# Particle Gibbs with Ancestor Sampling for Probabilistic Programs

**Jan-Willem van de Meent**
Dept of Statistics
Columbia University

**Hongseok Yang**
Dept of Computer Science
University of Oxford

**Vikash Mansinghka**
Computer Science & AI Lab
Mass Institute of Technology

**Frank Wood**
Dept of Engineering
University of Oxford

## Abstract

Particle Markov chain Monte Carlo techniques rank among current state-of-the-art methods for probabilistic program inference. A drawback of these techniques is that they rely on importance resampling, which results in degenerate particle trajectories and a low effective sample size for variables sampled early in a program. We here develop a formalism to adapt ancestor resampling, a technique that mitigates particle degeneracy, to the probabilistic programming setting. We present empirical results that demonstrate nontrivial performance gains.

## 1 Introduction

Probabilistic programming languages extend traditional languages with primitives for sampling and conditioning on random variables. Running a program $F$ generates random variates $\boldsymbol{x}$ for some subset of program expressions, which can be thought of as a sample from a prior $p(\boldsymbol{x} \mid F)$ implicitly defined by the program. In a conditioned program $F[\boldsymbol{y}]$, a subset of expressions is constrained to take on observed values $\boldsymbol{y}$. This defines a posterior distribution $p(\boldsymbol{x} \mid F[\boldsymbol{y}])$ on the random variates $\boldsymbol{x}$ that can be generated by the program $F[\boldsymbol{y}]$.

Probabilistic programs are in essence procedural representations of generative models. These representations are often both succinct and extendable, making it easy to iterate over alternative model designs. Any program that always samples and constrains a fixed set of variables $\{\boldsymbol{x}, \boldsymbol{y}\}$ admits an alternate representation as a graphical model. In general, a program $F[\boldsymbol{y}]$ may also have random variables that are only generated when certain conditions on previously sampled variates are met. The random variables $\boldsymbol{x}$ therefore need not have the same entries for all possible executions of $F[\boldsymbol{y}]$. In languages that have recursion and higher-order functions (i.e. functions that act on other functions), it is straightforward to define models that can instantiate an arbitrary number of random variables, such as certain Bayesian nonparametrics, or models that are specified in terms of a generative grammar. At the same time this greater expressivity makes it challenging to design methods for efficient posterior inference in arbitrary programs.

In this paper we show how a recently proposed technique known as particle Gibbs with ancestor sampling (PGAS) [Lindsten et al., 2012] can be adapted to inference in higher-order probabilistic programming systems [Mansinghka et al., 2014, Wood et al., 2014, Goodman et al., 2008]. A PGAS implementation requires combining a partial execution history, or prefix, with the remainder of a previously completed execution history, which we call a suffix. We develop a formalism for performing this operation in a manner that guarantees a consistent program state, correctly updates the probabilities associated with each sampled and observed random variable, and avoids unnecessary recomputation where possible. An empirical evaluation demonstrates that the increased statistical efficiency of PGAS can easily outweigh its greater computational cost.

### 1.1 Related Work

Current generation probabilistic programming systems fall into two broad categories. On the one hand, systems like Infer.NET [Minka et al., 2010] and STAN [Stan Development Team, 2014] restrict the language syntax in order to omit recursion. This ensures that the set of variables is bounded and has a well-defined dependency hierarchy. On the other hand languages such as Church [Goodman et al., 2008], Venture [Mansinghka et al., 2014], Anglican [Wood et al., 2014], and Probabilistic C [Paige and Wood, 2014] do not impose such restrictions, which makes the design of general-purpose inference methods more difficult.

Appearing in Proceedings of the $18^{th}$ International Conference on Artificial Intelligence and Statistics (AISTATS) 2015, San Diego, CA, USA. JMLR: W&CP volume 38.

Arguably the simplest inference methods for probabilistic programming languages rely on Sequential Monte Carlo (SMC) [Del Moral et al., 2006]. These "forward" techniques sample from the prior by running multiple copies of the program, calculating importance weights using the conditioning expressions as needed. The only non-trivial requirement for implementing SMC variants is that there exists an efficient mechanism for "forking" a program state into multiple copies that may continue execution independently.

Another well-known inference technique for probabilistic programs is lightweight Metropolis-Hasting (LMH) [Wingate et al., 2011]. LMH methods construct a database of sampled values at run time. A change to the sampled values is proposed, and the program is re-run in its entirety, substituting previously sampled values where possible. The newly constructed database of random variables is then either accepted or rejected. LMH is straightforward to implement, but costly, since the program must be re-run in its entirety to evaluate the acceptance ratio. A more computationally efficient strategy is offered by Venture [Mansinghka et al., 2014], which represents the execution history of a program as a graph, in which each evaluated expression is a node. A graph walk can then determine the subset of expressions affected by a proposed change, allowing partial re-execution of the program conditioned on all unaffected nodes.

SMC and MH based algorithms each have trade-offs. Techniques that derive from SMC can be run very efficiently, but suffer from particle degeneracy, resulting in a deteriorating quality of posterior estimates calculated from values sampled early in the program. In MH methods subsequent samples are typically correlated, and many updates may be needed to obtain an independent sample. As we will discuss in Section 3, PGAS can be thought of as a hybrid technique, in the sense that SMC is used to generate independent updates to the previous sample, which mitigates the degeneracy issues associated with SMC whilst increasing mixing rates relative to MH.

## 2 Probabilistic Functional Programs

### 2.1 Language Syntax

For the purposes of exposition we will consider a simple Lisp dialect, extended with primitives for sampling and observing random variables

```
e ::= c | s | (e &e) | (lambda (&s) e)
      | (if e e e) | (quote e)
v ::= bool | int | float | string | primitive
      | compound | (&v) | stochastic
```

An expression `e` is either a constant literal `c`, a symbol `s`, an application `(e &e)` with operator `e` and zero or more arguments `&e`, a function literal `(lambda (&s) e)` with argument list `(&s)`, an if-statement `(if e e e)`, or a quoted expression `(quote e)`. Each expression `e` evaluates to a value `v` upon execution. In addition to the self-explanatory `bool`, `int`, `float`, and `string` types, values can be `primitive` procedures (i.e. language built-ins such `+`, `-`, etc.), `compound` procedures (i.e., closures), and lists of zero or more values `(&v)`.

