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 $S^3$ tackles these problems using ideas from program analysis. First, $S^3$ 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, $S^3$ 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 $S^3$. We also empirically demonstrate the effectiveness of $S^3$, 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 \cite{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 (repre-
senting 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 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 S^3 which is presented in Section 3. In Section 4 we empirically demonstrate the effectiveness of S^3 over a set of benchmarks. In particular, we compare S^3 with the Qt algorithm [1] – our empirical results show that S^3 outperforms Qt 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
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 “\(x \sim \text{Dist}(\theta)\)” draws a sample from a distribution \(\text{Dist}\) with a vector of parameters \(\theta\), and assigns it to the variable \(x\). For instance, “\(x \sim \text{Gaussian}(\mu, \sigma^2)\)” draws a value from a Gaussian distribution with mean \(\mu\) and variance \(\sigma^2\), and assigns it to the variable \(x\).

2. The observe statement “\(\text{observe}(\varphi)\)” conditions a distribution with respect to a predicate or condition \(\varphi\) 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 \(\sigma\) of a program is a valuation to all its variables. The set of all states (which can be infinite) is denoted by \(\Sigma\). The probabilistic semantics \((S, \sigma) \mid\triangleright^p (p, \sigma')\) denotes that the outcome \(\sigma'\) of the program \(S\) with initial state \(\sigma\) is determined by a sequence of samples \(\bar{v}\) with probability density \(p\). The sequence of samples \(\bar{v}\) belongs to the measure space \(S\) defined by: \(S = \bigcup_{n \geq 0} (\mathbb{N} \cup \mathbb{R})^n\).
The operational semantics of PROB is completely specified using the following rules:

\[ (x = \epsilon, \sigma) \Downarrow \left(1, \sigma[x \leftarrow \sigma(\epsilon)]\right) \]
\[ (x \sim \text{Dist}(\theta), \sigma) \Downarrow \left(p, \sigma[x \leftarrow \cdot] \right) \text{ if } v \in \mathbb{N} \oplus \mathbb{R} \land p = \text{Dist}(\bar{\theta})(v) > 0 \]
\[ (\text{observe}(\varphi), \sigma) \Downarrow \left(1, \sigma \right) \text{ if } \sigma(\varphi) = \text{true} \]
\[ (\text{skip}, \sigma) \Downarrow \left(1, \sigma \right) \]
\[ (S_1; S_2, \sigma) \Downarrow (S_1 \times S_2, \sigma) \]
\[ (\text{if } \ell \text{ then } S_1 \text{ else } S_2, \sigma) \Downarrow 1 \]
\[ (\text{while } \ell \text{ do } S, \sigma) \Downarrow \left\{ \begin{array}{ll}
(p_1, \sigma) & \text{ if } \sigma(\ell) = \text{true} \\
(p_2, \sigma) & \text{ if } \sigma(\ell) = \text{false} 
\end{array} \right. \]

The operational semantics of a program \( S \) gives rise to an unnormalized PDF \( P_S \) on \( S \), which can be seen as a distribution on program executions because program executions are identified with a subset of \( S \).

\[
P_S(v) = \begin{cases} 
  p & \text{if } (S, \sigma_{\text{INIT}}) \Downarrow (p, \sigma) \\
  0 & \text{otherwise}
\end{cases} \quad (1)
\]

In the above equation \( \sigma_{\text{INIT}} \) 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 \( \tilde{P}_S \) is defined as follows:

\[
\tilde{P}_S(v) = \frac{P_S(v)}{Z_S} \quad \text{with} \quad Z_S = \int_{v \in S} P_S(v) \, dv \quad (2)
\]

