrow^{\bar{v}_1} (p_1, \sigma') \wedge (\mathcal{S}_2, \sigma') \Downarrow^{\bar{v}_2} (p_2, \sigma) \wedge \bar{v} = \bar{v}_1 \cdot \bar{v}_2 \wedge p = p_1 \times p_2$. By the induction hypothesis we have $\sigma'(\varphi') = \texttt{true}$ and $(\hat{\mathcal{S}}_2, \sigma') \Downarrow^{\bar{v}_2} (p_2, \sigma)$, again by the induction hypothesis we have $\sigma_0(\hat{\varphi}) = \texttt{true}$ and $(\hat{\mathcal{S}}_1, \sigma_0) \Downarrow^{\bar{v}_1} (p_1, \sigma')$. Thus $(\hat{\mathcal{S}}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$ holds by definition.

$\Leftarrow$: From $(\hat{\mathcal{S}}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$ we have $\bar{v}_1, \bar{v}_2, p_1, p_2, \sigma'$ such that $(\hat{\mathcal{S}}_1, \sigma_0) \Downarrow^{\bar{v}_1} (p_1, \sigma') \wedge (\hat{\mathcal{S}}_2, \sigma') \Downarrow^{\bar{v}_2} (p_2, \sigma) \wedge \bar{v} = \bar{v}_1 \cdot \bar{v}_2 \wedge p = p_1 \times p_2$. By the induction hypothesis we have $\sigma'(\varphi') = \texttt{true}$ and $(\mathcal{S}_1, \sigma_0) \Downarrow^{\bar{v}_1} (p_1, \sigma')$, again by the induction hypothesis we have $\sigma(\varphi) = \texttt{true}$ and $(\mathcal{S}_2, \sigma') \Downarrow^{\bar{v}_2} (p_2, \sigma)$. Thus $(\mathcal{S}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$ holds by definition.

- When $\mathcal{S}$ is $\texttt{if}\,\mathcal{E}\,\texttt{then}\,\mathcal{S}_1\,\texttt{else}\,\mathcal{S}_2$: we have $\hat{\mathcal{S}} = \texttt{if}\,\mathcal{E}\,\texttt{then}\,\hat{\mathcal{S}}_1\,\texttt{else}\,\hat{\mathcal{S}}_2$ 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 $(\hat{\mathcal{S}}_2, \varphi_2) = \mathrm{Pre}(\mathcal{S}_2, \varphi)$.

$\Rightarrow$: From $(\mathcal{S}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$ we have $(\sigma_0(\mathcal{E}) = \texttt{true} \wedge (\mathcal{S}_1, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma))$ or $(\sigma_0(\mathcal{E}) = \texttt{false} \wedge (\mathcal{S}_2, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma))$. In the former case, by the induction hypothesis, we have $\sigma_0(\varphi_1) = \texttt{true}$ and $(\hat{\mathcal{S}}_1, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$. Thus we have $\sigma_0(\hat{\varphi}) = (\sigma_0(\mathcal{E}) \wedge \sigma_0(\varphi_1)) \vee (\neg\sigma_0(\mathcal{E}) \wedge \sigma_0(\varphi_2)) = \texttt{true}$ and $(\hat{\mathcal{S}}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$. In the latter case, by the induction hypothesis, we have $\sigma_0(\varphi_2) = \texttt{true}$ and $(\hat{\mathcal{S}}_2, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$. Thus we have $\sigma_0(\hat{\varphi}) = (\sigma_0(\mathcal{E}) \wedge \sigma_0(\varphi_1)) \vee (\neg\sigma_0(\mathcal{E}) \wedge \sigma_0(\varphi_2)) = \texttt{true}$ and $(\hat{\mathcal{S}}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$.

$\Leftarrow$: From $(\hat{\mathcal{S}}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$ we have $(\sigma_0(\mathcal{E}) = \texttt{true} \wedge (\hat{\mathcal{S}}_1, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma))$ or $(\sigma_0(\mathcal{E}) = \texttt{false} \wedge (\hat{\mathcal{S}}_2, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma))$. In the former case, $\sigma_0(\varphi_1) = \texttt{true}$ follows from $\sigma_0(\hat{\varphi}) = \texttt{true}$ and thus by the induction hypothesis, we have $\sigma(\varphi) = \texttt{true}$ and $(\mathcal{S}_1, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$, from which $(\mathcal{S}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$ follows. In the latter case, $\sigma_0(\varphi_2) = \texttt{true}$ follows from $\sigma_0(\hat{\varphi}) = \texttt{true}$ and by the induction hypothesis, we have $\sigma(\varphi) = \texttt{true}$ and $(\mathcal{S}_2, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$, from which $(\mathcal{S}, \sigma_0) \Downarrow^{\bar{v}} (p, \sigma)$ follows.