The `stochastic` type represents stochastic processes, whose samples must either be i.i.d. or exchangeable. A stochastic process `sp` supports two operations

```
(sample sp) -> v
(observe sp v) -> v
```

The `sample` primitive draws a value from `sp`, whereas the `observe` primitive conditions execution on a sample `v` that is returned as passed. Both operations associate a value `v` to `sp` as a side-effect, which changes the probability of the execution state in the inference procedure. For exchangeable processes such as `(crp 1.0)` this also affects the probability of future samples.

Following the convention employed by Venture and Anglican, we define programs as sequences of three types of top-level statements

```
F ::= t F | END
t ::= [assume s e] | [observe e v] | [predict e]
```

Variables in the global environment are defined using the `assume` directive. The `observe` directive conditions execution in the same way as its non-toplevel equivalent. The `predict` directive returns the value of the expression `e` as inference output.

### 2.2 Importance Sampling Semantics

In many probabilistic languages the `(observe e v)` form is semantically equivalent to imposing a rejection-sampling criterion. For example, a program subject to `(observe (> a 0) true)` can be interpreted as a rejection sampler that repeatedly runs the program and only returns `predict` values when `(> a 0)` holds.

The semantics of `observe` that we have defined here imply an interpretation of a probabilistic program as an importance sampler, where `sample` draws from the prior and `observe` assigns an importance weight. Instead of constraining the value of an arbitrary expression `e`, `observe` conditions on the value of `(sample e)`. This restricted form of conditioning guarantees that we can calculate the likelihood $p(\mathtt{v} \mid (\mathtt{sample\ e}))$ for every `observe`, as long as we implement a density function for all possible `stochastic` values in the language.

More formally, we use the notation $F[\boldsymbol{y}]$ to refer to a program conditioned on values $\boldsymbol{y}$ via a sequence of top-level `[observe e v]` statements. Execution of $F[\boldsymbol{y}]$ will require the evaluation of a number of `(sample e)` expressions, whose values we will denote with $\boldsymbol{x}$. We now informally define $F[\boldsymbol{y}, \boldsymbol{x}]$ as the program in which all `(sample e)` expressions are replaced by conditioned equivalents `(observe e v)`, resulting in a fully deterministic execution. Similarly we can define $F[\boldsymbol{x}]$ as the program obtained from $F[\boldsymbol{y}, \boldsymbol{x}]$ by replacing `[observe e v]` statements with unconditioned forms `[assume s (sample e)]` with a unique symbol `s` in each statement.

Whereas top-level `observe` statements are fixed in number and order, $F$ may not evaluate the same combination of `sample` calls in every execution. To provide a more precise definition of $F[\boldsymbol{x}]$, we associate a unique address $\alpha$ with each `sample` call that can occur in the execution of $F$. We here use a scheme in which the run-time address of each evaluation is a concatenation $\alpha'::(t, p)$ of the address $\alpha'$ of the parent evaluation and a tuple $(t, p)$ in which $t$ identifies the expression type and $p$ is the index of the sub-expression within the form. This particular scheme labels every evaluation, not just the sample calls, allowing us to formally represent $F : \mathcal{A} \to \mathcal{E}$ as a mapping from addresses $\mathcal{A}$ to program expressions $\mathcal{E}$. We represent $\boldsymbol{x} : \mathcal{S} \to \mathcal{V}$ as a mapping from a subset of addresses $\mathcal{S} \subset \mathcal{A}$ associated with `sample` calls to values $\mathcal{V}$. Similarly, $\boldsymbol{y} : \mathcal{O} \to \mathcal{V}$ is a mapping from addresses $\mathcal{O} \subset \mathcal{A}$ associated with `observe` statements to values. With these definitions in place, a conditioned program $F[\boldsymbol{x}] : \mathcal{A} \to \mathcal{E}$ simply replaces $F(\alpha) =$ `(sample e)` with

$$F[\boldsymbol{x}](\alpha) = \texttt{(observe e } \boldsymbol{x}(\alpha)\texttt{)}, \qquad \forall \alpha \in \mathcal{S}.$$

The importance sampling interpretation of a program $F[\boldsymbol{y}] \rightsquigarrow W, \boldsymbol{x}$ (read as $F[\boldsymbol{x}]$ yields $W, \boldsymbol{x}$) is defined in terms of random variables $\boldsymbol{x}$ and a weight $W$

$$F[\boldsymbol{y}] \rightsquigarrow W, \boldsymbol{x} \qquad\qquad W = p(\boldsymbol{y} \,|\, F[\boldsymbol{x}]) \;.$$

By definition, the generated samples $\boldsymbol{x}$ are drawn from the prior $p(\boldsymbol{x} \,|\, F)$. The weight $W = p(\boldsymbol{y} \,|\, F[\boldsymbol{x}])$ is the joint probability of all top-level `observe` statements in $F[\boldsymbol{y}]$. Repeated execution of $F[\boldsymbol{y}]$ yields a weighted sample set $\{W^l, \boldsymbol{x}^l\}$ that may be used to approximate the posterior as

$$p(\boldsymbol{x} \,|\, F[\boldsymbol{y}]) \simeq p^L(\boldsymbol{x} \,|\, F[\boldsymbol{y}]) = \sum_l \frac{W^l}{\sum_k W^k} \delta_{\boldsymbol{x}^l}.$$

More generally the importance weight is defined as the joint probability of all `observe` calls (top-level and transformed) in the program

$$\begin{aligned} F[\boldsymbol{y}, \boldsymbol{x}] &\rightsquigarrow W && W = p(\boldsymbol{y}, \boldsymbol{x} \,|\, F), \\ F[\boldsymbol{x}] &\rightsquigarrow W, \boldsymbol{y} && W = p(\boldsymbol{x} \,|\, F[\boldsymbol{y}]), \\ F &\rightsquigarrow W, \boldsymbol{y}, \boldsymbol{x} && W = 1. \end{aligned}$$