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 \( S \), we can calculate the expectation \( E_S(f) \) as follows:

\[
E_S(f) = \int_{v \in S} R_S(f, \bar{v}) \times \tilde{P}_S(\bar{v}) \, d\bar{v} \quad (3)
\]

where

\[
R_S(f, \bar{v}) = \begin{cases} 
  f(\sigma) & \text{if } (S, \sigma_{\text{INIT}}) \Downarrow (p, \sigma) \\
  0 & \text{otherwise}
\end{cases} \quad (4)
\]

We define the semantics \([S]\) of a program \( S \) as its expectation function:

\[
[S] = E_S \quad (5)
\]

We say two programs \( S_1 \) and \( S_2 \) are equivalent when their semantics coincide:

\[
S_1 \equiv S_2 \iff [S_1] = [S_2] \quad (6)
\]

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

\[
\{(\bar{v}, p, \sigma) \mid (S, \sigma_{\text{INIT}}) \Downarrow (p, \sigma)\} = \bigcup_{i \in I} \{(\bar{v}, p, \sigma) \mid (S_i, \sigma_{\text{INIT}}) \Downarrow (p, \sigma)\} \quad (7)
\]

Thus, we have \( P_S(\bar{v}) = \sum_{i \in I} P_{S_i}(\bar{v}) \) and \( Z_S = \sum_{i \in I} Z_{S_i} \). So, the expectation of any function \( f \) for the program \( S \) can be rewritten as follows:

\[
E_S(f) = \int_{\bar{v} \in S} R_S(f, \bar{v}) \times \tilde{P}_S(\bar{v}) \, d\bar{v} = \int_{\bar{v} \in S} R_S(f, \bar{v}) \times \frac{P_S(\bar{v})}{Z_S} \, d\bar{v} = \frac{1}{Z_S} \int_{\bar{v} \in S} R_S(f, \bar{v}) \times \sum_{i \in I} P_{S_i}(\bar{v}) \, d\bar{v} = \frac{1}{Z_S} \sum_{i \in I} \int_{\bar{v} \in S} R_S(f, \bar{v}) \times P_{S_i}(\bar{v}) \, d\bar{v} \quad (\because R_S(f, \bar{v}) = R_S(f, \bar{v}) \text{ if } P_{S_i}(\bar{v}) \neq 0)
\]

\[
= \frac{1}{Z_S} \times \sum_{i \in I} \left( Z_{S_i} \times \int_{\bar{v} \in S} R_S(f, \bar{v}) \times \tilde{P}_{S_i}(\bar{v}) \, d\bar{v} \right)
\]
Therefore, we have

\[ E_S(f) = \frac{\sum_{i \in I} Z_{S_i} \times E_{S_i}(f)}{\sum_{i \in I} Z_{S_i}} \]  

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 \( \text{PRE} \) describes the main procedure \( S^3 \) of our semantics sensitive sampler. This procedure takes a \texttt{PROB} program \( P \) and parameters \( \kappa_1, \kappa_2 \) as input and returns the expected value of the variable \( \text{ret} \) as output. Line 1 calls the procedure \( \text{PRE} \) (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 \( \Pi \) that stores information about straight-line programs in \( \hat{P} \). Every straight-line program in \( \Pi \) corresponds to a valid path of \( \hat{P} \).