■

Now we prove Theorem 2. Suppose $(\hat{P}, \psi) = \textsc{Pre}(P, \mathtt{true})$. From the fact that P has a successful execution (*i.e.,* $(P, \sigma_{\text{INIT}}) \Downarrow^{\bar{v}} (p, \sigma)$ for some $\bar{v}$, p, σ), we have $\sigma_{\text{INIT}}(\psi) = \mathtt{true}$ by Lemma 1 since $\sigma(\mathtt{true}) = \mathtt{true}$. Since $\sigma_{\text{INIT}}(\psi) = \mathtt{true}$ and $\sigma(\mathtt{true}) = \mathtt{true}$ for any σ, again by Lemma 1 we have, for any $\bar{v}$, p, σ:

$$(P, \sigma_{\text{INIT}}) \Downarrow^{\bar{v}} (p, \sigma) \iff (\hat{P}, \sigma_{\text{INIT}}) \Downarrow^{\bar{v}} (p, \sigma) .$$

Thus we have $[\![P]\!] = [\![\hat{P}]\!]$ by definition of $[\![-]\!]$.

B Proof of Theorem 3

Let $(\hat{P}, \_) = \textsc{Pre}(P, \mathtt{true})$ for a straight-line probabilistic program P. Without loss of generality, suppose that $\hat{P}$ has the following n sampling with observe statements:

$$(x_1 \sim \mathtt{Dist}_{\mathtt{1}}(\bar{\theta}_1); \mathtt{observe}(\varphi_1)), \ldots, (x_n \sim \mathtt{Dist}_{\mathtt{n}}(\bar{\theta}_n); \mathtt{observe}(\varphi_n)).$$

During the execution of $(\hat{P}, \sigma_{\text{INIT}})$, the program state at the first sampling is uniquely determined (say $\sigma_1 \in \Sigma$), and the state at the second sampling is dependent on the value generated at the first sampling (say $\sigma_2 \in (\mathbb{N} \uplus \mathbb{R}) \to \Sigma$), and so on (say $\ldots, \sigma_n \in (\mathbb{N} \uplus \mathbb{R})^{n-1} \to \Sigma$). Then we simply write ψ_i and D_i for $i = 1, \ldots, n$ as follows:

$$\begin{aligned}
\psi_1 &:= \{u \mid \sigma_1[x \leftarrow u](\varphi_1)\} \\
\psi_2(v_1) &:= \{u \mid \sigma_2(v_1)[x \leftarrow u](\varphi_2)\} \\
&\;\vdots \\
\psi_n(v_1, \ldots, v_{n-1}) &:= \{u \mid \sigma_n(v_1, \ldots, v_{n-1})[x \leftarrow u](\varphi_n)\} \\
D_1(v_1) &:= \textsc{Density}(\mathtt{Dist}_{\mathtt{1}}(\sigma_1(\bar{\theta}_1)))(v_1) \\
D_2(v_2 \mid v_1) &:= \textsc{Density}(\mathtt{Dist}_{\mathtt{2}}(\sigma_2(v_1)(\bar{\theta}_2)))(v_2) \\
&\;\vdots \\
D_n(v_n \mid v_1, \ldots, v_{n-1}) &:= \textsc{Density}(\mathtt{Dist}_{\mathtt{n}}(\sigma_n(v_1, \ldots, v_{n-1})(\bar{\theta}_n)))(v_n)
\end{aligned}$$

We further write ψ and D as follows:

$$\begin{aligned}
\psi &:= \{(v_1, \ldots, v_n) \mid v_1 \in \psi_1 \wedge v_2 \in \psi_2(v_1) \wedge \ldots \wedge v_n \in \psi_n(v_1, \ldots, v_{n-1})\} \\
D(v_1, \ldots, v_n) &:= D_1(v_1) \times D_2(v_2 \mid v_1) \times \ldots \times D_n(v_n \mid v_1, \ldots, v_{n-1})
\end{aligned}$$

Then one can easily see that for any execution $(\hat{P}, \sigma_{\text{INIT}}) \Downarrow^{\bar{v}} (p, \sigma)$ we have $\bar{v} \in \psi$ and $p = D(\bar{v})$. Thus the unnormalized PDF $\mathbf{P}_{\mathcal{S}}$ on $(\mathbb{N} \uplus \mathbb{R})^n \subseteq \mathbb{S}$ can be rewritten as follows

$$\mathbf{P}_{\mathcal{S}}(\bar{v}) = \begin{cases} D(\bar{v}) & \text{if } \bar{v} \in \psi \\ 0 & \text{otherwise} \end{cases}$$