# 3 Particle MCMC methods

## 3.1 Sequential Monte Carlo

SMC methods are importance sampling techniques that target a posterior $p(\boldsymbol{x} \,|\, \boldsymbol{y})$ on as space $\mathcal{X}$ by performing importance sampling on unnormalized densities $\{\gamma_n(\boldsymbol{x}_n)\}_{n=1}^N$ defined on spaces of expanding dimensionality $\{\mathcal{X}_n\}_{n=1}^N$, where each $\mathcal{X}_n \subseteq \mathcal{X}_{n+1}$ and $\mathcal{X}_N = \mathcal{X}$. This results in a series of intermediate particle sets $\{w_n^l, \boldsymbol{x}_n^l\}_{n=1}^N$, which we refer to as generations. Each generation is sampled via two steps,

$$a_n^l \sim R(a_n \,|\, w_n), \qquad \boldsymbol{x}_n^l \sim \rho_n(\boldsymbol{x}_n \,|\, \boldsymbol{x}_{n-1}^{a_n^l}).$$

Here $R(a \,|\, w)$ is a resampling procedure that returns index $a = l$ with probability $w^l / \sum_{l'} w^{l'}$ and $\rho_n$ is a transition kernel. The samples $\boldsymbol{x}_n^l$ are assigned weights

$$w_n^l = \frac{\gamma_n(\boldsymbol{x}_n^l)}{\gamma_{n-1}(\boldsymbol{x}_{n-1}^{a_n^l}) \rho_n(\boldsymbol{x}_n^l \,|\, \boldsymbol{x}_{n-1}^{a_n^l})} \;.$$

In the context of probabilistic programs, we can define a series of partial programs $F_n[\boldsymbol{y}_n]$ that truncate at each top-level `[observe e v]` statement. We can then sequentially sample $F_n[\boldsymbol{y}_n, \boldsymbol{x}_{n-1}] \rightsquigarrow W_n, \boldsymbol{x}_n$ by partially conditioning on $\boldsymbol{x}_{n-1}$ at each generation. Since $\boldsymbol{x}_n$ is a sample from the prior, this results in an importance weight [Wood et al., 2014]

$$w_n^l = \frac{p(\boldsymbol{y}_n \,|\, F_n[\boldsymbol{x}_n^l])}{p(\boldsymbol{y}_{n-1} \,|\, F_{n-1}[\boldsymbol{x}_{n-1}^{a_n^l}])}.$$

Note that this is simply the likelihood of the $n$-th top-level `observe`. In practice we continue execution relative to $F_{n-1}[\boldsymbol{y}_{n-1}, \boldsymbol{x}_{n-1}]$ to avoid rerunning $F_n[\boldsymbol{y}_n, \boldsymbol{x}_{n-1}]$ in its entirety. The means that the inference backend must include a routine for forking multiple independent executions from a single state.

## 3.2 Iterative Conditional SMC

An advantage of SMC methods is that they provide a generic strategy for joint proposals in high dimensional spaces. An importance sampling scheme where $F[\boldsymbol{y}] \rightsquigarrow W, \boldsymbol{x}$ draws from the prior has a vanishingly small probability of generating a high-weight sample. By sampling the smallest possible set of variables $\boldsymbol{x}_n^l$ at each generation and selecting ancestors $a_{n+1}^l$ according to the likelihood of the next observed data point, we ensure that $\boldsymbol{x}_{n+1}^l$ is sampled conditioned on high-weight values of $\boldsymbol{x}_n$ from the previous generation. At the same time this strategy has a drawback: each time the particle set is resampled, the number of unique values at previous generations decreases, typically resulting in coalescence to a single common ancestor in $O(L \log L)$ generations [Jacob et al., 2013].

In many applications it is not practically possible (due to memory requirements) to set $L$ to a value large enough to guarantee a sufficient number of independent samples at all generations. In such cases, particle variants of MCMC techniques [Andrieu et al., 2010] can be used to combine samples from multiple SMC sweeps. An iterated conditional SMC (ICSMC) sampler repeatedly selects a retained particle $k$ with probability $w_N^k / \sum_{k'} w_N^{k'}$, and then performs a conditional SMC (CSMC) sweep, where the resampling step is conditioned on the inclusion of the retained particle at each generation. Formally, this procedure is a partially collapsed Gibbs sampler that targets a density $\phi(\boldsymbol{x}, a, k)$ on an extended space

$$\phi(\boldsymbol{x}_{1:N}^{1:L}, a_{2:N}^{1:L}, k) = \frac{w_N^k}{\sum_{k'} w_N^{k'}} \prod_{l=1}^{L} p(\boldsymbol{x}_1^l \mid F_1) \prod_{n=2}^{N} \prod_{l=1}^{L} \frac{w_{n-1}^{a_n^l}}{\sum_{l'} w_{n-1}^{l'}} p(\boldsymbol{x}_n^l \mid F_n[\boldsymbol{x}_{n-1}^{a_n^l}]) \ .$$

We use the shorthand $\boldsymbol{x}^k = \boldsymbol{x}_{1:N}^{b_1:b_N}$ and $a^k = a_{2:N}^{b_2:b_N}$ to refer to the sampled values and ancestor indices of the retained particle, whose index $b_n$ at each generation can be recursively defined via $b_N = k$ and $b_{n-1} = a_n^{b_n}$. The notation $\boldsymbol{x}^{-k}$ and $a^{-k}$ refers to the complements where the retained particle is excluded. An ICSMC sampler iterates between two updates

1. $\{\boldsymbol{x}^{*,-k}, a^{*,-k}\} \sim \phi(\boldsymbol{x}^{-k}, a^{-k} \mid \boldsymbol{x}^k, a^k, k)$
2. $k^* \sim \phi(k \mid \boldsymbol{x}^{*,-k}, a^{*,-k}, \boldsymbol{x}^k, a^k)$

If we interpret $\boldsymbol{x}^{-k}$, $a$ and $k$ as auxilliary variables, then the marginal on $\boldsymbol{x}_N^k$ leaves the density $\gamma_N(\boldsymbol{x}_N)$ invariant [Andrieu et al., 2010].

The advantage of ICSMC samplers is that the target space is iteratively explored over subsequent CSMC sweeps. A disadvantage is that consecutive sweeps often yield partially degenerate particles, since many newly generated particles will coalesce to the retained particle with high probability. For this reason ICSMC samplers mix poorly when the number of particles is not large enough to generate at least two completely independent lineages in a single CSMC sweep.