Lines 3 – 6 iterate \( \kappa_1 \) number of times. In each iteration, \( \text{init}(\hat{P}, P) \) 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 \( \Pi \) in line 5). For every run, \( \text{init} \) executes every sample statement in \( P \) 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 \( \text{PRE} \) operation is performed and this results in a semantically equivalent program \( \hat{Q} \). Next, a Metropolis-Hastings sampler \( \text{MH} \) 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 \( \text{ret} \) for the program \( \hat{P} \), which is calculated according to Equation 8.

```
Algorithm 2 \text{PRE}(S, \varphi)

Input: A predicate \( \varphi \) defined over program variables and a statement \( S \).

Output: A program \( \hat{S} \) 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 \( S \).

1: switch (\( S \))
2: case \( x = \epsilon \):
3: \quad return ((x = \epsilon), \varphi[\epsilon/x])
4: case \( x \sim \text{Dist}(\theta); \psi \):
5: \quad return ((x \sim \text{Dist}(\theta); \text{observe}(\varphi)), \exists x. \varphi)
6: case \text{observe}(\psi):
7: \quad return (\text{skip}, \varphi \land \psi)
8: case \text{skip}:
9: \quad return (\text{skip}, \varphi)
10: case \text{S}_1; \text{S}_2:
11: \quad (\text{S}'_1, \varphi') := \text{PRE}(\text{S}_1, \varphi)
12: \quad (\text{S}'_2, \varphi'') := \text{MH}(\text{S}_2, \varphi)
13: \quad return ((\text{S}'_1; \text{S}'_2), \varphi'')
14: case if \( E \) then \( \text{S}_1 \) else \( \text{S}_2 \):
15: \quad (\text{S}'_1, \varphi_1) := \text{PRE}(\text{S}_1, \varphi)
16: \quad (\text{S}'_2, \varphi_2) := \text{PRE}(\text{S}_2, \varphi)
17: \quad return (((\text{S}'_1; \text{S}'_2), (E \land \varphi_1)) \lor (\neg E \land \varphi_2))
18: case return:\n19: \quad return ((\text{return ret}), \varphi)
20: 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 \( \text{ret} \) for the program \( P \).
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 \( \sigma \) satisfies it if and only if there exists an execution starting from \( \sigma \) 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 \( \text{true} \) 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 \( \varphi \) with respect to an assignment statement \( x = E \) is defined to be \( \varphi[E/x] \) (this denotes the predicate obtained by replacing all occurrence of \( x \) in \( \varphi \) with \( E \)). 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 \( (\hat{P}, \cdot) = \text{PRE}(P, \text{true}) \) 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 \( \text{PRE} \) 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 \( \text{init} \) in line 4. The second call to \( \text{PRE} \) 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 \( \text{MH} \) (shown in Algorithm 3) takes a straight-line program \( P \) together with \( \kappa \) as inputs. The parameter \( \kappa \) 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 \( \text{RET} \) for \( P \). The procedure \( \text{MH} \) maintains two variables: (1) \( \alpha \) which is useful for computing the probability