Then we consider the Metropolis-Hastings simulation using the proposal distribution Q defined as follows:

$$\begin{aligned}
Q_1(u_1 \to v_1) &:= \textsc{Density}(\textsc{Prop}(\mathtt{Dist}_1(\sigma_1(\bar{\theta}_1)), u_1) \mid \psi_1)(v_1) \\
Q_2(u_2 \to v_2 \mid v_1) &:= \textsc{Density}(\textsc{Prop}(\mathtt{Dist}_2(\sigma_2(v_1)(\bar{\theta}_2)), u_2) \mid \psi_2(v_1))(v_2) \\
&\;\vdots \\
Q_n(u_n \to v_n \mid v_1, \ldots, v_{n-1}) &:= \textsc{Density}(\textsc{Prop}(\mathtt{Dist}_n(\sigma_n(v_1, \ldots, v_{n-1})(\bar{\theta}_n)), u_n) \mid \psi_n(v_1, \ldots, v_{n-1}))(v_n) \\
Q(u_1, \ldots, u_n \to v_1, \ldots, v_n) &:= Q_1(u_1 \to v_1) \times Q_2(u_2 \to v_2 \mid v_1) \times \ldots \times Q_n(u_n \to v_n \mid v_1, \ldots, v_{n-1})
\end{aligned}$$

By our definition of $\textsc{Prop}$, one can easily see that for any $\bar{u}, \bar{v}$ such that $\mathbf{P}_{\mathcal{S}}(\bar{u}) > 0$ and $\mathbf{P}_{\mathcal{S}}(\bar{v}) > 0$, we have $Q(\bar{u} \to \bar{v}) > 0$. Thus the proposal distribution Q is valid.

To generate the next sample $\bar{v}$ from the current sample $\bar{u}$ with $\mathbf{P}_{\mathcal{S}}(\bar{u}) > 0$ according to Q, one can generate each component v_i one by one as follows:

$$\begin{aligned}
v_1 &\sim Q_1(u_1 \to -) \\
v_2 &\sim Q_2(u_2 \to - \mid v_1) \\
&\;\vdots \\
v_n &\sim Q_n(u_n \to - \mid v_1, \ldots, v_{n-1})
\end{aligned}$$

And the acceptance rate β is calculated as follows:

$$\begin{aligned}\beta &= \frac{\mathbf{P}_{\mathcal{S}}(\bar{v}) \times Q(\bar{v} \to \bar{u})}{\mathbf{P}_{\mathcal{S}}(\bar{u}) \times Q(\bar{u} \to \bar{v})} = \frac{D(\bar{v}) \times Q(\bar{v} \to \bar{u})}{D(\bar{u}) \times Q(\bar{u} \to \bar{v})} \quad (\because \mathbf{P}_{\mathcal{S}}(\bar{u}) > 0 \land \bar{v} \in \psi) \\ &= \frac{D_1(v_1) \times Q_1(v_1 \to u_1)}{D_1(u_1) \times Q_1(u_1 \to v_1)} \times \frac{D_2(v_2 \mid v_1) \times Q_2(v_2 \to u_2 \mid u_1)}{D_2(u_2 \mid u_1) \times Q_2(u_2 \to v_2 \mid v_1)} \times \ldots \times \\ &\quad\ \frac{D_n(v_n \mid v_1, \ldots, v_{n-1}) \times Q_n(v_n \to u_n \mid u_1, \ldots, u_{n-1})}{D_n(u_n \mid u_1, \ldots, u_{n-1}) \times Q_n(u_n \to v_n \mid v_1, \ldots, v_{n-1})}\end{aligned}$$

Algorithm 3 generates samples $(\bar{v}_1, \ldots, \bar{v}_\kappa)$ this way and returns the mean of the value of ret:

$$\frac{\mathbf{R}_{\hat{P}}(f_{ret}, \bar{v}_1) + \ldots + \mathbf{R}_{\hat{P}}(f_{ret}, \bar{v}_\kappa)}{\kappa} \quad \text{for } f_{ret}(\sigma) = \sigma(ret).$$

Thus by correctness of Metropolis-Hastings simulation we have

$$\mathbf{E}_{\hat{P}}(f_{ret}) = \int_{\bar{v} \in \mathbb{S}} \mathbf{R}_{\hat{P}}(f_{ret}, \bar{v}) \times \widetilde{\mathbf{P}}_{\hat{P}}(\bar{v})\, d\bar{v} = \lim_{\kappa \to \infty} \frac{\mathbf{R}_{\hat{P}}(f_{ret}, \bar{v}_1) + \ldots + \mathbf{R}_{\hat{P}}(f_{ret}, \bar{v}_\kappa)}{\kappa}$$

This proves that the second component of the tuple returned by Algorithm 3 is the expected value of ret with respect to the distribution defined by $\hat{P}$.