### 3.3 Particle Gibbs with Ancestor Sampling

PGAS is a technique that augments the CSMC sweep with a resampling procedure for the index $a_n^{b_n}$ of the retained particle [Lindsten et al., 2012]. At a high level, this sampling scheme performs two updates

1. $\{\boldsymbol{x}^{*,-k}, a^*\} \sim \phi(\boldsymbol{x}^{-k}, a \mid \boldsymbol{x}^k, k)$
2. $k^* \sim \phi(k \mid \boldsymbol{x}^{*,-k}, a^{*,-k}, \boldsymbol{x}^k, a^k)$

Here update 1 differs from the normal CSMC update in that it samples a complete set of ancestor indices $a^*$, not the complement to the retained indices $a^{*,-k}$. At each generation the ancestor $a_n^{b_n}$ of the retained particle is resampled according to a weight

$$w_{n-1|N}^l = w_{n-1}^l p(\boldsymbol{y}_N, \boldsymbol{x}_N^k[\boldsymbol{x}_n^l] \mid F_N[\boldsymbol{y}_n, \boldsymbol{x}_n^l]).$$

Here $\boldsymbol{x}_N^k[\boldsymbol{x}_n^l]$ denotes the substitution of $\boldsymbol{x}_n^l$ into $\boldsymbol{x}_N^k$, which is defined as a mapping $\boldsymbol{x}_N^k[\boldsymbol{x}_n^l] : \mathcal{A}_N^k \cup \mathcal{A}_n^l \to \mathcal{V}$ where entries in $\boldsymbol{x}_n^l : \mathcal{A}_n^l \to \mathcal{V}$ augment or replace entries in $\boldsymbol{x}_N^k : \mathcal{A}_N^k \to \mathcal{V}$.

Intuitively, ancestor resampling can be thought of as proposing new program executions by complementing random variables $\boldsymbol{x}_n^l$ of a partial execution, or prefix, with retained values from $\boldsymbol{x}_N^k$ to specify the future of the execution, or suffix. This step is performed at each generation, allowing the retained lineage to potentially be resampled many times in the course of one sweep.

Incorporation of the ancestor resampling step results in the following updates at each $n = 2, \ldots, N$

1a. Update the retained particle

$$a_n^{*,b_n} \sim R(a_n \mid w_{n-1|N}^*)$$
$$\boldsymbol{x}_n^{*,b_n} \leftarrow \boldsymbol{x}_n^{b_n}$$

1b. Update the particles for $l \in \{1, ..., L\} \setminus b_n$

$$a_n^{*,l} \sim R(a_n \mid w_{n-1}^*)$$
$$\boldsymbol{x}_n^{*,l} \sim p(\boldsymbol{x}_n \mid F_n[\boldsymbol{x}_{n-1}^{*,a_n^{*,l}}])$$

## 4 Rescoring Probabilistic Programs

PGAS for probabilistic programs requires calculation of $p(\boldsymbol{y}_N, \boldsymbol{x}_N^k[\boldsymbol{x}_n^l] \mid F[\boldsymbol{y}_n, \boldsymbol{x}_n^l])$. This probability is, by definition, the importance weight of $F_N[\boldsymbol{y}_N, \boldsymbol{x}_N^k[\boldsymbol{x}_n^l]]$ and can therefore in principle be obtained by executing this re-conditioned form. The main drawback of this naive approach is that it requires $LN$ evaluations of the program in its entirety. This results in an $O(LN^2)$ computational cost, which quickly becomes prohibitively expensive as the number of generations $N$ increases. A second complicating factor is that naively rewriting part of the execution history of the program may not yield a set of random values $\boldsymbol{x}_N^k[\boldsymbol{x}_n^l]$ that could be generated by running $F_N[\boldsymbol{y}_N]$.

We here develop a formalism that allows regeneration of a self-consistent program execution starting from a partial execution with random variables $\boldsymbol{x}_n^l$, assuming all future random samples are inherited from retained values $\boldsymbol{x}_N^k$. To do so we introduce the notion of a trace, a data structure that annotates each evaluation

with information necessary to re-execute the expression relative to a new program state. Given a trace for the suffix, i.e. the remaining top-level statements in a program, it becomes possible to re-execute conditioned on future random values, in a manner that avoid unnecessary recomputation where possible.

### 4.1 Intuition

In higher order languages with recursion and memoization, regeneration is complicated by two factors:

1. The program $F_N[\boldsymbol{y}_N, \boldsymbol{x}_N^k[\boldsymbol{x}_n^l]]$ may be underconditioned, in the sense that $\boldsymbol{x}_N^k[\boldsymbol{x}_n^l]$ does not contain values for some `sample` calls that can be evaluated in its execution. It can also be overconditioned, when $\boldsymbol{x}_N^k[\boldsymbol{x}_n^l]$ contains values for `sample` calls that will never be evaluated.
2. The expression for the `stochastic` argument to each `observe` in $F_N[\boldsymbol{y}_N, \boldsymbol{x}_N^k[\boldsymbol{x}_n^l]]$ may need to be re-evaluated if it in some way depends on global variables defined in $F_n[\boldsymbol{y}_n, \boldsymbol{x}_n^l]$. Because each variable may in turn reference other variables, we must be able to reconstruct the program environment recursively in order to rescore each `observe`.

The first non-triviality arises from the existence of `if` expressions. As an example, consider the program

```
0: [assume random? (sample (flip-dist 0.5))]
1: [assume mu (if random?
                (sample (normal-dist 0.0 1.0))
                0.0)]
2: [observe (normal-dist mu 1.0) 0.1]
```

The `sample` expression in line `1` is only evaluated when the `sample` expression in line `0` evaluates to `true`. More generally, any program that contains `(sample e)` inside an `if` expression will not be guaranteed to instantiate the same random variables in cases where the predicate of the `if` expression itself depends on previously sampled values. In programming languages that lack recursion we have the option of evaluating both branches and including or excluding the associated probabilities conditioned on the predicate value. This is essentially the strategy that is employed to handle `if` expressions in Infer.NET and BUGS variants. In languages that do permit recursion, this is in general not possible. For example, the following program would require evaluation of an infinite number of branches:

```
0: [assume geom (lambda (p)
                  (if (sample (flip-dist p))
                    1
                    (+ 1 (geom p))))]
1: [observe (poisson-dist (geom 0.5)) 3]
```

In other words, we cannot in general pre-evaluate the values associated with both branches in the suffix. When a predicate in the suffix no longer takes on the same value, we have a choice of either rejecting the regenerated suffix outright, or updating it using a regeneration procedure that evaluates the newly chosen branch and removes reference to any values sampled in the invalidated branch. We here consider the former strict form of regeneration, which guarantees that the regenerated suffix references precisely the same set of sample values as before.