of executing the straight-line program in the context of the original program, and (2) \( \beta \) which is used to decide whether the sample generated is to be accepted or rejected. Both \( \alpha \) and \( \beta \) are initialized in line 3 together with a program state \( \sigma \) to be \( \sigma_{\text{INIT}} \). This state gets updated as the program is executed. We also initialize \( \alpha_{\text{PREV}} \) and \( \Theta_{\text{PREV}} \) 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 \( \text{“}(x \sim \text{Dist}(\bar{\theta}); \text{observe}(\varphi))\text{”} \) (line 6), then a value \( v \) is sampled from a proposal distribution \( \text{PROP}(\text{Dist}(\sigma(\bar{\theta})), \Theta_{\text{PREV}}[l](x)) \mid \{u \mid \sigma[x \leftarrow u](\varphi)\} \) (line 7), where \( \sigma[x \leftarrow u] \) is the result of updating \( \sigma \) 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 \( \varphi \) evaluated with respect to the current program state \( \sigma \) 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 \( \text{Dist} \mid X \), we denote the truncated distribution that results from restricting the domain of the distribution \( \text{Dist} \) to the set \( X \). Thus, the samples generated from the above truncated proposal distributions always satisfy the observe statement \( \text{observe}(\varphi) \). Specifically, our implementation uses the following underlying proposal distributions:
Algorithm 3 Mh($P, \kappa$)

Input: A straight-line program $P$, and $\kappa$, 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 $\text{ret}$ returned by $P$.

1: $\Omega \leftarrow \emptyset$, $\sigma_{\text{PREV}} \leftarrow \bot$, $\Theta_{\text{PREV}} \leftarrow \bot$
2: for $i = 1$ to $\kappa$ do
3: $\alpha = 1.0$, $\beta = 1.0$, $\sigma = \sigma_{\text{OUT}}$
4: for $i = 1$ to lines($P$) do
5: switch($\text{stat}(i)$)
6: case $(x \sim \text{Dist}(\varphi); \text{observe}(\varphi))$:
7: $v \sim \text{PROB}(\text{Dist}(\sigma(\varphi)), \Theta_{\text{PREV}} \mid l(x)) \mid \{u : \sigma[x \leftarrow u](\varphi)\}$
8: $\sigma_{\text{PREV}} = \Theta_{\text{PREV}}[\beta - 1], \Theta_{\text{PREV}} = \Theta_{\text{PREV}}[\beta][x]
9: $\omega := \int_{u \in \varphi} \text{DENSITY}(\text{Dist}(\sigma(\varphi)))(u) \ du$
10: $\alpha = \alpha \times \omega$
11: $\beta := \beta \times \text{DENSITY}(\text{Dist}(\sigma(\varphi)))(\alpha \times \text{DENSITY}(\text{PROB}(\text{Dist}(\sigma(\varphi)), \sigma_{\text{PREV}} \mid l(x) \mid \{u : \sigma[x \leftarrow u](\varphi)\}) (\nu_{\text{PREV}}))$
12: $\sigma := \sigma[x \leftarrow v]
13: \text{break}
14: default:
15: $\sigma = \text{eval}(\sigma, t, P)$
16: end switch
17: $\Theta[l] := \sigma$
18: end for
19: if $i = 1 \lor \beta \geq 1 \lor \text{BERNOULLI}(\beta)$ then
20: $\Omega[i] := (\alpha, \sigma(\text{ret}))$
21: $\Theta_{\text{PREV}} := \Theta$
22: $\sigma_{\text{PREV}} := \alpha$
23: else
24: $\Omega[i] := (\alpha_{\text{PREV}}, \Theta_{\text{PREV}}[l](\text{ret}))$
25: end if
26: end for
27: return $\left(\sum_{i=1}^{n} \Omega[i], \frac{1}{n} \sum_{i=1}^{n} \text{snd}(\Omega[i])\right)$