Now we show that Algorithm 3 correctly computes the probability $Z_{\hat{P}}$ of executing $\hat{P}$. First, we define α as follows (note that this is exactly the same as α at the end of a run of $\hat{P}$ in Algorithm 3):

$$\begin{aligned}\alpha(v_1, \ldots, v_n) &= \left(\int_{y_1 \in \psi_1} D_1(y_1)\, dy_1\right) \times \left(\int_{y_2 \in \psi_2(v_1)} D_2(y_2 \mid v_1)\, dy_2\right) \times \ldots \\ &\quad \times \left(\int_{y_n \in \psi_n(v_1, \ldots, v_{n-1})} D_n(y_n \mid v_1, \ldots, v_{n-1})\, dy_n\right)\end{aligned} \tag{9}$$

Then, for samples $(\bar{v}_1, \ldots, \bar{v}_\kappa)$ generated by Algorithm 3, by correctness of Metropolis-Hastings simulation we have the following:

$$\lim_{\kappa \to \infty} \frac{\frac{1}{\alpha(\bar{v}_1)} + \ldots + \frac{1}{\alpha(\bar{v}_\kappa)}}{\kappa} = \int_{\bar{x} \in \mathbb{S}} \frac{1}{\alpha(\bar{x})} \times \widetilde{\mathbf{P}}_{\hat{P}}(\bar{x})\, d\bar{x} \tag{10}$$

We can simplify the RHS as follows.

$$\begin{aligned}&\int_{\bar{x} \in \mathbb{S}} \frac{1}{\alpha(\bar{x})} \times \widetilde{\mathbf{P}}_{\hat{P}}(\bar{x})\, d\bar{x} \;=\; \int_{\bar{x} \in \psi} \frac{1}{\alpha(\bar{x})} \times \frac{D(\bar{x})}{Z_{\hat{P}}}\, d\bar{x} \;=\; \frac{1}{Z_{\hat{P}}} \times \int_{\bar{x} \in \psi} \frac{D(\bar{x})}{\alpha(\bar{x})}\, d\bar{x} \\ &= \frac{1}{Z_{\hat{P}}} \times \left(\int_{x_1 \in \psi_1} \frac{D_1(x_1)}{\int_{y_1 \in \psi_1} D_1(y_1)\, dy_1} \times \left(\int_{x_2 \in \psi_2(x_1)} \frac{D_2(x_2 \mid x_1)}{\int_{y_2 \in \psi_2(x_1)} D_2(y_2 \mid x_1)\, dy_2} \times \ldots \right.\right. \\ &\qquad \left.\left. \times \left(\int_{x_n \in \psi_n(x_1, \ldots, x_{n-1})} \frac{D_n(x_n \mid x_1, \ldots, x_{n-1})}{\int_{y_n \in \psi_n(x_1, \ldots, x_{n-1})} D_n(y_n \mid x_1, \ldots, x_{n-1})\, dy_n}\, dx_n\right) \ldots dx_2\right) dx_1\right) \\ &= \frac{1}{Z_{\hat{P}}} \times \left(\int_{x_1 \in \psi_1} \frac{D_1(x_1)}{\int_{y_1 \in \psi_1} D_1(y_1)\, dy_1} \times \left(\int_{x_2 \in \psi_2(x_1)} \frac{D_2(x_2 \mid x_1)}{\int_{y_2 \in \psi_2(x_1)} D_2(y_2 \mid x_1)\, dy_2} \times \ldots \right.\right. \\ &\qquad \left.\left. \times \left(\quad 1 \quad\right) \ldots dx_2\right) dx_1\right) \\ &\ \vdots \\ &= \frac{1}{Z_{\hat{P}}} \times 1\end{aligned}$$

Therefore, we have

$$Z_{\hat{P}} = \frac{1}{\int_{\bar{x} \in \mathbb{S}} \frac{1}{\alpha(\bar{x})} \times \widetilde{\mathbf{P}}_{\hat{P}}(\bar{x})\, d\bar{x}} = \frac{1}{\lim_{\kappa \to \infty} \frac{\frac{1}{\alpha(\bar{v}_1)} + \ldots + \frac{1}{\alpha(\bar{v}_\kappa)}}{\kappa}} = \lim_{\kappa \to \infty} \frac{\kappa}{\frac{1}{\alpha(\bar{v}_1)} + \ldots + \frac{1}{\alpha(\bar{v}_\kappa)}}$$

This proves that the first component of the tuple returned by Algorithm 3 is the probability $Z_{\hat{P}}$ of executing $\hat{P}$.