A second aspect that complicates rescoring is the existence of memoized procedures. As an example, consider the following infinite mixture model

```
0: [assume class-prior (crp 1.0)]
1: [assume class
     (mem (lambda (n)
       (sample class-prior)))]
2: [assume class-dist
     (mem (lambda (k)
       (normal-dist
         (sample (normal-dist 0.0 1.0))
         1.0)))]
3: [observe (class-dist (class 0)) 2.1]
4: [observe (class-dist (class 1)) 0.6]
...
N+3: [observe (class-dist (class N)) 1.2]
```

Here each `observe` makes a call to `class`, which samples an integer class label `k` from a Chinese restaurant Process (CRP). The call to `class-dist` either retrieves an existing `stochastic` value, or generates one when a new value `k` is encountered. This type of memoization pattern allows us to delay sampling of the parameters until they are in fact required in the evaluation of a top-level `observe`, and makes it straightforward to define open world models with unbounded numbers of parameters. At the same time it complicates analysis when performing rescoring. Memoized procedure calls are semantically equivalent to lazily defined variables. Programs that rely on memoization can therefore essentially define variables in a non-deterministic order. A regeneration operation must therefore dynamically determine the set of variables that need to be re-evaluated at run time.

### 4.2 Traced Evaluation

The operations that need to happen during a rescoring step are (1) the regeneration of a consistent set of global environment variables, which includes any bindings in the prefix, augmented with any bindings defined in the suffix (some of which may require re-evaluation as a result of changes to bindings in the prefix), (2) a verification that the flow control path in the suffix is consistent with the environment bindings in the prefix, and (3) the recomputation of the probabilities of any sampled and observed values whose density values depend on bindings in the prefix.

In order to make it possible to perform the above operations, we begin by introducing a set of annotations for each value $\mathtt{v}$ that is returned upon evaluation of an expression $\mathtt{e}$. In practical terms, each value in the language is *boxed* into a data structure which we call a trace. We represent a trace $\tau$ as tuple $(\mathtt{v}, \epsilon, l, \rho, \omega, \sigma, \phi)$. $\mathtt{v}$ is the value of the expression. $\epsilon$ is a partially evaluated expression, whose sub-expressions are themselves represented as traces. $l$ is the accumulated log-weight of `observes` evaluated within the expression. $\rho$ is a mapping $\{\mathtt{s} \to \tau\}$ from symbols to traces, containing the subset of the global environment variables that were referenced in the evaluation of $\mathtt{e}$. $\omega$ is a mapping $\{\alpha \mapsto (\tau, \mathtt{v}, l)\}$. It contains an entry at the address $\alpha$ of each `observe` that was evaluated in $\mathtt{e}$ and its sub-expressions. This entry is represented as a tuple $(\tau, \mathtt{v}, l)$ containing a trace of the first argument to the `observe` (which must be of the `stochastic` type), the observed value $\mathtt{v}$, and the associated log-weight $l$. Similarly $\sigma$ is a mapping $\{\alpha \mapsto (\tau, \mathtt{v})\}$ that contains an entry for each evaluated `sample` expression (which omits the associated log-weight). The last component $\phi$ is again a mapping $\{\alpha \mapsto \tau\}$ that records all traces that appear as conditions in `if` expressions and thereby influence the control flow of program execution.

We now describe the semantics of the traced evaluation $(\mathtt{e}, \alpha, R, \Lambda) \Downarrow \tau$. The evaluation operator $\Downarrow$ returns the trace $\tau$ of an expression $\mathtt{e}$ at address $\alpha$, relative to a global environment $R$ (i.e. variables defined via `assume` statements) and local bindings $\Lambda$ (i.e. variables bound in compound procedure calls), resulting in a trace $\tau = (\mathtt{v}, \epsilon, l, \rho, \omega, \sigma, \phi)$. We assume the implementation provides a standard evaluation function for primitive procedures $\mathit{eval}(\mathtt{prim}\ \mathtt{v}_1 \ldots \mathtt{v}_n) = \mathtt{v}$. We reiterate that the notation $\alpha{::}(t, p)$ denotes an evaluation address composed of a parent address $\alpha$, a type identifier $t$ and a sub-expression index $p$, where $t$ is one of `i` for `if`, `l` for `lambda`, `q` for `quote`, `a` for applications, and `b` when evaluating compound procedure bodies.

Constants $\mathtt{c}$ evaluate to:

$$(\mathtt{c}, \alpha, R, \Lambda) \Downarrow (\mathtt{c}, \mathtt{c}, 0.0, \{\}, \{\}, \{\}, \{\}) \ .$$

We call this type of trace "transparent", since it contains no references to other traces that may take on different values or probabilities in another execution.

Symbol lookups $\mathtt{s}$ in the global environment return the value of $\mathtt{s}$ stored in $R$:

$$(\mathtt{s}, \alpha, R, \Lambda) \Downarrow (\mathtt{v}, \mathtt{s}, 0.0, \{\mathtt{s} \mapsto \tau\}, \{\}, \{\}, \{\})$$
$$\text{if } R(\mathtt{s}) = \tau \text{ and } \tau = (\mathtt{v}, \_, \_, \_, \_, \_, \_) \ .$$

Lookups from the local environment are inlined:

$$(\mathtt{s}, \alpha, R, \Lambda) \Downarrow (\mathtt{v}, \epsilon, 0.0, \rho, \omega, \sigma, \phi)$$
$$\text{if } \Lambda(\mathtt{s}) = \tau \text{ and } \tau = (\mathtt{v}, \epsilon, l, \rho, \omega, \sigma, \phi) \ .$$

Calls to `sample` inherit annotations from the trace $\tau$ passed as an argument, and add an entry in $\sigma$:

$$((\mathtt{sample}\ \mathtt{e}), \alpha, R, \Lambda) \Downarrow (\mathtt{v}, \mathtt{v}, l, \rho, \omega, \sigma[\alpha \mapsto (\tau, \mathtt{v})], \phi)$$
$$\text{if } (\mathtt{e}, \alpha{::}(\mathtt{s}, 0), R, \Lambda) \Downarrow \tau = (\mathtt{v}_1, \_, l, \rho, \omega, \sigma, \phi) \text{ and}$$
$$\mathtt{v} \text{ is drawn from the } \mathtt{stochastic} \text{ process } \mathtt{v}_1 \ .$$