- For any discrete distribution, the underlying proposal distribution is the same distribution, being independent of the previous sample $\nu_{\text{PREV}}$:
  \[
  \text{PROP}(\text{Discrete}(\varphi), \nu_{\text{PREV}}) := \text{Discrete}(\varphi)
  \]
- For any continuous distribution, the underlying proposal distribution is always a Gaussian distribution with mean $\nu_{\text{PREV}}$ and a suitably chosen variance $\nu^2$:
  \[
  \text{PROP}(\text{Continuous}(\varphi), \nu_{\text{PREV}}) := \text{GAUSSIAN}(\nu_{\text{PREV}}, \nu^2)
  \]

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

In line 11, $\beta$ 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 $\sigma$ is updated with the value of $x$ set to $v$.

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

Finally, in line 27, Mh returns a tuple – the first component is the harmonic mean of the $\alpha$ 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 $\text{ret}$ with respect to the probability distribution defined by $P$.

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

Proof: Proof in the appendix of supplementary material.
Table 1: Benchmark programs.

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

Table 2: Evaluation results.

<table>
<thead>
<tr>
<th>NAME</th>
<th>ALGORITHM</th>
<th>SAMPLES</th>
<th>TIME TAKEN(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Burglar Alarm</td>
<td>$S^3$</td>
<td>90</td>
<td>0.012</td>
</tr>
<tr>
<td></td>
<td>Qt</td>
<td>90</td>
<td>2.36</td>
</tr>
<tr>
<td></td>
<td>$S^3$-SP</td>
<td>90</td>
<td>0.021</td>
</tr>
<tr>
<td>Noisy OR</td>
<td>$S^3$</td>
<td>1600</td>
<td>1.68</td>
</tr>
<tr>
<td></td>
<td>Qt</td>
<td>3200</td>
<td>154.34</td>
</tr>
<tr>
<td></td>
<td>$S^3$-SP</td>
<td>1600</td>
<td>17.68</td>
</tr>
<tr>
<td>Grass Model</td>
<td>$S^3$</td>
<td>1600</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td>Qt</td>
<td>2000</td>
<td>81.12</td>
</tr>
<tr>
<td></td>
<td>$S^3$-SP</td>
<td>1600</td>
<td>349.36</td>
</tr>
<tr>
<td>Chess</td>
<td>$S^3$</td>
<td>100000</td>
<td>2910</td>
</tr>
<tr>
<td></td>
<td>Qt</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>Halo</td>
<td>$S^3$</td>
<td>500000</td>
<td>3989.6</td>
</tr>
<tr>
<td></td>
<td>Qt</td>
<td>⊥</td>
<td>⊥</td>
</tr>
</tbody>
</table>

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 Qt 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 Qt 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 Qt does not produce an answer with precision comparable to $S^3$ even after 3 hours (denoted by ⊥ 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, Qt 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 Qt even when both algorithms generate the same number of samples. This is due to extra computation performed by Qt 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 structural-position based indexing is also more expensive than Qt. 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 straightline 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 S3. Our experimental results are also encouraging—we show that S3 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 S3. Though our current implementation of S3 is sequential, our algorithm is massively parallelizable: each path can be sampled independently, and within each path, 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


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 (S, ϕ) = Pre(S, φ), we have: ∀σ, v, p, σ.

\[(σ(φ) = \text{true} \land (S, σ_0) \Downarrow (p, σ)) \iff (σ(ϕ) = \text{true} \land (S, σ_0) \Downarrow (p, σ))\]

Proof: We prove the lemma by induction on the structure of S.

- When S is skip: we have S = S and ϕ = φ.
  The lemma holds trivially.

- When S is x = E: we have S = S and ϕ = φ | E/x.
  \[⇒: \text{We have } σ(x ← σ(E)) = σ \text{ by definition of } \Downarrow \text{ and thus we have } σ(ϕ) = σ(ϕ | E/x) = σ_0(x ← σ(E))(σ) = σ(ϕ) = \text{true}.\]
  \[⇐: \text{Similarly, we have } σ(φ) = σ(ϕ) = \text{true}.\]

- When S is x ∼ Dist(β): we have S = (x ∼ Dist(β); observe(φ)) and ϕ = ∃x. φ.
  \[⇒: \text{We have } σ(x ← v(1)) = σ \text{ by definition of } \Downarrow \text{. Thus } σ(ϕ) = σ(∃x. φ) = \text{true} \text{ follows from } σ(x ← v(1))(φ) = σ(φ) = \text{false}. \text{ Also, due to the assumption } σ(φ) = \text{true, we have } (S, σ_0) \Downarrow (p, σ) \text{ holds.} \]
  \[⇐: \text{We have } σ(φ) = \text{true due to observe(φ) statement. } (S, σ_0) \Downarrow (p, σ) \text{ holds trivially.}\]

- When S is observe(ψ): we have S = skip and ϕ = φ ∧ ψ.
  \[⇒: \text{We have } σ(φ) = \text{true by assumption and } σ = σ_0 \text{ and } σ(ψ) = \text{true by definition of } \Downarrow \text{. Thus we have } σ(ϕ) = σ(φ ∧ ψ) = \text{true}. \text{ And } (S, σ_0) \Downarrow (p, σ) \text{ holds trivially. }\]

- When S is S1; S2: we have S = S1; S2 with (S2, φ') = Pre(S2, φ) and (S1, φ) = Pre(S1, φ').
  \[⇒: \text{From } (S, σ_0) \Downarrow (p, σ) \text{ we have } σ_1, σ_2, p, σ' \text{ such that } (S_1, σ_0) \Downarrow (p_1, σ' ∧ (S_2, σ') \Downarrow (p_2, σ ∧ (σ_1, σ_0) \Downarrow (p_1, σ')) \text{ and } (S_2, σ') \Downarrow (p_2, σ). \text{ By the induction hypothesis we have } σ'(φ') = \text{true and } (S_2, σ') \Downarrow (p_2, σ) \text{, again by the induction hypothesis we have } σ(ϕ) = \text{true and } (S_1, σ_0) \Downarrow (p_1, σ') \text{ holds by definition.}\]

- When S is if E then S1 else S2: we have S = if E then S1 else S2 and ϕ = (E ∧ φ_1) ∨ (¬E ∧ φ_2) with (S1, φ_1) = Pre(S1, φ) and (S2, φ_2) = Pre(S2, φ).
  \[⇒: \text{From } (S, σ_0) \Downarrow (p, σ) \text{ we have } σ(ϕ) = \text{true ∧ } (S_1, σ_0) \Downarrow (p, σ) \text{ or } σ(ϕ) = \text{false ∧ } (S_2, σ_0) \Downarrow (p, σ) \text{. In the former case, by the induction hypothesis, we have } σ(ϕ) = \text{true and } (S_1, σ_0) \Downarrow (p, σ). \text{ Thus we have } σ(ϕ) = (σ(ϕ) ∧ σ(φ_1)) ∨ (¬σ(ϕ) ∧ σ(φ_2)) = \text{true and } (S_2, σ_0) \Downarrow (p, σ). \text{ In the latter case, by the induction hypothesis, we have } σ(ϕ) = \text{true and } (S_1, σ_0) \Downarrow (p, σ). \text{ Thus we have } σ(ϕ) = (σ(ϕ) ∧ σ(φ_1)) ∨ (¬σ(ϕ) ∧ σ(φ_2)) = \text{true and } (S_2, σ_0) \Downarrow (p, σ).\]