Calls to `observe` inherit from $\tau$ and add an entry in $\omega$:

$$((\mathtt{observe}\ \mathtt{e}_1\ \mathtt{v}_2), \alpha, R, \Lambda)$$
$$\Downarrow (\mathtt{v}_2, \mathtt{v}_2, l_1 + l_{12}, \rho, \omega[\alpha \mapsto (\tau, \mathtt{v}_2, l_{12})], \sigma, \phi)$$
$$\text{if } (\mathtt{e}_1, \alpha{::}(\mathtt{o}, 0), R, \Lambda) \Downarrow \tau = (\mathtt{v}_1, \_, l_1, \rho, \omega, \sigma, \phi))$$
$$\text{and } l_{12} = \mathcal{L}(\mathtt{v}_1, \mathtt{v}_2) \ .$$

Here $\mathcal{L}(\mathtt{v}_1, \mathtt{v}_2)$ is used to denote the log-density of value $\mathtt{v}_2$ relative to $\mathtt{v}_1$ (which must be of type `stochastic`).

Primitive procedure applications $(\mathtt{primop}\ \mathtt{e}_1\ \mathtt{e}_2)$ evaluate to $\mathit{eval}(\mathtt{prim}\ \mathtt{v}_1\ \mathtt{v}_2)$ where $\mathtt{v}_i$ is the result of evaluating $\mathtt{e}_i$:

$$((\mathtt{prim}\ \mathtt{e}_1\ \mathtt{e}_2), \alpha, R, \Lambda)$$
$$\Downarrow (\mathit{eval}(\mathtt{prim}\ \mathtt{v}_1\ \mathtt{v}_2), (\mathtt{prim}\ \tau_1\ \tau_2), l, \rho, \omega, \sigma, \phi) \ ,$$

if $(\mathtt{e}_i, \alpha{::}(\mathtt{p}, i), R, \Lambda) \Downarrow \tau_i$, and $\rho$, $\omega$, $\sigma$, and $\phi$ are obtained by merging the corresponding components of the $\tau_i$, and the log-density $l$ is the sum of the $l_i$.

Application of a single-argument compound procedure (i.e. closure) $\mathtt{e}_1$ leads to the evaluation of the body $\mathtt{e}$ of the procedure relative to the environments $R_1, \Lambda_1$ in which the compound procedure was defined:

$$((\mathtt{e}_1\ \mathtt{e}_2), \alpha, R, \Lambda) \Downarrow (\mathtt{v}, (\tau_1\ \tau_2), l, \rho, \omega, \sigma, \phi)$$

if

$$(\mathtt{e}_1, \alpha{::}(\mathtt{a}, 0), R, \Lambda) \Downarrow \tau_1,$$
$$\tau_1 = ((\mathtt{lambda}\ (\mathtt{s})\ \mathtt{e}), R_1, \Lambda_1), \_, \_, \_, \_, \_, \_),$$
$$(\mathtt{e}_2, \alpha{::}(\mathtt{a}, 1), R, \Lambda) \Downarrow \tau_2 = (\mathtt{v}_2, \_, \_, \_, \_, \_, \_),$$
$$(\mathtt{e}, \alpha{::}(\mathtt{b}, 0), R_1, \Lambda_1[\mathtt{s} \mapsto \tau_2]) \Downarrow \tau_3 = (\mathtt{v}, \_, \_, \_, \_, \_, \_),$$

and $l, \rho, \omega, \sigma, \phi$ are obtained by combining the corresponding components of the $\tau_i$. The case with multiple arguments is defined similarly.

Quote expressions $(\mathtt{quote}\ \tau)$ simply return $\tau$:

$$((\mathtt{quote}\ \mathtt{e}), \alpha, R, \Lambda) \Downarrow \tau \quad \text{if } (\mathtt{e}, \alpha :: (\mathtt{q}, 0), R, \Lambda) \Downarrow \tau \ .$$

Finally, an `if` expression $(\mathtt{if}\ \mathtt{e}\ \mathtt{e}_1\ \mathtt{e}_2)$ returns either the result of evaluating $\mathtt{e}_1$ or $\mathtt{e}_2$. We show only the case where the `true` branch is taken:

$$((\mathtt{if}\ \mathtt{e}\ \mathtt{e}_1\ \mathtt{e}_2), \alpha, R, \Lambda) \Downarrow (\mathtt{v}, \epsilon, l, \rho, \omega, \sigma, \phi[\alpha \mapsto \tau])$$

if

$$(\mathtt{e}, \alpha{::}(\mathtt{i}, 0), R, \Lambda) \Downarrow \tau = (\mathtt{true}, \_, \_, \_, \_, \_, \_) \ ,$$
$$(\mathtt{e}_1, \alpha{::}(\mathtt{i}, 1), R, \Lambda) \Downarrow \tau_1 = (\mathtt{v}, \epsilon, \_, \_, \_, \_, \_) \ ,$$

and $l, \rho, \omega, \sigma, \phi$ are obtained by combining the corresponding components of $\tau$ and $\tau_1$. The other case is that the value of $\tau$ is `false`, and has the semantics similar to the one above.

### 4.3 Regeneration and Rescoring

We now define an operation $\mathcal{R}(\tau, R) = \tau'$ that regenerates a traced value relative to an environment $R$. This operation performs the following steps:

1. Re-evaluate predicates: Compare $\tau = \phi(\alpha)$ to $\tau' = \mathcal{R}(\tau, R)$ for all $\alpha$. Abort if $\tau'$ and $\tau$ have different values $\mathtt{v}$. Otherwise update $\phi[\alpha \mapsto \tau']$.
2. Re-score `observe` expressions and statements: Let $(\tau, \mathtt{v}, l) = \omega(\alpha)$, and $\tau' = \mathcal{R}(\tau, R)$. If $\tau'$ and $\tau$ have different values $\mathtt{v}'_0$ and $\mathtt{v}_0$, recalculate $l' = \mathcal{L}(\mathtt{v}'_0, \mathtt{v})$ and update $\omega[\alpha \to (\tau', \mathtt{v}, l')]$. Otherwise, update $\omega[\alpha \to (\tau', \mathtt{v}, l)]$.
3. Re-score samples: Let $(\tau, \mathtt{v}) = \sigma(\alpha)$, and $\tau' = \mathcal{R}(\tau, R)$. Calculate $l' = \mathcal{L}(\tau', \mathtt{v})$ and update $\omega[\alpha \mapsto (\tau', \mathtt{v}, l')]$.
4. Regenerate the environment bindings: For all symbols $\mathtt{s}$ that do not exist in $R$, let $\tau' = \mathcal{R}(\rho(\mathtt{s}), R)$ and update $R[\mathtt{s} \mapsto \tau']$ and $\rho[\mathtt{s} \mapsto \tau']$. For existing symbols update $\rho[\mathtt{s} \mapsto R(\mathtt{s})]$. We for convenience assume that $R$ is updated in place, though this may be avoided by having $\mathcal{R}$ return a tuple $(R', \tau')$.
5. If any bindings were changed with new values in step 4, regenerate all sub-expressions $\tau_i$ in $\epsilon$ to reconstruct $\epsilon'$.
6. If $\epsilon'$ was reconstructed in step 5, evaluate $\epsilon'$ and update $\mathtt{v}$ to the result of this evaluation.

Rescoring an individual trace may be performed as part of the regeneration sweep by calculating a difference in log-density $\Delta l$. This $\Delta l$ is the sum of all terms $l' - l$ in step 2 and all terms $l'$ in step 3, and any $\Delta l'$ values return from recursive calls to $\mathcal{R}$.

We have omitted a few technical details in this high-level description. The first is that we build a map $\mathcal{C} = \{\tau \to \tau', \ldots\}$ on a call to $\mathcal{R}$, which is passed as an additional argument in recursive calls to $\mathcal{R}$, effectively memoizing the computation relative to a given initial environment $R$. This reduces the computation on recursive calls, which potentially expand the same traces $\tau$ many times as sub-expressions of $\epsilon$.

### 4.4 Ancestor Resampling

Given an implementation of a traced evaluator and a regenerating/rescoring procedure $\mathcal{R}$, an implementation for PGAS in probabilistic programs becomes straightforward. We represent the programs $F_n[\boldsymbol{y}_n, \boldsymbol{x}^l_n]$ evaluated up to the first $n$ top-level statements as pairs $(R^l_n, \tau^l_n)$. We now construct a concatenated suffix $\mathcal{T}_n$ by recursively re-evaluating $\mathcal{T}_n = (\mathtt{cons}\ \tau^{b_n}_n\ \mathcal{T}_{n+1})$. For $n = 1, \ldots, N$, we then define

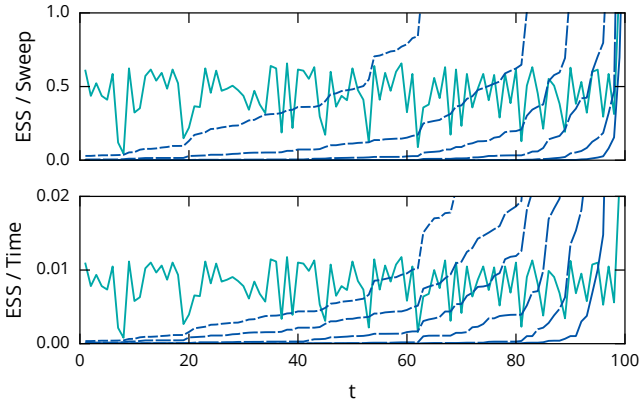


Figure 1: Effective sample size as a function of $t$, normalized by the number of PMCMC sweeps (top) and wall time in seconds (bottom). PGAS with 10 particles is shown in cyan. ICSMC results are shown in blue, with shorter dashes representing increasing particle counts 10, 20, 50, 100, 200, and 500. All lines show the median ESS over 25 independent restarts.

$p(\boldsymbol{y}_N, \boldsymbol{x}^k_N[\boldsymbol{x}^{*,l}_n] \,|\, F_N[\boldsymbol{y}_n, \boldsymbol{x}^l_n])$ as the log-weight of the rescored trace $\mathcal{R}(\mathcal{T}^l_n, R^l_{n-1})$. Note here that by construction, we may extract $\mathcal{T}_{n+1}$ from $\mathcal{T}_n$ without additional computation.

## 5 Experiments

To evaluate the mixing properties of PGAS relative to ICSMC, we consider a linear dynamical system (i.e. a Kalman smoothing problem) with a 2-dimensional latent space and a D-dimensional observational space,

$$\boldsymbol{z}_t \sim \mathrm{Norm}(\boldsymbol{A} \cdot \boldsymbol{z}_{t-1}, \boldsymbol{Q}), \quad \boldsymbol{y}_t \sim \mathrm{Norm}(\boldsymbol{C} \cdot \boldsymbol{z}_t, \boldsymbol{R}).$$

We impose additional structure by assuming that the transition matrix $\boldsymbol{A}$ is a simple rotation with angular velocity $\omega$, whereas the transition covariance $\boldsymbol{Q}$ is a diagonal matrix with constant coefficient $q$,

$$\boldsymbol{A} = \begin{bmatrix} \cos\omega & -\sin\omega \\ \sin\omega & \cos\omega \end{bmatrix}, \qquad \boldsymbol{Q} = q\boldsymbol{I}_2.$$

We simulate data with $D = 36$ dimensions, $T = 100$ time points, $\omega = 4\pi/T$, $q = 0.1$, $\alpha = 0.1$, and $r = 0.01$. We now consider an inference setting where $\boldsymbol{C}$ and $\boldsymbol{R}$ are assumed known and estimate the state trajectory $\boldsymbol{z}_{1:T}$, as well as the parameters of the transition model $\omega$ and $q$, which are given mildly informative priors, $\omega \sim \mathrm{Gamma}(10, 2.5)$ and $q \sim \mathrm{Gamma}(10, 100)$.