- When S is for E do p: we have S = for E do p and ϕ = (E ∧ φ_1) ∨ (¬E ∧ φ_2) with (S1, φ_1) = Pre(S1, φ) and (S2, φ_2) = Pre(S2, φ).
  \[⇒: \text{From } (S, σ_0) \Downarrow (p, σ) \text{ we have } σ(ϕ) = \text{true ∧ } (S_1, σ_0) \Downarrow (p, σ) \text{ or } σ(ϕ) = \text{false ∧ } (S_2, σ_0) \Downarrow (p, σ). \text{ In the former case, } σ(φ_1) = \text{true follows from } σ(ϕ) = \text{true and thus by the induction hypothesis, we have } σ(φ) = \text{true and } (S_1, σ_0) \Downarrow (p, σ), \text{ from which } (S, σ_0) \Downarrow (p, σ) \text{ follows. In the latter case, } σ(φ_2) = \text{true follows from } σ(ϕ) = \text{true and by the induction hypothesis, we have } σ(φ) = \text{true and } (S_2, σ_0) \Downarrow (p, σ), \text{ from which } (S, σ_0) \Downarrow (p, σ) \text{ follows.}\]
Now we prove Theorem 2. Suppose \((\hat{P}, \psi) = \text{PRE}(P, \text{true})\). From the fact that \(P\) has a successful execution (i.e., \((P, \sigma_{\text{INIT}}) \downarrow^\gamma (p, \sigma)\) for some \(\hat{v}, p, \sigma\), we have \(\sigma_{\text{INIT}}(\psi) = \text{true}\) by Lemma 1 since \(\sigma(\text{true}) = \text{true}\). Since \(\sigma_{\text{INIT}}(\psi) = \text{true}\) and \(\sigma(\text{true}) = \text{true}\) for any \(\sigma\), again by Lemma 1 we have, for any \(\hat{v}, p, \sigma\):

\[
(P, \sigma_{\text{INIT}}) \downarrow^\gamma (p, \sigma) \iff (\hat{P}, \sigma_{\text{INIT}}) \downarrow^\gamma (p, \sigma).
\]

Thus we have \([P] = [\hat{P}]\) by definition of \([\cdot]\).

### B Proof of Theorem 3

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

\[
(x_1 \sim \text{Dist}_1(\bar{\theta}_1); \text{observe}(\varphi_1)), \ldots, (x_n \sim \text{Dist}_n(\bar{\theta}_n); \text{observe}(\varphi_n)).
\]

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 \(\sigma_2 \in (\mathbb{N} \cup \mathbb{R}) \rightarrow \Sigma\)), and so on (say \(\sigma_n \in (\mathbb{N} \cup \mathbb{R})^{n-1} \rightarrow \Sigma\)). Then we simply write \(\psi_i\) and \(D_i\) for \(i = 1, \ldots, n\) as follows:

\[
\psi_1 := \{u \mid \sigma_1[x \leftarrow u](\varphi_1)\}
\]

\[
\psi_2(v_1) := \{u \mid \sigma_2(v_1)[x \leftarrow u](\varphi_2)\}
\]

\[
\vdots
\]

\[
\psi_n(v_1, \ldots, v_{n-1}) := \{u \mid \sigma_n(v_1, \ldots, v_{n-1})[x \leftarrow u](\varphi_n)\}
\]

\[
D_1(v_1) := \text{DENSITY}(\text{Dist}_1(\sigma_1(\bar{\theta}_1)))(v_1)
\]

\[
D_2(v_2 | v_1) := \text{DENSITY}(\text{Dist}_2(\sigma_2(\bar{\theta}_2)))(v_2)
\]

\[
\vdots
\]

\[
D_n(v_n | v_1, \ldots, v_{n-1}) := \text{DENSITY}(\text{Dist}_n(\sigma_n(v_1, \ldots, v_{n-1}))(\bar{\theta}_n))(v_n)
\]

We further write \(\psi\) and \(D\) as follows:

\[
\psi := \{(v_1, \ldots, v_n) \mid v_1 \in \psi_1 \land v_2 \in \psi_2(v_1) \land \ldots \land v_n \in \psi_n(v_1, \ldots, v_{n-1})\}
\]

\[
D(v_1, \ldots, v_n) := D_1(v_1) \times D_2(v_2 | v_1) \times \ldots \times D_n(v_n | v_1, \ldots, v_{n-1})
\]

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

\[
P_S(\bar{v}) = \begin{cases} 
D(\bar{v}) & \text{if } \bar{v} \in \psi \\
0 & \text{otherwise}
\end{cases}
\]

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

\[
Q_1(u_1 \rightarrow v_1) := \text{DENSITY}(\text{PROP}(\text{Dist}_1(\sigma_1(\bar{\theta}_1)), u_1) | \psi_1)(v_1)
\]

\[
Q_2(u_2 \rightarrow v_2 | v_1) := \text{DENSITY}(\text{PROP}(\text{Dist}_2(\sigma_2(\bar{\theta}_2)), u_2) | \psi_2(v_1))(v_2)
\]

\[
\vdots
\]

\[
Q_n(u_n \rightarrow v_n | v_1, \ldots, v_{n-1}) := \text{DENSITY}(\text{PROP}(\text{Dist}_n(\sigma_n(v_1, \ldots, v_{n-1})(\bar{\theta}_n)), u_n) | \psi_n(v_1, \ldots, v_{n-1}))(v_n)
\]

\[
Q(u_1, \ldots, u_n \rightarrow v_1, \ldots, v_n) := Q_1(u_1 \rightarrow v_1) \times Q_2(u_2 \rightarrow v_2 | v_1) \times \ldots \times Q_n(u_n \rightarrow v_n | v_1, \ldots, v_{n-1})
\]

By our definition of \(\text{PROP}\), one can easily see that for any \(\hat{u}, \hat{v}\) such that \(P_S(\hat{u}) > 0\) and \(P_S(\hat{v}) > 0\), we have \(Q(\hat{u} \rightarrow \hat{v}) > 0\). Thus the proposal distribution \(Q\) is valid.

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

\[
v_1 \sim Q_1(u_1 \rightarrow -)
\]

\[
v_2 \sim Q_2(u_2 \rightarrow - | v_1)
\]

\[
\vdots
\]

\[
v_n \sim Q_n(u_n \rightarrow - | v_1, \ldots, v_{n-1})
\]
Therefore, we have
\[ \alpha(v_1, \ldots, v_n) = \left( \int_{y_1 \in \psi_1} D_1(y_1) \, dy_1 \right) \times \left( \int_{y_2 \in \psi_2(v_1)} D_2(y_2 | v_1) \, dy_2 \right) \times \ldots \times \left( \int_{y_n \in \psi_n(v_1, \ldots, v_{n-1})} D_n(y_n | v_1, \ldots, v_{n-1}) \, dy_n \right) \] (9)

Then, for samples \((\tilde{v}_1, \ldots, \tilde{v}_\kappa)\) generated by Algorithm 3, by correctness of Metropolis-Hastings simulation we have the following:
\[ \lim_{\kappa \to \infty} \frac{1}{\alpha(v_1)} + \ldots + \frac{1}{\alpha(v_n)} = \int_{\bar{x} \in \psi} \frac{1}{Z_P} \times P_\beta(\bar{x}) \, d\bar{x} \] (10)

We can simplify the RHS as follows.
\[ \int_{\bar{x} \in \psi} \frac{1}{Z_P} \times P_\beta(\bar{x}) \, d\bar{x} = \int_{\bar{x} \in \psi} \frac{1}{Z_P} \times D(\bar{x}) \, d\bar{x} = \frac{1}{Z_P} \times \int_{\bar{x} \in \psi} D(\bar{x}) \, d\bar{x} \]
\[ = \frac{1}{Z_P} \times \left( \int_{\bar{x}_1 \in \psi_1} \frac{D_1(\bar{x}_1)}{\alpha(\bar{x})} \, d\bar{x}_1 \right) \times \left( \int_{\bar{x}_2 \in \psi_2(\bar{x}_1)} \frac{D_2(\bar{x}_2 | \bar{x}_1)}{\alpha(\bar{x})} \, d\bar{x}_2 \right) \times \ldots \times \left( \int_{\bar{x}_n \in \psi_n(\bar{x}_1, \ldots, \bar{x}_{n-1})} \frac{D_n(\bar{x}_n | \bar{x}_1, \ldots, \bar{x}_{n-1})}{\alpha(\bar{x})} \, d\bar{x}_n \right) \, d\bar{x}_1 \]
\[ = \frac{1}{Z_P} \times \left( \int_{\bar{x}_1 \in \psi_1} \frac{D_1(\bar{x}_1)}{\alpha(\bar{x})} \, d\bar{x}_1 \right) \times \left( \int_{\bar{x}_2 \in \psi_2(\bar{x}_1)} \frac{D_2(\bar{x}_2 | \bar{x}_1)}{\alpha(\bar{x})} \, d\bar{x}_2 \right) \times \ldots \times \left( \frac{1}{\alpha(\bar{x})} \right) \, d\bar{x}_1 \]
\[ = \frac{1}{Z_P} \times 1 \]

Therefore, we have
\[ Z_P = \lim_{\kappa \to \infty} \frac{1}{\alpha(v_1)} + \ldots + \frac{1}{\alpha(v_n)} = \lim_{\kappa \to \infty} \frac{1}{\alpha(v_1)} + \ldots + \frac{1}{\alpha(v_n)} \]

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