While this is a toy problem where an expectation maximization (EM) algorithm could likely be derived, it is illustrative of the manner in which probabilistic programs can extend models by imposing additional structure, in this case the dependency of $\boldsymbol{A}$ on $\omega$. This

modified Kalman smoothing problem can be described in a small number of program lines

```
[assume C [...]] ; assumed known
[assume R [...]] ; assumed known
[assume omega (* (sample (gamma-dist 10. 2.5))
                 (/ pi T))]
[assume A [[(cos omega) (* -1 (sin omega))]
           [(sin omega) (cos omega)       ]]]
[assume q (sample (gamma-dist 10. 100.))]
[assume Q (* (eye 2) q)]
[assume x
  (mem (lambda (t)
    (if (< t 1)
      [1. 0.]
      (sample (mvn-dist (mmul A (x (dec t))) W)))))]
[observe (mvn (mmul C (x 1)) R) [...]]
...
[observe (mvn (mmul C (x 100)) R) [...]]
[predict omega]
[predict q]
```

Here `[...]` refers to a vector or matrix literal.

We compare results for PGAS with 10 particles to ICSMC with $10, 20, 50, 100, 200$, and 500 particles. In each case we run 100 PMCMC sweeps and 25 restarts with different random seeds. To characterize mixing rates we calculate the effective sample size (ESS) of the aggregate sample set $\{w_t^{s,l}, \boldsymbol{z}_t^{s,l}\}$ over all sweeps $s = \{1, \ldots, 100\}$,

$$\mathrm{ESS}_t = \frac{1}{\sum_k (V_t^k)^2}, \quad V_t^k = \sum_{s=1}^{100} \sum_{l=1}^{L} w_t^{s,l} I[\boldsymbol{z}_t^k = \boldsymbol{z}_t^{s,l}].$$

Here $V_t^k$ represents the total importance weight associated with each unique value $\boldsymbol{z}_t^k$ in $\{\boldsymbol{z}_t^{s,l}\}$.

Figure 1 shows the ESS as a function of $t$. ICSMC shows a decreasing ESS as $t$ approaches 0, indicating poor mixing for values sampled at early generations. PGAS, in contrast, exhibits an ESS that fluctuates but is otherwise independent of $t$. ESS estimates varied approximately 15% relative to the mean across independent restarts. This suggests that fluctuations in the ESS reflect variations in the prior probability of latent transitions $p(\boldsymbol{z}_t \mid \boldsymbol{z}_{t-1}, \boldsymbol{A})$.

For this model, our PGAS implementation with 10 particles has a computational cost per sweep comparable to that of ICSMC with 300 particles. However, when we consider the ESS per computation time, the increases in mixing efficiency outweigh increases in computational cost for state estimates below $t \simeq 50$. This is further illustrated in Figure 2, which shows the standard deviation of sample estimates of the parameters $\omega$ and $q$. PGAS shows better convergence per sweep, particularly for estimates of $q$. ICSMC with 500 particles performs similarly to PGAS with 10 particles when estimating $\omega$, though ICSMC with 500 particles notably has a higher cost per sweep.

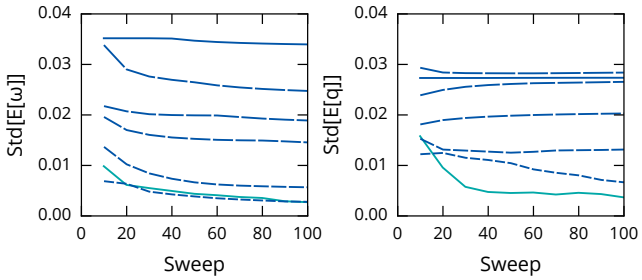


Figure 2: Standard deviation of posterior estimates $E[\omega]$ and $E[q]$ over 25 independent restarts, as a function of the PMCMC sweep. PGAS results are shown in cyan, ICSMC results are shown in blue, with shorter dashes indicating a larger particle count.

## 6 Discussion

Relative to other PMCMC methods such as ICSMC, PGAS methods have qualitatively different mixing characteristics, particularly for variables sampled early in a program execution. Implementing PGAS in the context of probabilistic programs poses technical challenges when programs can make use of recursion and memoization. The technique for traced evaluation developed here incurs an additional computational overhead, but avoids unnecessary recomputation during regeneration. When the cost of recomputation is large this will result in computational gains relative to a naive implementation that re-executes the suffix fully. Note that our approach tracks upstream, not downstream dependencies. In other words, we know what environment variables affect the value of a given expression, but not which expressions in a suffix depend on a given variable. All referenced symbol values must therefore be checked during regeneration, which can require a $O(LN^2)$ computation in itself. Further gains could be obtained constructing a downstream dependency graph for the suffix, allowing more targeted regeneration via graph walk techniques analogous to those employed in Venture [Mansinghka et al., 2014]. At the same time, the empirical results presented here are indicative of the fact that, even without these additional optimizations, PGAS can easily yield better statistical results in cases where the parameter space is large and ICSMC sampling fails to mix.

## 7 Acknowledgements

We would like to thank our anonymous reviewers, as well as Brooks Paige and Dan Roy for their comments on this paper. JWM was supported by Google and Xerox. HY was supported by the EPSRC. FW and VKM were supported under DARPA PPAML. VKM was additionally supported by the ARL and ONR.

## References


Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.

Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential Monte Carlo samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(3):411–436, June 2006. ISSN 1369-7412. doi: 10.1111/j.1467-9868.2006.00553.x.

Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. In *Proc. 24th Conf. Uncertainty in Artificial Intelligence (UAI)*, pages 220–229, 2008.

Pierre E. Jacob, Lawrence M. Murray, and Sylvain Rubenthaler. Path storage in the particle filter. *Statistics and Computing*, December 2013. ISSN 0960-3174. doi: 10.1007/s11222-013-9445-x.

Fredrik Lindsten, Michael I Jordan, and Thomas B. Schön. Ancestor Sampling for Particle Gibbs. *Neural Information Processing Systems*, 2012.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv*, page 78, 2014. URL `http://arxiv.org/abs/1404.0099`.

T Minka, J Winn, J Guiver, and D Knowles. Infer .NET 2.4, Microsoft Research Cambridge, 2010.

Brooks Paige and Frank Wood. A Compilation Target for Probabilistic Programming Languages. *International Conference on Machine Learning (ICML)*, 32, 2014.

Stan Development Team. Stan: A C++ library for probability and sampling, version 2.5.0, 2014. URL `http://mc-stan.org/`.

David Wingate, Andreas Stuhlmueller, and Noah D Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the 14th international conference on Artificial Intelligence and Statistics*, page 770_778, 2011.

F Wood, JW van de Meent, and V Mansinghka. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*, pages 1024–1032, 2014.